Mario Zechner

Beginning Android Games

Get started with game apps development for the Android platform

Apress®

Марио Цехнер

Программирование игр под Android

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2013

Марио Цехнер

Программирование игр под Android

Перевели с английского Егор Сидорович и Евгений Зазноба

Заведующая редакцией	*К. Галицкая*
Руководитель проекта	*Д. Виницкий*
Ведущий редактор	*Е. Каляева*
Научные редакторы	*О. Трафимович, Ю. Заровский*
Художник	*К. Радзевич*
Корректоры	*О. Андросик, Е. Павлович*
Верстка	*Г. Блинов*

ББК 32.973.2-018.2
УДК 004.451

Цехнер Марио

П78 Программирование игр под Android. — СПб.: Питер, 2013. — 688 с.: ил.

ISBN 978-5-459-01554-6

Из данной книги вы узнаете все необходимое, чтобы стать успешным разработчиком под Android. Вы начнете обучение с фундаментальных вопросов проектирования игр и с основ программирования, а потом перейдете к созданию игрового движка и интересных игр. Этой информации вам будет достаточно, чтобы приступить к творческой работе и создавать собственные приложения для Android. В книге подробно описан весь процесс создания отличных игр для платформы Android. Вы узнаете, как настроить и использовать инструменты для разработки в Android, получите информацию о классическом программировании двухмерных игр и создании собственных завораживающих экшенов и игр-платформеров. В издании рассмотрены графика и аудио для Android, игровая механика (обнаружение соударений, физика и спрайтовая анимация), а также добавление в игры трехмерных эффектов. Кроме того, описано, как опубликовать игру, получать сообщения об отказах программы и организовать техническую поддержку для пользователей.

ISBN 978-1430230427 англ.
ISBN 978-5-459-01554-6

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 22.08.12. Формат 70x100/16. Усл. п. л. 55,470. Тираж 1500. Заказ
Отпечатано с готовых диапозитивов в ГППО «Псковская областная типография».
180004, Псков, ул. Ротная, 34.

Краткое содержание

Оглавление

Моим кумирам — маме и папе и любви моей жизни — Стефании

Введение

Добро пожаловать в мир разработки игр для платформы Android. Меня зовут Марио, я буду вашим гидом в течение следующих 14 глав. Вы здесь для того, чтобы узнать больше о разработке игр для Android, и я надеюсь, что стану тем, кто поможет вам реализовать ваши идеи.

Мы рассмотрим широкий спектр тем и вопросов: основы Android, программирование звука и графики, использование математики и физики, а также пугающую вещь под названием OpenGL ES. На основании всех этих знаний мы создадим три разные игры, одна из которых будет выполнена в 3D.

Разработка игр — несложная задача, если вы знаете, что делаете. Поэтому я старался представить материал в таком виде, чтобы вы не просто брали из него полезные блоки программного кода, но и понимали общую картину процесса разработки игр. Осознание основных принципов — ключ к реализации сложных игровых концепций. Вы сможете не только создавать игры, похожие на примеры из этой книги, но получите достаточно знаний, чтобы использовать материал из Интернета и других книг для реализации ваших самых смелых идей в разработке игр.

Целевая аудитория

Эта книга адресована в первую очередь начинающим разработчикам игр. Вы можете приступать к созданию игр, не обладая каким-либо знанием о предмете. Я проведу вас с самого начала. Однако я рассчитываю на ваше (пусть самое небольшое) знакомство с Java. Других требований у меня нет — знакомство с Android и Eclipse также не требуется!

Это издание будет полезно и разработчикам среднего уровня, которые хотят поближе познакомиться с Android. Некоторые материалы из книги будут им хорошо известны. Однако в ней много трюков и подсказок, которые сделают книгу полезной и для них. Android иногда — вещь в себе, и это издание поможет с ней справиться.

Как построена эта книга

В книге использован интерактивный подход, благодаря которому вы постепенно, но уверенно пройдете путь от самых основ до экзотических усовершенствований (вплоть до использования аппаратного ускорения). В ходе изучения глав вы создадите повторно используемую программную базу, поэтому я советую изучать главы по порядку. Конечно, более опытные читатели могут пропускать некоторые разделы,

посвященные вещам, с которыми они хорошо знакомы. И все же будет полезно пробежаться по коду в этих главах, чтобы быть уверенными в понимании того, как классы и интерфейсы используются в следующих, более сложных разделах.

Как получить исходные коды

Эта книга построена по принципу «все в одном»; весь необходимый для запуска примеров и игр код содержится в ней самой. Но копировать листинги из книги в Eclipse — не самая удачная мысль. Кроме того, игры состоят не только из кода, но и из активов, которые вы не сможете скопировать из книги. Процесс набора кода из книги в Eclipse может привести к появлению ошибок. Роберт (технический рецензент книги) и я позаботились о том, чтобы в листингах этой книги не было ошибок, но барабашки не дремлют.

Чтобы облегчить вам процесс, я создал проект в Google Code, содержащий следующие элементы.

- Весь программный код и активы под лицензией GPL 3, доступные из репозитория Subversion проекта.
- Небольшое текстовое руководство, описывающее процесс импорта проектов в Eclipse, а также видеоролик, демонстрирующий то же самое.
- Трекер возникающих проблем, с помощью которого вы можете сообщать обо всех ошибках в самой книге или в программном коде. После помещения описания проблемы в трекере я смогу разместить исправленные версии в репозитории Subversion. Таким образом, вы всегда будете обладать актуальной, лишенной ошибок (надеюсь) версией кода из данной книги, которой смогут пользоваться и другие читатели.
- Группа обсуждения книги, вступить в которую может каждый. Я, конечно, тоже буду там.

Для каждой главы, содержащей программный код, в репозитории Subversion имеется аналогичный проект Eclipse. Проекты независимы друг от друга, поскольку в каждой последующей главе мы наращиваем созданный в предыдущих главах функционал. Код для глав 5 и 6 содержится в проекте `ch06-mrnom`.

Google Code-проект размещен по адресу `http://code.google.com/p/beginning-android-games`.

Об авторе

В настоящее время Марио Цехнер работает программистом в компании R&D, а по вечерам преображается в самозабвенного разработчика игр. Он выпускает их под маркой Badlogic Games. Он написал игру Newton для Android, Quantum для Windows, Linux и Mac OS X, а также массу других небольших игр. В настоящее время он работает над свободным кроссплатформенным решением для разработки игр, которое называется libgdx. Кроме программирования он также активно занимается подготовкой руководств и написанием статей, посвященных разработке игр. Эти материалы можно найти в Интернете, в частности в блоге Марио (http://www.badlogicgames.com/wordpress).

О техническом редакторе

Роберт Грин — независимый разработчик видеоигр из Портленда, штат Орегон, США. Свои работы он выпускает под маркой Battery Powered Games. Он написал шесть игр для Android, в том числе Deadly Chambers, Antigen, Wixel, Light Racer и Light Racer 3D. Прежде чем полностью посвятить себя разработке мобильных игр, Роберт трудился в софтверных компаниях в Миннеаполисе и Чикаго, в том числе в IBM Interactive. В настоящее время Роберт занимается кроссплатформенной разработкой игр и проблемами повышения производительности, связанными с мобильными играми.

Благодарности

Хотелось бы поблагодарить команду Apress, так как без них эта книга бы не состоялась. В частности, искренне благодарю Кэндис Инглиш и Адама Хиза, моих потрясающих редакторов, которые неустанно отвечали на все мои наивные вопросы. Мэтью Муди помог мне правильно структурировать разделы, давал ценнейшие подсказки и предложения, благодаря которым книга стала значительно лучше. Благодарю Деймона Ларсона и Джеймса Комптона, великодушных людей, исправивших мои многочисленные грамматические ошибки. Спасибо, ребята, работать с вами было исключительно приятно.

Спасибо моему дорогому Роберту Грину, который выполнил техническую редактуру этой книги. Он позаботился о том, чтобы мои искания были описаны технически верно. Кроме того, он стал моим мальчиком для битья — если читатели найдут в книге ошибки или недочеты, значит, их не нашел Роберт.

Еще остается поблагодарить всех моих друзей по всему миру, которые делились со мной идеями и отзывами — иногда было так интересно, что я не замечал, что снова засиделся за работой до трех утра. Мои личные благодарности адресованы Натану Свиту, Дэйву Клейтону, Дэйву Фраска, Морицу Посту, Кристофу Уидуллу и Тому Вонгу — эти программеры-спецы трудятся вместе со мной над libgdx; Джону Филу и Али Мосавиану — вместе с ними мы работали программистами в Швеции. Кроме того, благодарю Романа Керна и Маркуса Мура — коллег по моей нынешней работе.

И наконец, огромной благодарности достойна Стефания, моя любимая, которая часто ночевала одна, пока я работал над книгой, и нашла в себе силы с этим мириться. Кроме того, она простила мне мою излишнюю раздражительность. Спасибо!

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты halickaya@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

1 Android: новенький в классе

Будучи ребенком начала 1990-х, я рос вместе со своей верной приставкой Nintendo Game Boy. Я потратил бесчисленные часы, помогая Марио спасти принцессу, пытаясь улучшить свой результат в «Тетрисе» и гоняясь со своими друзьями в RC Pro-Am с джойстиком в руках. Я брал этот потрясающий гаджет с собой, куда бы ни шел. Пристрастие к играм вызвало у меня желание создавать собственные миры и делиться ими со своими друзьями. Я начал программировать на PC, но вскоре обнаружил, что не могу закачать свои шедевры на Game Boy. Мой программистский энтузиазм начал угасать. Да еще и Game Boy сломался...

Перенесемся в 2010 год. Смартфоны становятся новой мобильной игровой платформой, соревнуясь с классическими карманными игровыми системами вроде Nintendo DS или Playstation Portable. Мой интерес к этой теме вернулся, и я начал исследовать, соответствуют ли мобильные платформы моим потребностям разработчика. iOS от Apple выглядела хорошим кандидатом для разработки. Однако я быстро обнаружил, что это не открытая система — я мог делиться своими продуктами с друзьями, только если Apple позволит мне это, к тому же для разработки нужен был Mac. Затем я обнаружил платформу Android и немедленно в нее влюбился. Ее среда разработки работает на всех крупных платформах без каких-то ограничений. У нее есть живое сообщество разработчиков, в котором всегда будут рады вам помочь решить проблему, а также предоставить исчерпывающую документацию. Я могу делиться своими играми с кем угодно без какой-либо платы за это, а если захочу на этом заработать, то могу опубликовать свое творение на глобальном ресурсе с миллионами пользователей за считанные минуты.

Единственное, что мне оставалось неясным, — как писать игры для Android и как воспользоваться моими знаниями о разработке игр для PC для этой новой платформы. В последующих главах я поделюсь с вами своим опытом и помогу вам начать разрабатывать игры для Android. У меня есть и дополнительная, эгоистичная цель — я хочу, чтобы было больше игр, в которые я мог бы поиграть!

Итак, начнем знакомство с нашим новым другом — Android.

Краткая история Android

Первое громкое упоминание Android прозвучало в 2005 году, когда Google приобрел маленький стартап-проект Android, Inc. Это действие вызвало множество спекуляций на тему выхода Google на мобильный рынок. Конец слухам положил в 2008 году релиз Android 1.0, после чего Android стала новым игроком на самом перспективном рынке. С тех пор идет битва между ней и уже устоявшимися платформами вроде iOS (ранее известной как iPhone OS) и Blackberry, причем шансы новичка выглядят весьма хорошими.

Поскольку Android — проект с открытым исходным кодом, барьер вхождения для производителей устройств довольно низок. Они могут выпускать аппараты для любых ценовых сегментов и модифицировать систему для ее лучшего взаимодействия с конкретными устройствами. Таким образом, область применения Android не ограничивается смартфонами высшего ценового уровня, что делает ее потенциальную аудиторию весьма широкой.

Ключевой составляющей успеха Android стало образование Open Handset Alliance (OHA) в конце 2007 года. В OHA входят такие компании, как HTC, Qualcomm, Motorola и NVIDIA, которые сотрудничают в разработке открытых стандартов для мобильных устройств. Хотя ядро Android создано в основном в Google, все члены OHA вносят в него свой вклад в той или иной форме.

Android, по сути, представляет собой мобильную операционную систему (ОС) и платформу, основанную на ядре Linux версии 2.6, свободную для использования в коммерческих и некоммерческих целях. Многие члены OHA собирают собственные версии Android для своих устройств с измененным пользовательским интерфейсом (User Interface, UI), например HTC Sense или Motorola MOTOBLUR. Открытая природа Android позволяет фанатам делать собственные сборки. Они обычно называются модами, прошивками или ROM. Самая известная прошивка на момент написания книги разработана парнем, известным как Cyanogen, и предназначена для оснащения новейшими улучшениями всех Android-устройств.

Со времени своего первого релиза в 2008 году Android получила семь обновлений версии, каждая из которых названа по имени какого-нибудь десерта (за исключением неактуальной сейчас Android 1.1). Каждая версия добавляла новую функциональность, в том числе и значимую для разработчиков игр. В версии 1.5 (Cupcake) была добавлена поддержка включения собственных библиотек в приложения Android — ранее они должны были быть написаны исключительно на Java. Собственный код может быть очень полезен в ситуациях, когда производительность превыше всего. Версия 1.6 (Donut) получила поддержку различных разрешений экрана. Мы будем обсуждать этот факт неоднократно, поскольку он имеет некоторое влияние на процесс разработки игр для Android. С версии 2.0 (Eclair) стали поддерживаться мультитач-дисплеи, а версия 2.2 (Froyo) может похвастаться своевременной компиляцией (Just-in-Time, JIT) в виртуальной машине Dalvik, что заметно усилило мощь Java-приложений в Android. JIT значительно ускоряет

выполнение приложений Android — в некоторых сценариях до пяти раз. На момент написания последней является версия 2.3 (Gingerbread), в которую добавлен сборщик мусора для Dalvik. Кстати, если вы еще не поняли, Android-приложения пишутся на языке Java.

В 2011 году была выпущена специальная версия Android для планшетов, названная Honeycomb и получившая индекс 3.0. Она не предназначена для запуска на мобильных телефонах, однако некоторые ее функции могут быть портированы на смартфоны. На момент написания книги версия 3.0 недоступна для общего использования и на рынке нет устройств, ее использующих. Android 2.3 может быть установлена на многих устройствах, использующих собственную прошивку. Единственный гаджет, применяющий Gingerbread, — Nexus S, продаваемый непосредственно Google.

Разделение

Гибкость Android имеет свою цену: компании, предпочитающие разрабатывать собственные пользовательские интерфейсы, вынуждены постоянно гнаться за релизами новых версий ОС. Устройства, выпущенные всего несколько месяцев назад, становятся устаревшими, поскольку операторы и производители не хотят создавать обновления ПО, чтобы пользователи могли применять новые возможности Android. Проблема, получившая название «разделение», — результат этого процесса.

Разделение многолико. Для конечного пользователя оно означает невозможность устанавливать и задействовать определенные приложения и возможности, оставаясь на старой версии Android. Для разработчика разделение выражается в необходимости поддерживать работу своих приложений для всех версий операционной системы. Приложения для старых версий обычно работают и на новых, а вот обратное утверждение неверно. Некоторые возможности, добавленные в новых релизах, безусловно, недоступны в старых (например, поддержка мультитач). Поэтому программисты вынуждены разделять код для разных версий ОС.

Однако все не так плохо. Хотя это звучит пугающе, но меры, которые необходимо предпринять, минимальны. В большинстве случаев вы вообще можете забыть об этой проблеме и считать, что существует только одна версия Android. Для нас как разработчиков игр более важны изменения аппаратной составляющей, нежели API. Тут существует другая форма разделения, также являющаяся проблемой для таких платформ, как iOS (хотя это предпочитают не произносить вслух). В этой книге я освещу вопросы, с которыми вы можете столкнуться при разработке вашей новой игры для Android.

Роль Google

Хотя официально Android — детище Open Handset Alliance, Google является очевидным лидером как в реализации самой системы, так и в создании необходимой экосистемы для ее развития.

Android Open Source Project

Достижения Google объединены под названием Android Open Source Project. Большая часть кода лицензирована под Apache License 2 открытой и неограниченной лицензией (по крайней мере по сравнению, например, с GNU General Public License (GPL)). Допускается свободное использование исходного кода для создания собственных систем. Однако системы, объявленные как совместимые с Android, должны для начала пройти Android Compatibility Program — процесс, удостоверяющий базовую совместимость со сторонними приложениями, которые созданы такими разработчиками, как мы с вами. Совместимые системы могут вливаться в экосистему Android, включающую в себя Android Market[1].

Android Market

Android Market был открыт для публики в октябре 2008 года компанией Google. Это онлайн-магазин приложений, позволяющий пользователям находить и устанавливать сторонние приложения. Магазин доступен как с помощью приложения на телефоне, как и с использованием браузера на компьютере.

Магазин позволяет независимым разработчикам предлагать свои приложения бесплатно или за деньги.

Пользователи могут получить доступ к магазину после регистрации учетной записи Google. Покупатель вправе вернуть приложение в течение 15 минут после покупки, чтобы получить свои деньги обратно. Ранее время возврата составляло 24 часа, и столь резкое его уменьшение не слишком обрадовало пользователей.

Программисту, чтобы публиковать свои приложения, необходимо зарегистрироваться в Google в качестве разработчика Android. Это будет стоить ему $25 единовременно. Уже через несколько минут после успешной регистрации становится возможным публиковать свои творения.

В Android Market отсутствует процесс утверждения опубликованного приложения, но имеется система разрешений. Пользователю перед установкой демонстрируются разрешения, которые необходимо предоставить приложению для его работы. Эти разрешения включают в себя доступ к телефонным службам, сети, карте памяти и т. д. Установка приложения совершается только после одобрения пользователем этих разрешений. Система полагается на пользователя в данном вопросе. Надо сказать, что для настольных приложений (особенно это касается Windows-систем) такой подход работает не слишком хорошо. Что касается Android — пока это эффективная мера; всего несколько приложений были удалены с Android Market по причине их вредоносного поведения.

Для продажи своих приложений разработчик должен дополнительно зарегистрировать учетную запись типа Google Checkout Merchant (это делается бесплатно). Все финансовые операции используют эту учетную запись.

---

[1] На момент подготовки русского издания книги Android Market был переименован в Google play (play.google.com). Таким образом, здесь и далее в тексте под Android Market будет подразумеваться Google play. — *Примеч. ред.*

Соревнования, распространение устройств и Google I/O

Продолжая усилия по привлечению к платформе Android новых разработчиков, Google начала проводить различные состязания. Первое из них, названное Android Developer Challenge (ADC), было проведено в 2008 году. Победившим проектам были обещаны щедрые денежные призы. ADC проводилось и в следующем году, в нем также приняло участие большое количество претендентов. В 2010 году ADC не проводилось, что может быть объяснено тем, что Android набрала необходимую базу разработчиков и поэтому необходимость в привлечении новых участников отпала.

Кроме того, Google в начале 2010 года начал программу распространения устройств. Каждый разработчик, приложение (или приложения) которого было скачано более 5000 раз при среднем пользовательском рейтинге не менее 3,5 звезды, получал новый телефон Motorola Droid, Motorola Milestone или Nexus One. Эта акция вызвала бурную реакцию сообщества разработчиков, хотя поначалу столкнулась с недоверием. Многие считали полученные по электронной почте уведомления продуманной мистификацией. К счастью, все оказалось правдой, и тысячи разработчиков по всему миру получили в подарок устройства — удачный шаг со стороны Google по привлечению новых разработчиков и поддержке уверенности в уже привлеченных.

Помимо этого Google предлагает разработчикам специальную версию аппаратов — Android Dev Phone (ADP). Первым ADP был вариант T-Mobile G1 для разработчиков (также известный как HTC Dream). Следующее поколение, ADP 2, было вариантом HTC Magic. К тому же Google продавала собственный телефон (Nexus One) конечным пользователям. Хотя изначально он не позиционировался как девелоперский, многие рассматривали его как потомка ADP 2. В итоге Google прекратила продажи Nexus One потребителям, и теперь он доступен только для ее партнеров и разработчиков. В конце 2010 года был выпущен последний на момент написания книги ADP. Этот аппарат компании Samsung, работающий на Android 2.3 (Gingerbread), называется Nexus S. ADP-устройства могут быть приобретены на Android Market (для этого необходимо иметь учетную запись разработчика). Кроме того, Nexus S можно купить отдельно через сайт Google (www.google.com/phone).

Ежегодная конференция Google I/O — мероприятие, которое каждый разработчик Android ждет, как Рождества. На ней демонстрируются новейшие и лучшие технологии и проекты Google, и Android в последние годы занимает среди них особое место. На Google I/O обычно проводится несколько сессий, связанных с Android; позже эти сессии становятся доступны на YouTube (канал Google Developer).

Возможности и архитектура Android

Android — это не просто еще один дистрибутив Linux для мобильных устройств. При разработке для Android вам, скорее всего, не придется иметь дело с сами ядром Linux. С точки зрения программиста, Android — платформа, абстрагирующая разработчика от ядра и позволяющая ему создавать код на Java. Android

обладает несколькими полезными возможностями. Во-первых, это фреймворк, предлагающий большой набор API для создания различных типов приложений и, кроме того, обеспечивающий возможности повторного использования и замены компонентов, которые предлагаются платформой и сторонними приложениями. Во-вторых, наличие виртуальной машины Dalvik, отвечающей за запуск приложений на Android. Кроме того, к услугам разработчика набор графических библиотек для 2D- и 3D-приложений, поддержка мультимедиа-форматов (Ogg Vorbis, MP3, MPEG-4, H.264, PNG), API для доступа к камере, GPS, компасу, акселерометру, сенсорному экрану, джойстику и клавиатуре. Имеется даже специальное API для воспроизведения фоновых звуковых эффектов, которое пригодится нам при разработке игр. Не все Android-устройства обладают всеми этими возможностями — налицо аппаратное разделение. Конечно, список возможностей Android не исчерпывается упомянутыми мной. Однако для разработки игр они будут наиболее важны.

Архитектура Android формируется из набора компонентов. Каждый компонент построен на основе элементов более низкого уровня. На рис. 1.1 представлен краткий обзор главных компонентов Android.

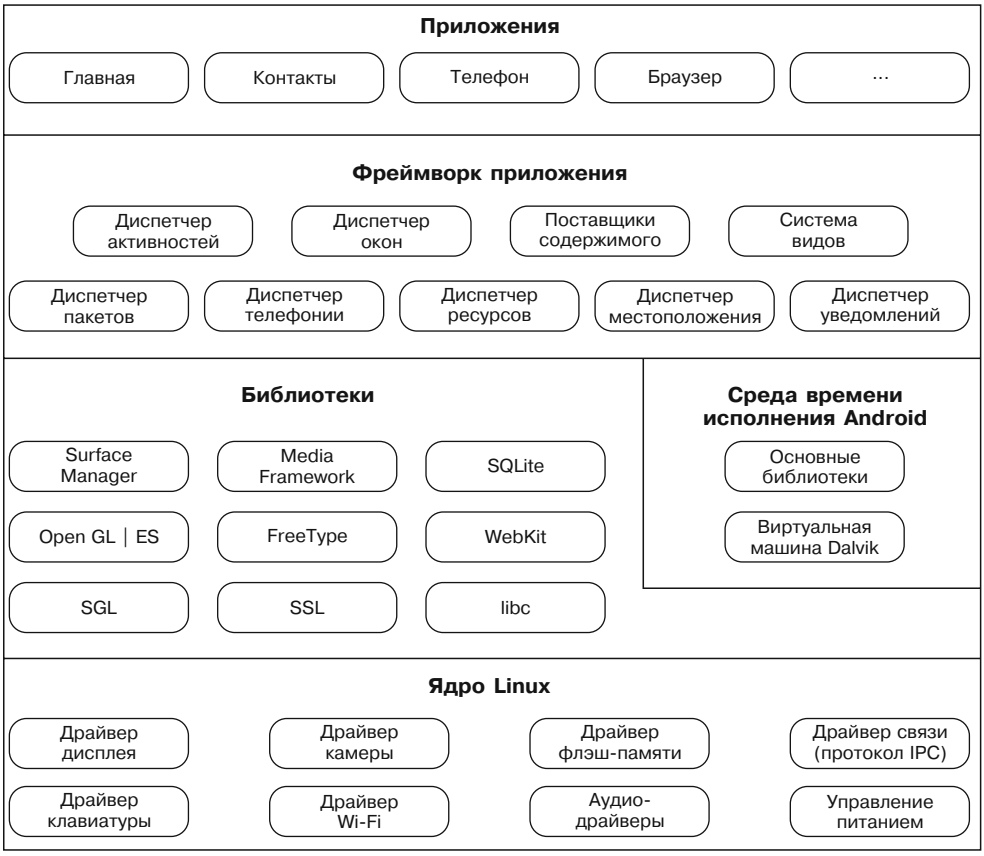

Рис. 1.1. Обзор архитектуры Android

Ядро

В нижней части рисунка вы можете увидеть, что ядро Linux предлагает основные драйверы для аппаратных компонентов системы. Кроме того, ядро отвечает за память, управление процессами, поддержку сети и т. д.

Среда выполнения и Dalvik

Среда выполнения Android, являющаяся надстройкой над ядром, отвечает за порождение и выполнение приложений Android. Каждая программа работает в собственном процессе со своей виртуальной машиной Dalvik.

Dalvik запускает программы в байт-кодовом формате DEX. Обычно вы превращаете обычные Java-файлы с расширением CLASS в формат DEX с помощью специальной утилиты dx, имеющейся в SDK. Формат DEX занимает намного меньше места в памяти, чем классические файлы типа CLASS, что достигается большим сжатием, разбиением на таблицы и слиянием нескольких CLASS-файлов. Виртуальная машина Dalvik взаимодействует с библиотеками ядра, предлагающими базовый функционал для Java-программ. Эти библиотеки располагают большим, но не полным набором классов, доступных через Java SE, и используют часть реализации Apache Harmony. Помимо прочего это означает, что в Java SE для Dalvik отсутствуют Swing и Abstract Window Toolkit, а также все классы из Java ME. Однако вы можете использовать (с осторожностью) сторонние библиотеки для Java SE на Dalvik.

До Android 2.2 (Froyo) весь код был интерпретируемым. В Froyo был представлен отслеживающий JIT-компилятор, способный компилировать части байт-кода в машинный код на лету. Это значительно увеличивает производительность приложений, требующих больших вычислений. JIT-компилятор может использовать возможности процессора, специально предназначенные для сложных вычислений, например для операций с плавающей точкой (Floating Point Unit, FPU).

Кроме того, в Dalvik включен собственный сборщик мусора (Garbage Collector, GC). Он работает по принципу «отметить и убрать», что иногда ставит разработчиков в тупик. Однако если внимательно им пользоваться, вы сможете эффективно применять его при разработке игр. Последняя версия Android (2.3) обладает улучшенным параллельным GC, работа с которым причиняет меньше неприятностей. Мы рассмотрим проблемы пользования GC позже в этой книге.

Каждое приложение, запускающееся в экземпляре виртуальной машины Dalvik, имеет в своем распоряжении от 16 до 24 Мбайт оперативной памяти. Это необходимо держать в уме, жонглируя картинками и звуковыми ресурсами.

Системные библиотеки

Помимо библиотек ядра, предлагающих некоторую функциональность Java SE, существует также набор родных библиотек на C/C++, создающих основу для фреймворка приложения (расположенного на уровень выше, чем библиотеки рис. 1.1). Эти системные библиотеки в большинстве своем отвечают за сложные

в вычислительном смысле задачи (прорисовка графики, воспроизведение звука, доступ к базе данных), не очень подходящие для виртуальной машины Dalvik. API в них обернуты с помощью классов Java во фреймворк приложения, который мы будем использовать при написании игр. В той или иной форме нам понадобятся следующие библиотеки.

- Skia Graphics Library (Skia) — этот программный визуализатор 2D-графики используется для рендеринга пользовательского интерфейса приложений Android. Он будет нужен нам при написании первой 2D-игры.
- OpenGL for Embedded Systems (OpenGL ES) — индустриальный стандарт для аппаратной прорисовки графики. OpenGL ES 1.0 и 1.1 имеются в Java во всех версиях Android. OpenGL ES 2.0, в котором появилась поддержка шейдеров, поддерживается начиная с Android 2.2 (Froyo). Нужно заметить, что Java-реализации для OpenGL ES 2.0 неполные — в них отсутствует несколько важных методов. Кроме того, эмулятор и большинство старых устройств (занимающих, однако, значительную долю рынка) не поддерживают OpenGL ES 2.0. Мы будем иметь дело с OpenGL ES 1.0 и 1.1, чтобы добиться максимальной совместимости.
- OpenCore — библиотека записи и воспроизведения аудио- и видеофайлов. Она поддерживает хороший набор форматов (Ogg Vorbis, MP3, H.264, MPEG-4 и т. д.). Нас прежде всего будет интересовать работа со звуком, которая напрямую с Java не связана, но используется через несколько классов и сервисов.
- FreeType — библиотека загрузки и обработки растровых и векторных шрифтов (в большинстве случаев формата TrueType). FreeType поддерживает стандарт Юникод, включая написание справа налево для арабских шрифтов и другие подобные случаи. Увы, это не очень согласуется с Java, не поддерживающей арабскую типографику. Как и в случае с OpenCore, FreeType напрямую в Java не реализован, но обернут в несколько подходящих классов.

Эти системные библиотеки удовлетворяют большой круг потребностей разработчиков игр и выполняют за них много тяжелой работы. Они — причина того, чтобы писать наши игры на старой доброй Java.

ПРИМЕЧАНИЕ

Хотя возможностей Dalvik обычно вполне хватает для наших потребностей, иногда необходимо больше производительности. Такое может произойти при имитации сложных физических процессов или множественных 3D-вычислениях — тогда нам придется писать собственный код. В этой книге данный аспект разработки не рассматривается. Существует набор открытых библиотек для Android, благодаря которым вы сможете реализовать это на Java. По адресу http://code.google.com/p/libgdx/ можно посмотреть пример.

Фреймворк приложения

Фреймворк приложения связывает вместе системные библиотеки и среду выполнения, создавая таким образом пользовательскую сторону Android. Фреймворк управляет приложениями и предлагает продуманную среду, в которой они работают. Разработчики создают приложения для этого фреймворка с помощью набора

программных интерфейсов на Java, охватывающих такие области, как разработка пользовательского интерфейса, фоновые службы, оповещения, управление ресурсами, доступ к периферии и т. д. Все ключевые приложения, поставляемые вместе с ОС Android (например, почтовый клиент), написаны с помощью этих API.

Приложения, будь они с интерфейсом или с фоновыми службами, могут связываться с другими приложениями. Эта связь позволяет одному приложению использовать компоненты других. Простой пример — программа, делающая фотоснимок и потом обрабатывающая его. Приложение запрашивает у системы компонент другого приложения, обеспечивающий это действие. Далее первое приложение может повторно использовать этот компонент (например, от встроенного приложения камеры или от фотогалереи). Подобный алгоритм снимает значительную часть ноши с программиста, а также позволяет настроить многообразие аспектов поведения Android.

Как разработчики игр мы будем создавать приложения с пользовательским интерфейсом в этом фреймворке. Посему нас будет интересовать архитектура приложения, его жизненный цикл, а также его взаимодействие с пользователем. Фоновые сервисы обычно играют небольшую роль в разработке игр, поэтому я не буду заострять на них внимание.

Software Development Kit

Для разработки приложений для Android мы будем использовать Android Software Development Kit (SDK). Он состоит из широкого набора инструментов, документации, утилит и примеров, которые помогут вам быстро начать работу. В него также включены Java-библиотеки, необходимые для создания приложений для Android и содержащие API для фреймворка приложения. В качестве средства разработки поддерживаются все основные операционные системы.

К основным возможностям SDK можно отнести:

- отладчик, способный отлаживать приложения, запущенные на реальном устройстве или эмуляторе;
- профиль памяти и производительности, помогающий обнаружить утечки памяти и найти неэффективный код;
- эмулятор устройства, основанный на QEMU (виртуальной машине с открытым кодом, эмулирующей различные аппаратные платформы), он довольно точен, хотя не всегда быстр;
- утилиты командной строки для связи с устройствами;
- скрипты и утилиты для создания пакетов и развертывания приложений.

SDK может быть интегрирован в Eclipse — открытую популярную и функциональную среду разработки (IDE) для Java. Эта интеграция достигается с помощью плагина Android Development Tools (ADT), добавляющего новые возможности в Eclipse для создания проектов для Android, их исполнения и отладки в эмуляторе или на устройстве, создания пакетов для их развертывания на Android Market.

Отметим, что SDK может быть интегрирован и в другие IDE (например, в NetBeans), но официальной поддержки для них нет.

ПРИМЕЧАНИЕ

В главе 2 описан процесс интеграции SDK в среду разработки Eclipse.

SDK и плагин ADT для Eclipse постоянно обновляются, в них добавляются новые возможности. Поэтому регулярно их обновлять — хорошая идея.

Любой хороший SDK должна сопровождать исчерпывающая документация. Android SDK не исключение — помимо документации с ним поставляется много примеров. Кроме того, найти руководство разработчика и полное описание API для всех модулей фреймворка приложения можно по адресу http://developer.android.com/guide/index.html.

Сообщество разработчиков

Одна из слагаемых успеха Android — ее сообщество разработчиков, объединяющее программистов по всему миру. Место наиболее массового скопления энтузиастов — группа Android Developers, расположенная по адресу http://groups.google.com/group/android-developers. Это место номер один для того, чтобы задавать вопросы или искать помощи в случае возникновения нерешаемой (как вы думаете) проблемы. Группу посещают различные разработчики Android, от системных программистов до создателей приложений и игр. Иногда ценные советы дают даже инженеры Google, отвечающие за соответствующие части Android. Регистрация на ресурсе бесплатна, и я очень рекомендую начать чтение этой группы прямо сейчас. Помимо предоставления площадки для задавания вопросов это также прекрасное место для поиска уже готовых решений проблем и ответов. Поэтому прежде, чем задать вопрос, проверьте, не был ли уже дан на него ответ.

Любое развитое сообщество разработчиков имеет свой талисман. У Linux есть пингвин Tux, у GNU есть, хм... гну, а Mozilla Firefox красуется своей лисой. Android — не исключение. В качестве ее символа был выбран маленький зеленый робот. На рис. 1.2 вы можете полюбоваться этим маленьким дьяволом.

Рис. 1.2. Безымянный талисман Android

Хотя выбор цвета талисмана может вызывать вопросы, этот безымянный маленький робот уже засветился в нескольких популярных играх для Android. Наиболее заметным его появлением было участие в Replica Island — бесплатном и открытом платформере, созданном инженером Google Крисом Прюэттом в свободное от основной работы время.

Устройства, устройства, устройства!

Android не ограничивается одной аппаратной экосистемой. Такие известные производители устройств, как HTC, Motorola и Samsung, быстро сориентировались и предлагают широкий спектр устройств, работающих на этой платформе. Помимо смартфонов на рынке в последнее время появилось много планшетов, использующих Android. Некоторые ключевые концепции соблюдаются для всех устройств, что несколько упрощает жизнь разработчика игр.

Аппаратная составляющая

Для Android-устройств не определены минимальные требования к аппаратной начинке. Однако Google рекомендует следующие характеристики, которым удовлетворяют (и чаще всего значительно их превосходят) практически все Android-гаджеты.

- ARM-процессор CPU — на момент написания книги это требование перестало быть обязательным, теперь Android работает и на архитектуре x86. Новейшие устройства на базе ARM также поддерживают двухъядерные процессоры.
- 128 Мбайт ОЗУ — это минимальное требование. В настоящее время устройства high-end оснащаются 512 Мбайт оперативной памяти, преодоление рубежа в 1 Гбайт ожидается в самое ближайшее время.
- 256 Мбайт флэш-памяти — это минимальный объем памяти для хранения системных изображений и приложений. Долгое время недостаток памяти был прокрустовым ложем для пользователей Android, поскольку сторонние приложения не могли устанавливаться на карту памяти. Все изменилось с выходом версии Froyo.
- Карта памяти Mini SD или Micro SD — большинство устройств поставляются с картой памяти на несколько гигабайт, которая может быть заменена пользователем на носитель большего объема.
- 16-битный TFT LCD сенсорный дисплей с разрешением HVGA — до выпуска версии 1.6 Android поддерживала только HVGA-экраны (480×320 пикселов). Начиная с версии 1.6 поддерживаются также более высокие и более низкие разрешения. Нынешние устройства высшей ценовой категории могут похвастаться WVGA-дисплеями (800×480, 848×480 или 852×480 пикселов), при этом некоторые бюджетные модели используют разрешение QVGA (320×280 пиксе-

лов). Сенсорные экраны почти повсеместно емкостные, поддержка мультитач отсутствует только на устаревших устройствах.

- Выделенные аппаратные клавиши используются для навигации. В большинстве телефонов присутствуют кнопки «Меню», «Поиск», «Домашний экран» и «Назад». Однако некоторые производители отклоняются от этой рекомендации и снабжают свои аппараты частью этих клавиш или вообще обходятся без них.

Естественно, в устройствах Android много и других аппаратных составляющих. Почти все гаджеты обладают GPS, акселерометром и компасом. Во многих есть датчики приближения и освещенности. Эти устройства предоставляют разработчикам игр новые возможности для взаимодействия с пользователем (мы рассмотрим их позже). Некоторые смартфоны оснащаются полной QWERTY-клавиатурой и трекболом (последний чаще всего можно обнаружить на устройствах HTC).

Камера также присутствует на большинстве устройств. Некоторые телефоны и планшеты имеют даже две камеры: одну на задней части, другую спереди (для видеочатов). Решающее значение для разработки игр имеет наличие выделенных графических процессоров (GPU). Самый первый Android-аппарат уже имел графический процессор, совместимый с OpenGL ES 1.0. Более современные устройства снабжаются GPU, по производительности сравнимыми с Xbox и PlayStation 2 и поддерживающими версию OpenGL ES 2.0. При отсутствии графического процессора платформа предлагает программный обработчик графики PixelFlinger. Многие бюджетные устройства полагаются на программный рендерер, чьей мощности обычно достаточно для экранов с низким разрешением.

Помимо графического процессора современные Android-гаджеты оснащаются и выделенным аппаратным звуковым процессором. Многие аппаратные платформы также имеют специальные чипы аппаратного декодирования различных мультимедиа-форматов (H.264).

Возможности подключения реализуются аппаратными компонентами для мобильной связи, Wi-Fi и Bluetooth. Все они в большинстве случаев интегрируются в одном чипе (SoC); подобное решение часто используется во встроенных системах.

Первое поколение, второе поколение, следующее поколение

Исходя из различия в возможностях (особенно с точки зрения производительности), Android-разработчики обычно делят все устройства на три группы — первого, второго и следующего поколений. Эта терминология довольно красноречива, особенно когда дело касается разработки игр для Android. Рассмотрим значение используемых терминов.

Каждое поколение обладает своим набором характеристик, представляющим собой комбинацию версии ОС, процессора/графического чипа и разрешения дисплея. Хотя аппаратные характеристики не меняются, это не касается версии Android, установленной на аппарате.

Начало: первое поколение

Устройства первого поколения — стартовая позиция эволюции Android. Лучше всего эту общность иллюстрирует один из наиболее выдающихся ее образцов — HTC Hero.

Это был один из первых телефонов на Android, объявленный «убийцей iPhone». Он вышел в ноябре 2009 года и работал на версии 1.5, что было стандартом для платформы в том году. Последнее официально выпущенное обновление для Hero — прошивка версии Android 2.1. Обновления до более свежей версии возможны только при получении прав root — полного доступа к системе. Hero обладает 3,2-дюймовым сенсорным емкостным HVGA-экраном, комбинацией процессора/графического чипа Qualcomm с частотой 528 МГц MSM7201A, акселерометром, компасом, а также 5-мегапиксельной камерой. Набор навигационных клавиш традиционен, кроме того, имеется трекбол.

Hero — апофеоз устройств первого поколения. Поддержка мультитач сенсорного дисплея страдает ограниченностью — возможны лишь некоторые жесты (например, увеличение и уменьшение масштаба). Нужно заметить, что официально мультитач-жесты устройством не поддерживаются (как и официальным API версии 1.5). С этой точки зрения Hero стал большим разочарованием для разработчиков игр, надеявшихся на присутствие такого же набора возможностей мультитач, что и в iPhone.

Другой характерной особенностью устройств первого поколения является разрешение экрана 480 × 320 пикселов (стандарт для версий ОС до 1.6).

Что касается процессора (в том числе графического), Hero использует довольно стандартный для своего поколения чип MSM7201A от Qualcomm, не поддерживающий на аппаратном уровне операции с плавающей точкой (еще один весьма важный момент для создания игр). MSM7201A совместим с OpenGL ES 1.0, использующим фиксированный конвейер вместо изменяемого на основе шейдеров. Графический процессор довольно быстр, но сильно уступает по производительности используемому в iPhone 3G чипу PowerVR MBX Lite (появившемуся в продаже в то же время). HTC использовал тот же чип и в других устройствах первого поколения, например в знаменитом HTC Dream (T-Mobile G1). MSM7201A быстро превращается в бюджетный процессор, когда дело доходит до аппаратной 3D-графики. Поэтому он становится вашим худшим врагом, когда дело доходит до обеспечения совместимости со всей линейкой Android.

Таким образом, устройства можно отнести к первому поколению по следующим признакам:

- процессор с частотой до 500 МГц без аппаратной поддержки операций с плавающей точкой;
- графический чип (в большинстве случаев MSM7201A), поддерживающий OpenGL ES 1.x;
- разрешение экрана 480 × 320 пикселов;
- ограниченная поддержка мультитач;
- первоначальная версия используемой ОС — Android 1.5/1.6 или ниже.

Конечно, эту классификацию нельзя назвать строгой — многие выходящие и сейчас бюджетные устройства обладают примерно тем же набором характеристик. Хотя, строго говоря, они не относятся к первому поколению, мы можем поместить их в ту же категорию, что и Hero и ему подобные. На момент написания книги смартфоны первого поколения все еще занимают приличную долю рынка, поэтому нам придется учитывать их ограничения и соответствующим образом адаптировать для них игры.

Больше мощи: второе поколение

В конце 2009 года на сцену вышли устройства второго поколения. Первыми появились Motorola Droid и Nexus One (январь 2010) и продемонстрировали невиданные ранее в мобильных телефонах вычислительные мощности. Nexus One оснащен процессором Qualcomm QSD8250 (частота 1 ГГц) из семейства чипов Snapdragon. Motorola Droid использует чип OMAP3430 (550 МГц) от Texas Instruments. Оба процессора на аппаратном уровне поддерживают векторные операции с плавающей точкой за счет использования расширений Vector Floating Point (VFP) и NEON ARM соответственно. Объем оперативной памяти Nexus One — 512 Мбайт, Motorola Droid — 256 Мбайт.

В телефонах применяется WVGA-экран — с разрешением 800 × 480 пикселов, выполненный по технологии Active-Matrix Organic Light-Emiting Diode (AMOLED) (в случае с Nexus One) и LCD с разрешением 854 × 480 пикселов (у Motorola Droid). Оба эти экрана — емкостные и поддерживают мультитач. Впрочем, несмотря на заявленную поддержку мультитач, в некоторых ситуациях он работает некорректно — наиболее часто встречающейся проблемой является неверный расчет координат касания при близком расстоянии между пальцами и краями экрана.

Nexus One первоначально поставлялся с Android версии 2.1, Motorola Droid использовал версию 2.0. В дальнейшем оба телефона получили обновления до версии Android 2.2. Что важно для разработчиков игр — наличие встроенного графического процессора. PowerVR SGX530 — мощный чип, используемый также в iPhone 3GS. Обратите внимание, что экран в iPhone 3GS вдвое меньше, чем в Motorola Droid, что дает творению Apple небольшое преимущество в производительности (ведь нужно прорисовывать меньше пикселов для каждого кадра). Чип Adreno 200, применяемый в Nexus One (продукт Qualcomm), — слегка менее производительный, чем PowerVR SGX530. В зависимости от вида обрабатываемой сцены оба этих процессора могут быть на порядок производительнее, чем используемый в большинстве устройств первого поколения MSM7201A.

Устройства второго поколения можно отличить по следующим признакам:

- процессор с частотой от 550 МГц до 1 ГГц с аппаратной поддержкой операций с плавающей точкой;
- программируемый графический ускоритель с поддержкой OpenGL ES 1.x и 2.0;
- WVGA-экран;
- поддержка мультитач;
- Android версий 2.0, 2.0.1, 2.1 или 2.2.

Стоит заметить, что некоторые телефоны первого поколения получили обновление операционной системы до версии 2.1, что оказало положительное влияние на их общую производительность, но, конечно, никак не выравняло отставание их аппаратной составляющей от устройств второго поколения. Таким образом, разграничение между первым и вторым поколениями можно проводить только с учетом всех факторов (процессор, графический чип и разрешение экрана).

В течение 2010 года появилось множество аппаратов второго поколения (таких как HTC Evo или Samsung i9200 Galaxy S). Хотя их аппаратная составляющая улучшилась по сравнению с пионерами Nexus One и Motorola Droid за счет больших размеров дисплея и немного более быстрых процессоров, эти телефоны также относятся ко второму поколению.

Будущее: следующее поколение

Производители устройств пытаются сохранить информацию о своих новинках в секрете как можно дольше, но им редко удается избежать утечек. Общая тенденция для всех будущих устройств — двухъядерные процессоры, больше оперативной памяти, более мощные графические чипы и улучшенные разрешения экрана. Один из таких аппаратов — Samsung i9200 Galaxy S2, по слухам несущий на борту AMOLED-дисплей с разрешением 1280×720, двухъядерный процессор с частотой 2 ГГц и 1 Гбайт оперативной памяти. О графической составляющей известно немногое — возможным кандидатом считают новое семейство чипов NVIDIA Tegra 2, обещающее значительное усиление графических мощностей.

Хотя мобильные телефоны в ближайшем будущем останутся в центре экосистемы Android, в ее эволюции будут играть свою роль и новые форм-факторы. Производители создают планшеты и нетбуки, использующие в качестве операционной системы Android. Портирование Android на другие архитектуры (например, x86) идет полным ходом, что увеличивает количество потенциальных платформ. Для планшетов появилась даже специальная версия — Android 3.0. Каким бы ни было будущее, Android будет к нему готова!

Игровые контроллеры

Осознавая различия методов ввода в различных гаджетах для Android, некоторые производители предлагают специальные игровые контроллеры для них. Поскольку специального API для них в операционной системе не предусмотрено, производители игр вынуждены реализовывать их поддержку с помощью SDK, предлагаемых изготовителями.

Один из таких контроллеров называется Zeemote JS1 (рис. 1.3) и представляет собой аналоговый джойстик с набором кнопок.

Контроллер подключается к устройству с помощью Bluetooth. Разработчики игр обеспечивают поддержку данного устройства с помощью отдельного API, реализованного в Zeemote SDK. Некоторые игры для Android уже поддерживают этот контроллер. Теоретически пользователи могут подключить к своим устройствам

Рис. 1.3. Контроллер Zeemote JS1

контроллер Nintendo Wii (также через Bluetooth). Были показаны несколько прототипов, реализующих возможность использования Wii, но не существует официально поддерживаемого SDK — что делает процесс интеграции слегка затруднительным.

Game Gripper — остроумное изобретение, предназначенное для использования с Motorola Droid и Milestone. Это простой резиновый аксессуар, надетый на QWERTY-клавиатуру телефона и реализующий более-менее стандартную компоновку игрового джойстика поверх обычной клавиатуры. Разработчикам игр необходимо лишь добавить поддержку клавиатурных элементов управления — им нет нужды подключать специальную библиотеку для работы с Gripper. В конце концов, это всего лишь кусок резины.

Игровые контроллеры все еще слишком экзотичны в реальности Android. Однако в некоторых успешных играх уже есть их поддержка, что было с энтузиазмом встречено игроками. Поэтому мы будем обсуждать интеграцию поддержки для таких периферийных устройств.

Игры для мобильников — особая штука

Игры для мобильников были широко распространены задолго до того, как iPhone и Android стали бороться за этот сегмент рынка. Однако появление новых устройств и идеологий заметно изменило картину. Мобильные игры перестали быть прерогативой детей — многие солидные дяди были замечены за игрой на своих мобильниках.

В газетах печатаются истории об успешных разработчиках игр, сколотивших капитал на рынке мобильных игр. Известные фирмы стремятся привлечь на свою сторону лучших разработчиков. Мы, в свою очередь, должны оперативно распознавать эти изменения и реагировать соответственно. Рассмотрим, что может нам предложить новая экосистема.

Игровая консоль в каждом кармане

Смартфоны повсюду. Это главный вывод, который необходимо извлечь из данного раздела. Из него мы можем вывести все остальные факты о мобильных играх. В условиях неуклонного уменьшения стоимости мобильных устройств, сопровождающегося столь же неуклонным увеличением вычислительной мощности, мобильники все более подходят на роль игровой консоли.

Сегодня без мобильников невозможно обойтись, рынок огромен. Многие люди, меняющие свои старые мобильные телефоны на новые смартфоны, открывают в них новые возможности в виде огромного количества приложений.

Раньше людям приходилось делать сознательный выбор — купить для игр приставку или персональный компьютер. Теперь они могут получить эту функциональность от своих телефонов бесплатно. Это не требует дополнительных вложений (конечно, если вы не считаете таковыми расходы на мобильный Интернет), и ваше игровое устройство всегда с вами. Достаньте его из кармана или сумки — и все, можно играть.

Помимо преимущества от ношения одного устройства для телефонии, Интернета и игр еще один фактор делает игры на мобильном телефоне интересными для все большей аудитории: вы можете найти интересующую вас игру на телефоне, установить ее и сразу начать играть. Нет нужды идти в магазин или скачивать что-либо на компьютер, чтобы в конце концов обнаружить, что вы потеряли USB-кабель, необходимый для переброски игры на смартфон.

Заметно возросшая мощность телефонов нынешнего поколения также влияет на расширение наших возможностей как разработчиков игр. Даже устройства среднего класса способны поддерживать игры, по своими возможностям схожие с теми, в которые вы играли на старом Xbox или PlayStation 2. При такой хорошей аппаратной поддержке мы можем экспериментировать со все более детально разработанными играми, использующими реальную физику. И это очень перспективное для развития поле.

С новыми устройствами появляются и новые методы ввода, о чем мы уже немного говорили. Некоторые игры уже используют GPS и/или компас, имеющиеся в большинстве Android-телефонов. Применение акселерометра уже стало практически обязательной функцией для большинства игр, а мультитач-экраны предлагают новые способы взаимодействия с игровым миром. По сравнению с классическими игровыми консолями (оставим за скобками Wii) это больше изменение для

разработчиков. Многое уже сделано в этом направлении, но простор для инноваций все еще велик.

Всегда на связи

Смартфоны обычно покупают в связке с тарифным планом, включающим в себя Интернет. Уже давно они используются не только для телефонных разговоров, но и для серфинга по любимым сайтам. Пользователь со смартфоном с большой вероятностью будет подключен к Сети постоянно (за исключением случаев ошибок в аппаратной составляющей).

Постоянное подключение открывает совершенно новые горизонты для мобильных игр. Люди могут сыграть в шахматы с живыми соперниками по всему миру, вместе исследовать виртуальные миры или пытаться укокошить лучшего друга в виртуальной схватке один на один. И все это может происходить на автобусной остановке, в поезде или любимом уголке местного парка.

Помимо мультиплеера важную роль в мобильном игровом мире стали играть социальные сети. В играх предлагается возможность твитнуть[1] ваш лучший результат или сообщить другу о вашем достижении. Хотя подобные сервисы существуют и в мире традиционных игровых приставок (например, в Xbox и PlayStation 2), рыночная доля таких площадок, как Facebook и Twitter, неизмеримо выше. Кроме того, пользователь мобильного телефона может публиковать информацию в несколько социальных сетей сразу.

Простые и крутые

Стремительное увеличение количества пользователей смартфонов помимо прочего означает, что люди, никогда не державшие в руках джойстика от NES, открывают для себя игровой мир. Их представление о хорошей игре часто очень отличается от мыслей по тому же поводу прожженного геймера. Применяя мобильный телефон, пользователи больше склоняются к казуальным играм, на которые они могут потратить несколько минут в автобусе или в автомобильной пробке. Такие забавы очень похожи на те маленькие флэш-игры на персональных компьютерах, заставляющие лихорадочно нажимать комбинацию Alt+Tab каждый раз, когда босс проходит у вас за спиной.

Спросите себя: как много времени вы собираетесь тратить, играя на мобильном телефоне? Вы можете себе представить, как это — сыграть «по-быстрому» в «Цивилизацию» на подобных устройствах? Есть, конечно, люди, готовые продать душу за возможность сыграть в любимых Dungeons & Dragons на своем мобильнике. Однако список самых продаваемых игр для iPhone и Android свидетельствует, что таких меньшинство. Самые популярные игры обычно весьма просты по сути,

[1] Твитнуть — опубликовать сообщение на сайте Twitter (twitter.com). — *Примеч. ред.*

однако обладают при этом характерной особенностью: среднее время сеанса игры в них составляет минуты, но вы снова и снова к ним возвращаетесь. Достигается это разными путями: игра может предложить продуманную онлайн-систему учета результатов, позволяющую вам похвастаться своим успехом. Однако на самом деле это может быть и весьма крутая игра. Предложите пользователям простой путь сохранения своих достижений, и вы сможете продать им свою сложную игру как простую!

Крупный рынок, мелкие разработчики

Низкий порог вхождения — главный привлекательный момент для многих независимых разработчиков и просто интересующихся. В случае с Android этот барьер особенно низок — скачайте SDK и программируйте. Вам даже не нужно реальное устройство, можно использовать эмулятор (хотя я бы все же рекомендовал иметь хотя бы один аппарат). Открытая природа Android также способствует большой активности в Веб. Информацию обо всех аспектах разработки можно найти бесплатно онлайн. Нет необходимости подписывать соглашение о неразглашении или ждать подтверждения на доступ к экосистеме.

Когда писалась эта книга, самые успешные игры были разработаны компаниями, состоящими из одного или нескольких человек. Крупные издатели пока не захватили этот рынок (по крайней мере успешно). Gameloft — весьма красноречивый пример. Добившись успеха на рынке игр для iPhone, компания не смогла повторить его на поле Android и в итоге решила продавать игры на собственном сайте вместо Android Market. В Gameloft могли быть недовольны отсутствием схемы управления цифровыми правами (которая теперь доступна и для Android) — в итоге количество потенциальных потребителей их игр заметно уменьшилось.

Среда также весьма открыта для экспериментов и инноваций, предназначенных для рыщущих по магазину в поисках маленьких шедевров пользователей (включая новые идеи и игровую механику). Эксперименты на классических игровых платформах (PC или консоли) часто приводят к провалам. Однако Android Market позволяет вам обратиться к большой аудитории, открытой для нового, и достичь результата с намного меньшими усилиями.

Конечно, это не значит, что вам не нужно будет продвигать вашу игру. Один из способов — распространять информацию о вашей новинке в различных блогах и на специализированных сайтах. Многие пользователи Android — энтузиасты и регулярно изучают эти источники в поисках нового и лучшего.

Другой путь — воспользоваться Android Market. Разместите на нем ваше приложение, и оно немедленно станет доступно всем, кто на него заходит. Многие разработчики сообщают о гигантском росте скачиваний после размещения их программы в магазине. Хотя точного рецепта, как стать популярным, увы, нет. Лучший способ — придумать красивую идею и тщательно ее реализовать. При этом не имеет значения, крупный ли вы издатель или энтузиаст-одиночка.

Подводя итог

Android — прекрасная миниатюрная штуковина. Вы увидели, из чего она состоит, и немного узнали о ее экосистеме. Android предлагает нам очень интересную систему для разработки с точки зрения программной и аппаратной составляющей при очень низком пороге вхождения (все, что вам нужно, — бесплатный SDK). Сами устройства весьма производительны, что позволяет нам создавать полноценные игровые миры для наших пользователей. Применение различных датчиков (например, акселерометра) дает возможность реализовать новые идеи взаимодействия с игроком. А после создания игры у нас будет прекрасная возможность распространить ее по всему миру в течение нескольких минут. Звучит впечатляюще? Тогда пора заняться программным кодом!

2 Первые шаги с Android SDK

Android SDK предлагает набор инструментов для почти мгновенного создания приложений. Эта глава познакомит вас с процессом создания простой программы для платформы Android с использованием этих инструментов. Данный процесс состоит из следующих шагов.

1. Настройка среды разработки.
2. Создание нового проекта в Eclipse и написание кода.
3. Запуск приложения на эмуляторе или реальном устройстве.
4. Отладка и анализ приложения.

Начнем по порядку — с настройки среды разработки.

ПРИМЕЧАНИЕ

Поскольку Web — штука переменчивая, я не даю вам конкретных URL-адресов. Уверен, что ваш любимый поисковик найдет все, что вам нужно.

Настройка среды разработки

Android SDK — весьма гибкий продукт. Он способен интегрироваться с различными средами разработки. Ортодоксы могут пользоваться только командной строкой для создания приложения, но мы изберем более простой, наглядный и комфортный способ — будем применять IDE (интегрированную среду разработки).

Вот вам список ПО, которое необходимо скачать и установить в заданном здесь порядке:

- Java Development Kit (JDK), версии 5 или 6; я советую использовать более свежую;
- Android Software Development Kit (Android SDK);
- Eclipse для Java, версия 3.4 или 3.5;
- плагин Android Development Tools (ADT) для Eclipse.

Выполним всю последовательность действий, требуемых для правильной настройки окружения.

Настройка JDK

Скачайте JDK одной из указанных выше версий для вашей операционной системы. В большинстве случаев он предлагается в виде инсталлятора или пакета, так что трудностей быть не должно. После установки JDK разумным будет установить переменную окружения `JDK_HOME`, указывающую на каталог установки JDK. Кроме того, вы должны добавить каталог `$JDK_HOME/bin` (`%JDK_HOME%\bin` в Windows) в вашу переменную окружения `PATH`.

Настройка Android SDK

Android SDK также доступен для трех основных настольных операционных систем. Выберите вашу платформу и скачайте SDK в виде архива ZIP или TAR GZIP. Разархивируйте его в подходящую папку (например, `c:\android-sdk` в Windows или `/opt/android-sdk` на Linux). SDK поставляется с несколькими утилитами командной строки, размещенными в каталоге `tools`. Создайте переменную окружения `ANDROID_HOME`, указывающую на корневой каталог установленного SDK, и добавьте каталог `$ANDROID_HOME/tools` (`%ANDROID_HOME%\tools` в Windows) в вашу переменную `PATH`. Теперь вы сможете легко использовать утилиты из командной строки при необходимости.

После выполнения предыдущих шагов вы получите минимальную версию среды разработки, состоящую из командных утилит создания, компиляции и развертывания проектов Android, а также SDK and AVD manager (утилиты для инсталляции компонентов SDK и создания виртуальных устройств для эмулятора). Самих по себе этих инструментов недостаточно для начала разработки, поэтому необходимо установить дополнительные компоненты. Для этого и нужен SDK and AVD manager, управляющий пакетами (это очень похоже на управление пакетами в Linux). Данный инструмент позволяет вам устанавливать следующие виды компонентов.

- *Платформа Android*. Для каждого официального релиза ОС Android выпускается компонент SDK, включающий библиотеки выполнения, образ системы для эмулятора и другие специфичные для конкретной версии инструменты.
- *Дополнения SDK*. Обычно это внешние библиотеки и утилиты, не связанные с конкретной версией платформы. Яркий пример — API для работы с картами Google в вашем приложении.
- *Драйвер USB для Windows*. Необходим для запуска и отладки приложения на реальном устройстве при разработке в Windows. В операционных системах Mac OS X и Linux специальный драйвер вам не понадобится.
- *Примеры*. Для каждой платформы предлагается набор специфичных для нее примеров. Это хороший источник получения знаний.
- *Документация*. Локальная копия документации по последней версии API фреймворка Android.

Мы, конечно, хотим установить все эти компоненты, чтобы иметь в своем распоряжении полный функциональный набор. Поэтому сначала мы запустим SDK and AVD manager. В случае с Windows для этого необходимо запустить файл SDK `manager.exe`, расположенный в корневом каталоге SDK. При работе с Linux и Mac OS X вы просто запускаете скрипт `android`, расположенный в каталоге `tools` в SDK.

При первом запуске SDK and AVD manager подключится к серверу-репозиторию и получит список доступных пакетов. Он продемонстрирует его вам в виде диалога, показанного на рис. 2.1, чтобы вы могли выбрать отдельные пункты из списка. В нашем случае мы просто усановим переключатель в положение Accept All (Принять все) и нажмем кнопку Install (Установить), после чего заварим чаю или кофе и будем ждать — для установки всех пакетов понадобится некоторое время.

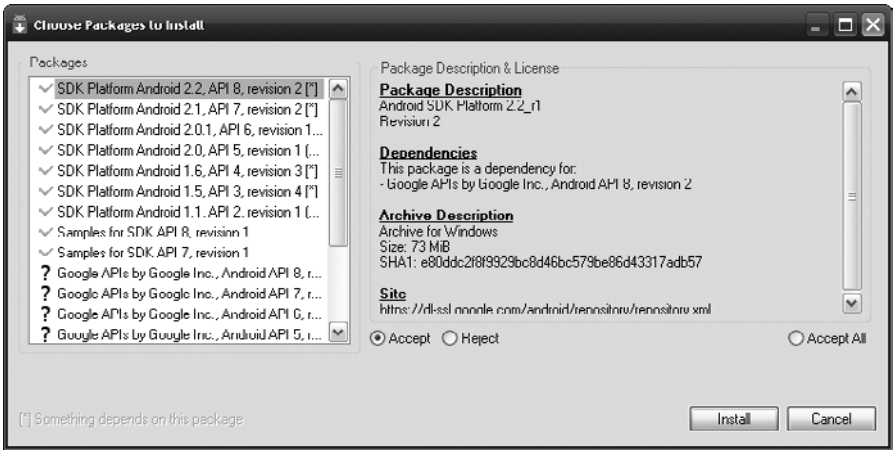

Рис. 2.1. Первое знакомство с SDK and AVD manager

Вы можете использовать SDK and AVD manager в любое время, чтобы обновить компоненты или установить новые. Утилита применяется также для создания новых AVD, что понадобится нам позже при запуске и отладке программ на эмуляторе.

После окончания процесса установки можно переходить к следующему шагу настройки среды разработки.

Установка Eclipse

Eclipse поставляется в нескольких различных формах. Для разработчиков Android я рекомендую использовать Eclipse for Java Developers версии 3.6. Как и Android SDK, Eclipse упаковывается в один архив формата ZIP или TAR GZIP. Просто разархивируйте его в каталог по вашему выбору. После распаковки вы можете создавать на вашем Рабочем столе ярлык к исполняемому файлу `eclipse` для запуска среды, расположенному в установочном каталоге.

При первом запуске среды вам будет предложено определить каталог для рабочего пространства. На рис. 2.2 показан соответствующий диалог.

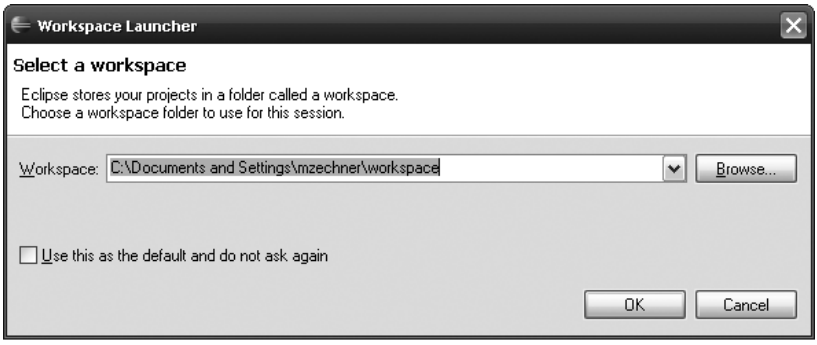

Рис. 2.2. Выбор рабочего пространства

Рабочим пространством в Eclipse называется каталог, содержащий проекты. Вы можете использовать одно пространство для всех проектов или несколько пространств по вашему усмотрению. Проекты-примеры, сопутствующие этой книге, организованы в единое рабочее пространство, которое вы можете задать в данном диалоговом окне. Сейчас мы просто создадим пустое пространство в каком-нибудь каталоге.

После этого Eclipse приветствует нас экраном «Добро пожаловать», который спокойно можно проигнорировать и закрыть. В результате вы окажетесь в Java-перспективе по умолчанию. В следующем разделе нам предстоит познакомиться с Eclipse поближе. Пока же нам достаточно того, что он запускается.

Установка плагина ADT для Eclipse

Последний шаг нашей головоломки — установка плагина ADT для Eclipse. Данная среда основывается на концепции плагинов, расширяющих ее функциональность за счет продуктов сторонних разработчиков. Плагин ADT соединяет инструменты Android SDK с мощью Eclipse. Благодаря ему можно полностью забыть об использовании утилит командной строки Android SDK — плагин ADT прозрачно интегрирует их в рабочий поток Eclipse.

Установка плагинов для Eclipse может осуществляться как вручную (копированием отдельных компонентов ZIP-файла в папку плагинов Eclipse), так и с помощью интегрированного в среду менеджера плагинов. Мы, конечно, изберем второй путь.

1. Для установки нового плагина выполните команду меню Help ▸ Install New Software (Справка ▸ Установить новое программное обеспечение), в результате чего откроется диалог установки. В нем можно выбрать источник установки плагина. Прежде всего необходимо добавить репозиторий, откуда будет скачиваться

плагин ADT. Нажмите кнопку Add (Добавить), чтобы увидеть диалог, представленный на рис. 2.3.

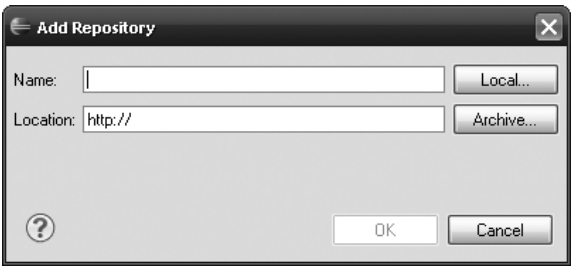

Рис. 2.3. Добавление репозитория

2. В первом текстовом поле вы можете ввести произвольное название репозитория (вполне сойдет что-то вроде `ADT repository`). Во второе поле необходимо ввести URL-адрес репозитория. В случае с плагином ADT поле должно быть заполнено так: https://dl-ssl.google.com/android/eclipse/. Замечу, что данный адрес может отличаться для более новых версий, поэтому перед обновлением стоит проверить сайт плагина ADT.
3. После подтверждения данных вы вновь вернетесь в диалог установки, который теперь должен содержать список доступных плагинов репозитория. Установите флажок Developer Tools (Инструменты разработчика) и нажмите кнопку Next (Далее).
4. Теперь Eclipse рассчитает все необходимые зависимости, после чего продемонстрирует диалог, содержащий все плагины и зависимости, планируемые к установке. Подтвердите установку нажатием кнопки Next (Далее).
5. Появится еще один диалог, предлагающий вам принять лицензионные соглашения для каждого устанавливаемого дополнения. Вы, конечно, должны принять все, после чего инициировать инсталляцию нажатием кнопки Finish (Завершить).

ПРИМЕЧАНИЕ

Во время установки вам будет предложено подтвердить инсталляцию неподписанного программного обеспечения. Не беспокойтесь, это просто означает, что плагины не имеют удостоверенной подписи. Согласитесь на установку для продолжения процесса.

6. В конце Eclipse спросит вас о необходимости перезапуска среды после установки плагинов. Вы можете выбрать полный перезапуск или применение изменений без перезапуска. Нажмите Restart Now (Перезапустить сейчас) для перезагрузки.

После преодоления всего этого диалогового безумия вы увидите то же самое окно Eclipse, что и раньше. То же, да не совсем — на панели инструментов появится несколько связанных с Android кнопок, позволяющих вам запустить SDK and

AVD manager непосредственно из Eclipse, а также создавать новые проекты Android. Эти кнопки можно увидеть на рис. 2.4.

Рис. 2.4. Кнопки панели инструментов ADT

Первая слева кнопка предназначена для запуска AVD and SDK Manager. Следующая — ярлык для создания новых Android-проектов. Остальные две кнопки — создание проекта юнит-теста и файла манифеста (эти функции мы не будем рассматривать в данной книге).

В качестве последнего шага в установке плагина ADT необходимо указать ему расположение Android SDK.

1. Выполните команду меню Window ▸ Preferences (Окно ▸ Настройки) и в появившемся диалоге выберите Android из дерева.
2. Нажмите кнопку Browse (Обзор), расположенную справа, для выбора корневого каталога вашей инсталляции Android SDK.
3. Щелкните на кнопке OK для закрытия диалога. Все, теперь вы можете создавать свое первое Android-приложение.

Краткий обзор среды Eclipse

Eclipse — открытая IDE, которую вы можете использовать для разработки приложения на различных языках. Обычно, говоря Eclipse, подразумевают Java. Однако благодаря модульной архитектуре для Eclipse разработано множество расширений, позволяющих писать программы на C/C++, Scala или Python. Возможности почти безграничны; существует, например, плагин для создания проектов LaTeX (которые очень слабо напоминают программы, к которым вы привыкли).

Экземпляр Eclipse работает с рабочей областью, содержащей один или несколько проектов. Мы определили эту рабочую область ранее, и теперь в ней будут храниться все созданные нами проекты (а кроме того, конфигурационные файлы, определяющие внешний вид среды для данной рабочей области).

Пользовательский интерфейс (UI) связан с двумя концепциями:

- *представление*, являющееся отдельным компонентом интерфейса (например, редактор кода, консоль или инспектор объектов);
- *перспектива* — набор отдельных представлений, которые необходимы для различных задач разработки (например, редактирования кода, отладки, анализа, синхронизации, контроля версий и т. п.).

Eclipse для Java имеет набор предустановленных перспектив. Самые интересные из них — Java и Debug (Отладка). Перспектива Java показана на рис. 2.5. Она содержит представление Package Explorer (Диспетчер пакетов) в левой части, представление для редактирования кода в середине (оно пустое, потому что мы еще

не открыли ни одного исходника), список задач справа, представление Outline (Контур) и набор закладок, в котором скомпонованы представления Problems (Проблемы), Javadoc и Declaration (Объявление).

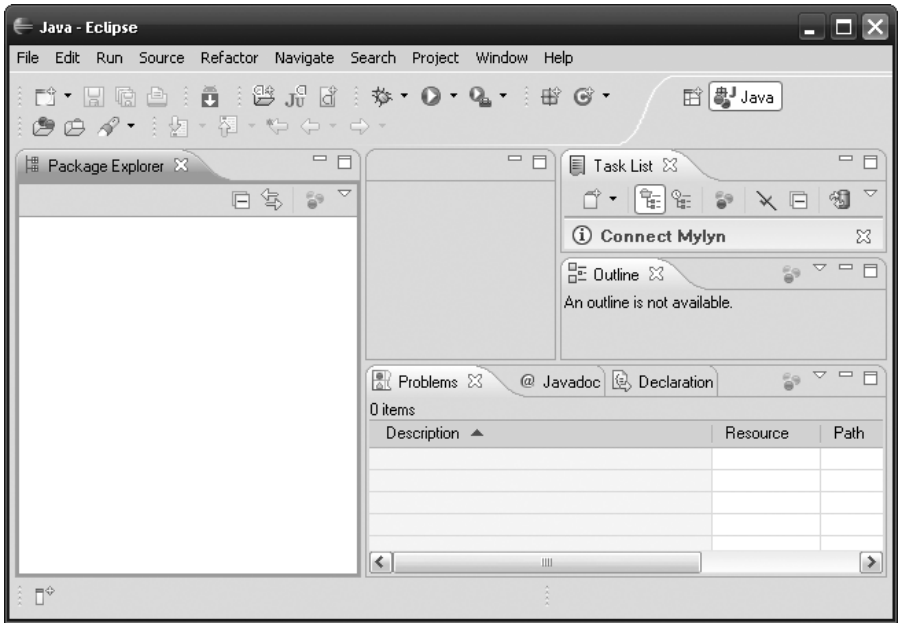

Рис. 2.5. Eclipse в действии — перспектива Java

Вы можете перемещать представления внутри перспективы по своему усмотрению простым перетаскиванием, а также изменять их размеры. Кроме того, набор представлений не закреплен жестко — вы можете добавлять их или удалять из перспективы. Чтобы добавить представление, перейдите в меню Window ▸ Show View (Окно ▸ Показать вид) и либо отметьте что-нибудь из списка, либо выберите Other (Другие), чтобы получить список всех доступных представлений. Для переключения на другую перспективу вы можете перейти в меню Window ▸ Open Perspective (Окно ▸ Показать перспективу) и выбрать ту, которая вам нужна. Более быстрый способ переключения между открытыми перспективами доступен вам в левом верхнем углу Eclipse. Здесь вы можете узнать, какие перспективы уже открыты и какая из них активна. При открытии дополнительных перспектив они будут добавлены в данный список.

Панели инструментов, показанные на рис. 2.5, — на самом деле тоже представления. В зависимости от выбранной перспективы панели инструментов также могут меняться. Помните, как после установки плагина ADT на панели появились новые кнопки? Такое происходит со всеми плагинами: они обычно добавляют новые представления и перспективы. В случае с ADT мы после установки получаем доступ к перспективе DDMS (предназначенной для отладки и анализа приложений Android и дополняющей при работе стандартные перспективы Java и Debug

(Отладка)). Кроме того, плагин ADT добавляет в среду несколько представлений, включая LogCat (вывод журнала операций с любого подключенного устройства или эмулятора).

После настройки под свои нужды перспектив и представлений Eclipse уже не выглядит таким пугающим. Далее мы изучим некоторые перспективы и представления, которые понадобятся нам при написании игр для Android. Я, конечно, не могу осветить все подробности разработки в Eclipse из-за его сложности. Вместо этого дам совет изучить подробную справку по системе в случае возникновения такой необходимости.

Hello World в стиле Android

Настроив среду разработки, мы можем, наконец, создать наш первый Android-проект в Eclipse. Вместе с плагином ADT вы получили набор помощников, делающих процесс создания новых проектов для Android действительно легким.

Создание проекта

Существуют два способа создания нового Android-проекта. Щелкните правой кнопкой мыши в представлении Package Explorer (Диспетчер пакетов) (см. рис. 2.5) и выберите New Project (Новый проект) из появившегося контекстного меню. В открывшемся диалоге выберите Android Project (Проект Android) в категории Android. Как видите, тут есть множество других параметров создания проекта. Это стандартный способ создать новый проект любого типа в Eclipse. После подтверждения откроется помощник создания проектов Android. Второй способ намного проще: просто нажмите кнопку создания нового Android-проекта (см. рис. 2.4).

В диалоговом окне создания проекта Android необходимо установить несколько значений.

1. Дать проекту название. По соглашению о наименовании принято использовать только маленькие буквы. В данном примере проект будет называться `hello world`.
2. Определить цель сборки. Пока выберем цель Android 1.5 — первая программа будет очень проста, и нам не потребуются продвинутые возможности вроде мультитач.

ПРИМЕЧАНИЕ

В главе 1 вы видели, что каждый новый релиз Android добавляет новые классы во фреймворк API. Параметр, задающий цель сборки, определяет, какую версию этого API вы хотите использовать в вашем приложении. Например, выбрав цель Android 2.3, вы получаете доступ к самому полному и актуальному набору функций. Но платой за это является риск разделения: если кто-то запустит ваше приложение на аппарате с более старой версией ОС (например, Android 1.5), то при попытке обращения вашей программы к возможностям API, доступным только в версии 2.3, произойдет сбой. В таких случаях вам необходимо будет программно определять используемую версию во время выполнения и использовать расширенный функционал только при работе с соответствующей версией операционной системы. Это звучит ужасно, но, как вы увидите в главе 5, при правильной архитектуре вы вполне можете реализовать подобный выбор.

3. Определить название вашего приложения (например, классическое `hello world`), название пакета Java, в котором будут помещаться ваши исходные файлы (например, `com.helloworld`), и название активности. Понятие «активность» в Android — синоним окна или диалога в настольных операционных системах. Назовем его `HelloWorldActivity`.
4. Задать минимальную версию Android, необходимую для запуска вашего приложения. Для этого используется поле Min SDK Version (Минимальная версия SDK). Это необязательный параметр, однако его установка считается хорошим тоном. Версии SDK нумеруются начиная с 1 (1.0) и увеличиваются по мере выхода новых релизов. С версии ОС 1.5 актуальным является третий релиз, поэтому введем в поле значение 3. Помните, что ранее вам необходимо было выбрать цель сборки, которая может оказаться новее, чем минимальная версия SDK. Это позволяет вам работать с более высоким уровнем API, сохраняя, однако, совместимость с более старыми версиями ОС (конечно, необходимо убедиться, что вы вызываете только те методы, которые поддерживаются API для данной версии).

После этого нажмите Finish (Готово) для создания вашего первого проекта Android.

ПРИМЕЧАНИЕ

Установка минимального уровня версии SDK имеет свои последствия. Программа может запускаться только на устройствах с версией Android, равной или большей минимальному уровню SDK, который вы определили. Когда пользователь посещает Android Market с помощью стандартного системного приложения, то он видит только приложения, удовлетворяющие требованиям минимального SDK.

Исследование проекта

В представлении Package Explorer (Диспетчер пакетов) вы должны увидеть проект, названный `hello world`. Если вы развернете его и его подразделы, то увидите нечто, похожее на рис. 2.6. Это общая структура большинства Android-проектов. Рассмотрим ее подробнее.

- `AndroidManifest.xml` — описывает приложение. Он определяет, какие активности и службы в нем содержатся, на какой минимальной и целевой версии Android его предполагается запускать и какие разрешения ему необходимы для работы (например, доступ к карте памяти или к сети).
- `default.properties` — хранит различные настройки сборки. Мы не будем трогать этот файл — ADT сам меняет его содержимое при необходимости.
- `src/` — содержит все исходники на Java. Обратите внимание — пакет имеет имя, определенное нами на этапе создания проекта.
- `gen/` — содержит исходники на Java, сгенерированные сборочной системой Android. Они создаются автоматически, и мы не будем иметь с ними дело напрямую.

Рис. 2.6. Структура проекта Hello World

- `assets/` — место хранения файлов, необходимых приложению (например, файлы конфигурации и звуковые дорожки). Эти файлы при сборке включаются в пакет.
- `res/` — место хранения ресурсов для приложения (значков, строковых значений для интернационализации, XML-файлов компоновки). Как и ресурсы (`assets`), они включаются в пакет при сборке приложения.
- `Android 1.5` — сообщает нам, что мы собираем приложение для версии ОС 1.5. Это на самом деле зависимость в форме стандартных JAR-файлов, хранящих в себе классы API для Android 1.5.

Представление Package Explorer (Диспетчер пакетов) скрывает от нас еще один каталог — `bin/`, содержащий компилированный код, который можно разворачивать на устройстве или эмуляторе. Как и в случае с папкой `gen/`, обычно можно не задумываться, что в ней происходит.

Можно с легкостью добавлять в представление Package Explorer (Диспетчер пакетов) новые исходники, папки и другие ресурсы. Для этого нужно щелкнуть правой кнопкой мыши на необходимом каталоге, выбрать в контекстном меню пункт New (Новый) и указать тип ресурса, который мы хотим добавить. Однако в данном случае у нас уже есть все необходимое. Следующая наша задача — немного изменить исходный код.

Написание кода приложения

К этому моменту мы не написали еще ни единой строчки кода — пора изменить это положение вещей. Помощник создания проекта Android создал для нас класс шаблона активности, называющийся `HelloWorldActivity`. Именно эта активность будет показываться на экране телефона при запуске приложения на эмуляторе или

реальном устройстве. Откройте исходный код класса двойным нажатием файла в представлении Package Explorer (Диспетчер пакетов). Мы заменим код этого шаблона строками из листинга 2.1.

Листинг 2.1. HelloWorldActivity.java

```
package com.helloworld;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class HelloWorldActivity extends Activity

,                                implements View.OnClickListener {
    Button button;
    int touchCount;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        button = new Button(this);
        button.setText( "Touch me!" );
        button.setOnClickListener(this);
        setContentView(button);
    }

    public void onClick(View v) {
        touchCount++;
        button.setText("Touched me " + touchCount + " time(s)");
    }
}
```

Проанализируем листинг 2.1, чтобы понять, что он делает. Оставим скучные подробности для следующих глав. Все, что нам сейчас необходимо, — общее понимание происходящего.

Файл программного кода начинается со стандартного объявления Java-пакетов и нескольких деклараций импорта. Большая часть классов, составляющих фреймворк Android, содержится в пакете `android`:

```
package com.helloworld;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
```

Далее определяем класс `HelloWorldActivity` и наследуем его от базового класса `Activity`, предлагаемого API фреймворка Android. `Activity` — это, в общем-то, ана-

лог окна в классических интерфейсах настольных систем, имеющий одно ограничение — оно всегда должно занимать весь экран устройства (за исключением строки оповещения в верхней части интерфейса Android).

Кроме того, мы реализуем в этом классе интерфейс OnClickListener. Если у вас был опыт работы с другими инструментами разработки интерфейсов, то, возможно, вы можете предугадать следующий шаг:

```
public class HelloWorldActivity extends Activity
                                 implements View.OnClickListener {
```

В нашей активности будет два элемента: Button и число, хранящее количество нажатий этой кнопки:

```
    Button button;
    int touchCount;
```

Каждый класс типа Activity должен реализовывать абстрактный метод Activity.onCreate(), вызываемый системой однократно при первом появлении активности. Это своего рода замена привычному конструктору, используемому обычно для создания экземпляра класса. Вызов метода onCreate() базового класса обязательно должен быть первой строкой в теле метода:

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

Далее создаем объект типа Button и устанавливаем для него первоначальный текст. Button — один из многих виджетов, предлагаемых API фреймворка Android. Виджет — понятие, синонимичное представлению в Android. Обратите внимание — наша кнопка является членом класса HelloWorldActivity. Чуть позже нам понадобится ссылка на него:

```
        button = new Button(this);
        button.setText( "Touch me!" );
```

Следующая строка метода onCreate() устанавливает OnClickListener для объекта Button. OnClickListener — интерфейс обратного вызова с единственным методом, OnClickListener.onClick(), вызываемым каждый раз при нажатии кнопки. Мы хотим получать уведомления об этих нажатиях, поэтому реализуем в HelloWorldActivity данный интерфейс и зарегистрируем его в качестве OnClickListener для нашего объекта Button:

```
        button.setOnClickListener(this);
```

Последняя строка нашего метода onCreate() назначает объект Button так называемым «представлением содержимого» нашей активности. Представления могут быть вложенными, и представление содержимого является корневым звеном этой иерархии. В данном случае мы просто устанавливаем объект Button как представление, отображаемое нашей активностью. Для простоты мы не будем

вдаваться в детали того, как активность будет компоноваться с данным представлением содержимого.

```
        setContentView(button);
    }
```

Следующий шаг — реализация метода `OnClickListener.onClick()`, чего требует интерфейс, подключенный к нашему классу `Activity`. Этот метод вызывается каждый раз, когда нажимается кнопка. Внутри него мы увеличиваем на единицу значение счетчика `touchCount` и присваиваем тексту кнопки это новое значение.

```
    public void onClick(View v) {
        touchCount++;
        button.setText("Touched me" + touchCount + "times");
    }
```

Подводя итог наших усилий по разработке приложения `hello world`, можно сказать, что мы создали активность, содержащую элемент типа `Button`. При каждом его нажатии мы реагируем, изменяя текст кнопки. Возможно, это не самое впечатляющее приложение в мире, но для демонстрации оно вполне подходит. Заметьте — нам пока ничего не пришлось компилировать вручную. Плагин ADT в сочетании с Eclipse будет пересобирать проект каждый раз, когда мы добавляем, изменяем или удаляем файлы кода или ресурсы. Результат этой сборки — файл с расширением APK, который готов к разворачиванию на эмуляторе или реальном устройстве. Файл APK располагается в папке `bin/` проекта. Вы будете использовать данное приложение в следующих разделах для понимания процесса запуска и отладки Android-приложений на эмуляторах или телефонах.

Запуск и отладка приложений Android

После написания первой версии кода нашей программы мы, конечно, захотим запустить и протестировать его для обнаружения потенциальных проблем (или чтобы просто похвастаться). Сделать это можно двумя способами:

- запустить приложение на реальном устройстве, подключенном к компьютеру с помощью USB-кабеля;
- вызвать эмулятор, включенный в SDK, и протестировать приложение на нем.

В обоих случаях нам нужно произвести некоторые действия, чтобы увидеть нашу программу в действии.

Подключение устройства

Перед подключением устройства для тестирования необходимо убедиться в том, что оно распознается операционной системой. В Windows для этого нужно установить соответствующий драйвер, являющийся частью инсталляции SDK, которую мы установили ранее. Просто подсоедините устройство и следуйте инструкциям по установке стандартного драйвера для Windows, указав папку `driver/` в вашем установочном каталоге SDK. Драйверы для некоторых устройств придется скачать

с сайта их производителей. На Linux и Mac OS X обычно нет нужды устанавливать драйверы отдельно — они поставляются с операционной системой. В зависимости от версии Linux может понадобиться совершить несколько дополнительных действий (обычно в части создания нового файла правил для udev). Для разных устройств набор действий может различаться — веб-поиск вам в помощь.

Создание виртуального устройства Android

SDK поставляется с эмулятором, запускающим так называемые виртуальные устройства Android (AVD). Это виртуальное устройство состоит из образа определенной версии операционной системы Android, оболочки и набора атрибутов, включающих разрешение дисплея, размер карты памяти и т. д. Для создания нового AVD необходимо запустить SDK and AVD manager. Вы можете сделать это как описанным в инструкции по инсталляции SDK способом, так и напрямую в Eclipse, нажав кнопку SDK manager (Диспетчер комплекта разработки) на панели инструментов.

1. Выберите Virtual Devices (Виртуальные устройства) из списка в левой части. В результате вы увидите список доступных виртуальных устройств. Если вы ранее не пользовались SDK manager, данный список будет пуст; изменим это положение вещей.
2. Для создания нового AVD нажмите кнопку New (Новый) в правой части. Появится диалог (рис. 2.7).

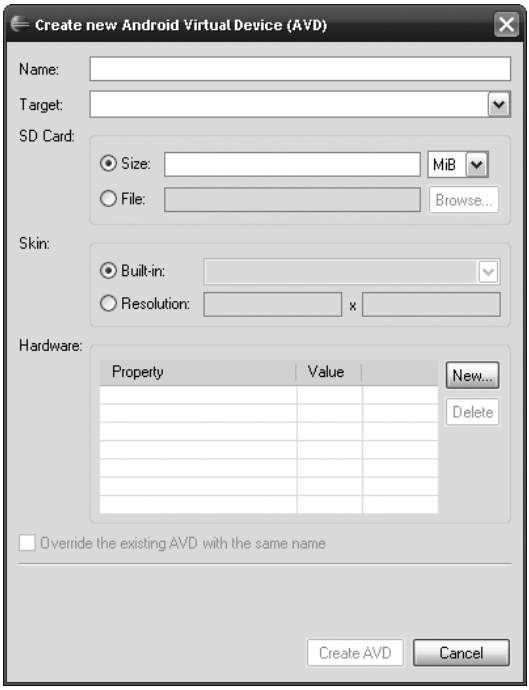

Рис. 2.7. Диалог создания AVD в SDK manager

3. Каждое виртуальное устройство обладает именем (поле Name (Имя)), по которому вы впоследствии будете на него ссылаться. В Target (Цель) определяется версия Android, которую должен использовать AVD. Кроме того, вы можете определить объем карты памяти для AVD, а также разрешение экрана. Для нашего простого проекта `hello world` можно выбрать в качестве цели Android 1.5, оставив остальные параметры без изменений. В условиях реального тестирования обычно приходится создавать несколько виртуальных устройств, чтобы проверить работу приложения для различных версий ОС и размеров дисплея.

ПРИМЕЧАНИЕ

Если у вас нет реальных аппаратов на Android разных версий и с различными экранами, для дополнительного тестирования совместимости приложения удобнее использовать эмулятор.

Запуск приложения

Теперь после настройки устройств и AVD вы наконец можете запустить ваше приложение. В Eclipse это делается просто — щелчком правой кнопкой мыши на проекте `hello world` в представлении Package Explorer (Диспетчер пакетов) и выбором пункта Run As Android Application (Выполнить как приложение Android) (или же нажатием кнопки Run (Выполнить) на панели инструментов). В результате среда выполнит в фоновом режиме следующие действия.

1. Скомпилирует проект в файл APK (если с момента прошлой компиляции произошли изменения в файлах).
2. Создаст новую конфигурацию запуска для проекта Android если она еще не существует (скоро мы поговорим о конфигурации запуска).
3. Установит и запустит приложение с помощью запуска нового или использования уже запущенного эмулятора соответствующей версии Android либо его развертывания и запуска на подключенном устройстве (на котором также установлена версия ОС не ниже определенной параметром Min SDK Version (Минимальная версия SDK) при создании проекта).

Если вы только что создали AVD для Android 1.5 (как было описано выше), плагин ADT для Eclipse запустит новый экземпляр эмулятора, развернет в нем APK проекта `hello world` и запустит приложение. На выходе вы увидите нечто, похожее на рис. 2.8.

Эмулятор работает почти так же, как реальное устройство, и вы можете взаимодействовать с ним посредством мыши, как будто используете палец. Однако есть и несколько отличий от работы с реальным аппаратом.

- Эмулятор не поддерживает мультитач. Двигайте указателем мыши и представляйте, что это палец.
- В эмуляторе отсутствуют некоторые приложения (например, Android Market).
- Для изменения ориентации экрана бесполезно трясти монитор. Вместо этого используйте клавишу 7 на дополнительном цифровом блоке клавиатуры для поворота дисплея. Чтобы не набрать вместо этого цифру 7, необходимо сначала нажать Num Lock.

Рис. 2.8. Потрясающее приложение hello world в действии!

- Эмулятор очень, очень медленно работает. Не судите о производительности вашего приложения по скорости его работы на эмуляторе.
- На момент написания книги эмулятор поддерживает только OpenGL ES 1.0 с несколькими расширениями (мы поговорим об OpenGL ES подробнее в главе 7). Для наших целей этого достаточно (за исключением того, что реализация графической библиотеки на эмуляторе страдает погрешностями и иногда вы можете получать не те результаты, что на реальном устройстве). Пока просто запомните, что не стоит тестировать приложения, использующие OpenGL ES, на эмуляторе.

Попробуйте различные действия с эмулятором, чтобы привыкнуть к нему.

ПРИМЕЧАНИЕ

Запуск нового экземпляра занимает значительное время (до нескольких минут в зависимости от характеристик рабочей станции). Чтобы сэкономить время, оставляйте эмулятор запущенным весь сеанс разработки, не перезапуская его каждый раз.

Иногда при запуске приложения Android автоматический выбор эмулятора/устройства, выполняемый плагином ADT, становится помехой. Например, мы подключили несколько аппаратов или эмуляторов и хотим протестировать проект на каком-то одном из них. Чтобы этого добиться, можно отключить автоматический выбор эмулятора/устройства в конфигурации запуска проекта Android. Кстати, что такое конфигурация запуска?

Конфигурация запуска предлагает способ сообщить среде Eclipse, каким именно образом она должна запускать ваше приложение, получив соответствующую команду. Обычно это выражается в возможности определить аргументы командной строки, передаваемые программе, аргументы виртуальной машины (в случае с настольными приложениями на Java SE) и т. д. Eclipse и сторонние плагины предлагают разные конфигурации запуска для определенных типов проекта. ADT не исключение — он тоже добавляет свою конфигурацию запуска в набор. При первом запуске нашего приложения Eclipse и ADT создали новую конфигурацию Android Application Run с параметрами по умолчанию.

Чтобы получить доступ к конфигурации запуска вашего проекта, выполните следующие действия.

1. Щелкните правой кнопкой мыши на проекте в представлении Package Explorer (Диспетчер пакетов) и выберите Run As ▸ Run Configurations (Выполнить как ▸ Выполнить конфигурацию).
2. Выберите проект `hello world` из списка слева.
3. В правой части окна вы можете изменить название конфигурации запуска, а также скорректировать другие настройки на вкладках Android, Target (Цель) и Commons tabs (Общие вкладки).
4. Для переключения развертывания из автоматического в ручной режим перейдите на вкладу Target (Цель) и выберите Manual (Вручную).

Теперь при запуске приложения вам будет предложено выбрать подходящий эмулятор или устройство для развертывания. Этот диалог показан на рис. 2.9. Для наглядности я добавил несколько виртуальных устройств с разными версиями целевой ОС, а также подключил два реальных устройства.

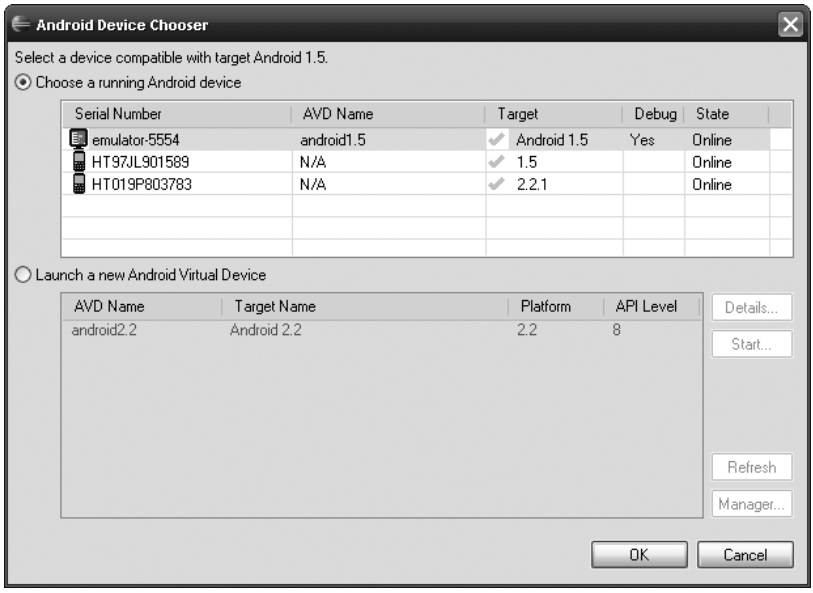

Рис. 2.9. Выбор эмулятора/устройства для запуска приложения

Диалог показывает все запущенные эмуляторы и подключенные в данный момент устройства, также остальные AVD, не запущенные в данный момент.

Отладка приложения

Иногда приложение ведет себя неожиданно или перестает работать. Чтобы определить причину неприятностей, необходима возможность отладки программы. Eclipse и ADT предлагают невероятно мощные возможности для приложений Android. Мы можем устанавливать в коде точки прерывания, получать значения переменных, текущее состояние стека и многое другое.

Перед тем как начать использовать отладку, необходимо подкорректировать файл `AndroidManifest.xml`. Этот момент — своего рода проблема курицы и яйца, поскольку ранее мы не изучали файлы манифестов. На данном этапе нам достаточно знать, что файл манифеста определяет некоторые атрибуты нашего приложения. Один из них — возможность отладки приложения. Данный параметр задан в форме XML-атрибута тега `<application>`. Для включения возможности отладки мы просто добавляем следующий атрибут тегу `<application>` в файле манифеста:

```
android:debuggable="true"
```

В процессе разработки приложения вы можете оставить этот атрибут в файле манифеста. Однако не забудьте убрать его, перед тем как передавать пакет на Android Market.

Теперь, включив для приложения возможность отладки, вы можете реализовать ее на эмуляторе или устройстве. Обычно это выражается в установке точек прерывания для анализа состояния программы на определенных этапах. Чтобы установить точку прерывания, откройте файл программного кода в Eclipse и сделайте двойной щелчок в серой зоне перед той строкой кода, которая вам нужна. Для демонстрации этой возможности сделаем это в строке 23 класса `HelloWorldActivity`. Это заставит отладчик останавливать ход выполнения программы при каждом нажатии экранной кнопки. Точка прерывания отмечается в редакторе кода маленьким кружком перед строкой, на которой вы ее установили (рис. 2.10). Чтобы убрать точку прерывания, снова сделайте двойной щелчок на ней в редакторе кода.

```
    public void onClick(View v) {
        touchCount++;
        button.setText("Touched me "+touchCount+" time(s)");
    }
}
```

Рис. 2.10. Установка точки прерывания

Запуск отладки очень похож на процесс запуска приложения, описанный выше. Щелкните правой кнопкой мыши на проекте в представлении Package Explorer (Диспетчер пакетов) и выберите Debug As ▸ Android Application (Отладка ▸ Приложение Android). Таким образом вы создадите новую конфигурацию отладки для вашего проекта (точно так же, как вы делали при простом запуске программы). Вы можете изменить настройки по умолчанию для данной конфигурации, выбрав Debug As ▸ Debug Configurations (Отладка ▸ Конфигурация отладки) в контекстном меню.

ПРИМЕЧАНИЕ

Вместо применения контекстного меню проекта в представлении Package Explorer (Диспетчер пакетов) вы можете использовать меню Run (Выполнить) для запуска и отладки приложения, а также получать доступ к настройкам.

Если вы стартуете сессию отладки впервые, Eclipse спросит вас, не хотите ли вы переключиться в перспективу Debug (Отладка), на что вы можете соглашаться без колебаний. Рассмотрим эту перспективу. На рис. 2.11 показан ее внешний вид после запуска процесса отладки нашего приложения `hello world`.

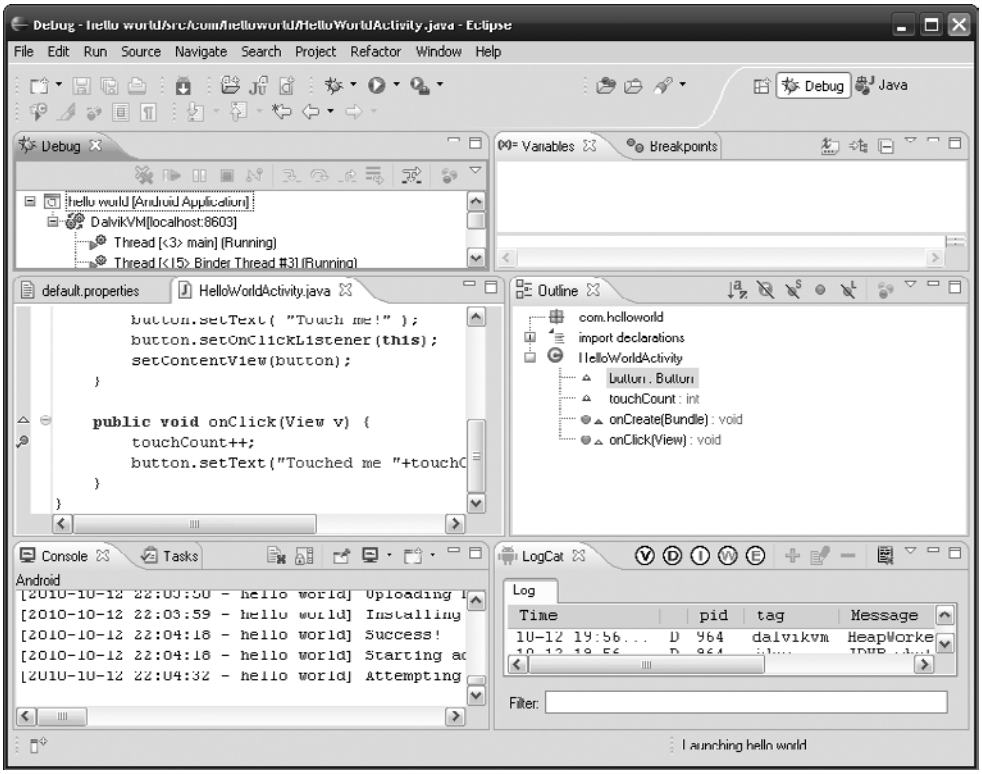

Рис. 2.11. Перспектива Debug (Отладка)

Если вы помните наш краткий обзор Eclipse, то знаете, что в ней существуют несколько перспектив, состоящих из набора представлений для определенных задач. Перспектива Debug (Отладка) по виду сильно отличается от перспективы Run (Выполнить).

- Первое из новых представлений, на которое следует обратить внимание, — Debug (Отладка) в левом верхнем углу. В нем показаны все запущенные в данный момент приложения и стеки всех их потоков, если они запущены в режиме отладки.

- Ниже представления Debug (Отладка) находится представление для редактирования кода, с которым мы уже знакомились при изучении перспективы Java.
- Представление Console (Консоль) выводит сообщения от плагина ADT, информируя нас о том, что происходит.
- Представление LogCat станет одним из наших лучших друзей при разработке приложений. В нем показан журнал сообщений, поступающих от системных компонентов, других приложений и нашей программы. В нем также можно увидеть трассировку стека, если приложение выйдет из строя, и наши собственные сообщения в реальном времени. Более подробно LogCat будет рассмотрено в следующем разделе.
- Представление Outline (Контур) не очень полезно в данной перспективе. Вы, скорее всего, будете изучать переменные и точки прерывания, и текущее положение в программе вам будет ни к чему. Я обычно закрываю это представление из перспективы Debug (Отладка), чтобы оставить больше места для других.
- Представление Variables (Переменные) особенно полезно для отладочных целей. Когда отладчик достигает точки прерывания, у нас появляется возможность изучить и изменить переменные в текущем контексте программы.
- Наконец, представление Breakpoints (Точки прерывания) демонстрирует список установленных нами точек прерывания.

Если вы любознательны, то, вероятно, уже нажали кнопку в работающем приложении, чтобы увидеть работу отладчика. Он остановится на строке 23 в соответствии с установленной точкой прерывания. Вы также можете заметить, что в представлении Variables (Переменные) появились переменные текущего блока программы, состоящего из самой активности (this) и параметра метода (v). Раскрывая список переменных, вы можете исследовать их более детально.

Представление Debug (Отладка) показывает трассировку стека, относящегося к текущему методу. Обратите внимание: у вас может быть несколько запущенных потоков, любой из которых вы можете приостанавливать в любое время в представлении Debug (Отладка). Строка, на которой установлена точка прерывания, подсвечивается, указывая позицию в коде программы.

Вы можете приказать отладчику выполнить текущее выражение (нажав F6), зайти в методы, вызываемые текущим методом (нажав F5), или продолжить выполнение программы обычным образом (нажав F8). Добиться тех же целей можно также, используя меню Run (Выполнить). Учтите, что на самом деле параметров отладки больше, чем я вам сейчас рассказал. Как и всегда, я предлагаю вам самим поэкспериментировать над тем, что вам нужно.

ПРИМЕЧАНИЕ

Любознательность — основной строительный материал для успешной разработки Android-игр. Вам придется действительно близко познакомиться со средой разработки, чтобы получить от нее наибольшую отдачу. Книга не может описать все подробности Eclipse, поэтому я вновь призываю вас экспериментировать.

LogCat и DDMS

Плагин ADT устанавливает много новых представлений и перспектив для использования в Eclipse. Одно из самых полезных представлений (о котором вскользь упомянуто в предыдущем разделе) называется LogCat.

LogCat — система журналирования событий в Android, позволяющая системным компонентам и приложениям выводить информацию на различных уровнях. Каждая запись в журнале состоит из даты, времени, уровня журналирования, ID процесса-источника записи, тега (определяемого приложением самостоятельно) и собственно сообщения.

Представление LogCat собирает и выводит эту информацию с подключенного эмулятора или реального устройства. На рис. 2.12 показан пример вывода в представлении LogCat.

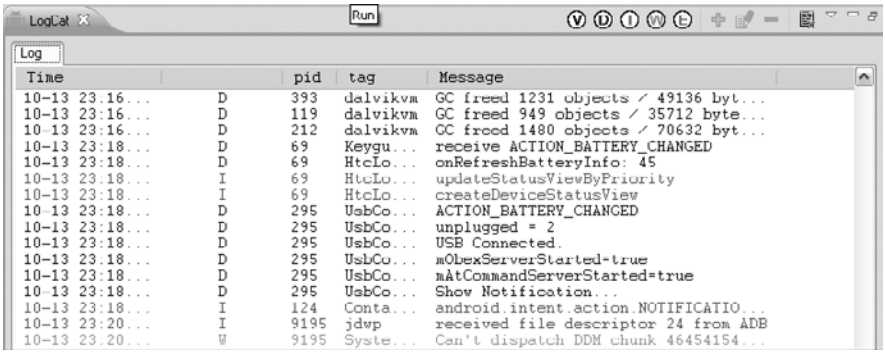

Рис. 2.12. Представление LogCat

Обратите внимание на кнопки в верхнем правом углу LogCat.

- Первые пять из них позволяют выбрать уровни журналирования, которые вы хотите увидеть.
- Кнопка «зеленый плюс» дает возможность определить фильтр, основанный на теге, ID процесса, уровне журналирования. Он очень поможет вам, если вы захотите увидеть лог только вашего приложения (которое, вероятно, будет использовать особый тег).
- Остальные кнопки позволяют редактировать и изменять фильтр, а также очищать окно вывода.

Если подключено одновременно несколько устройств/эмуляторов, LogCat будет выводить информацию только от одного из них. Чтобы получать более подробную информацию, вы можете воспользоваться перспективой DDMS.

DDMS (Dalvik Debugging Monitor Server) предлагает более разнообразные сведения о процессах и виртуальных машинах Dalvik, запущенных на всех подключенных устройствах. Переключиться на перспективу DDMS можно в любой момент с помощью пункта меню Window ▸ Open Perspective ▸ Other ▸ DDMS (Окно ▸ Открыть

перспективу ▸ Другие ▸ DDMS). На рис. 2.13 показано, как обычно выглядит перспектива DDMS.

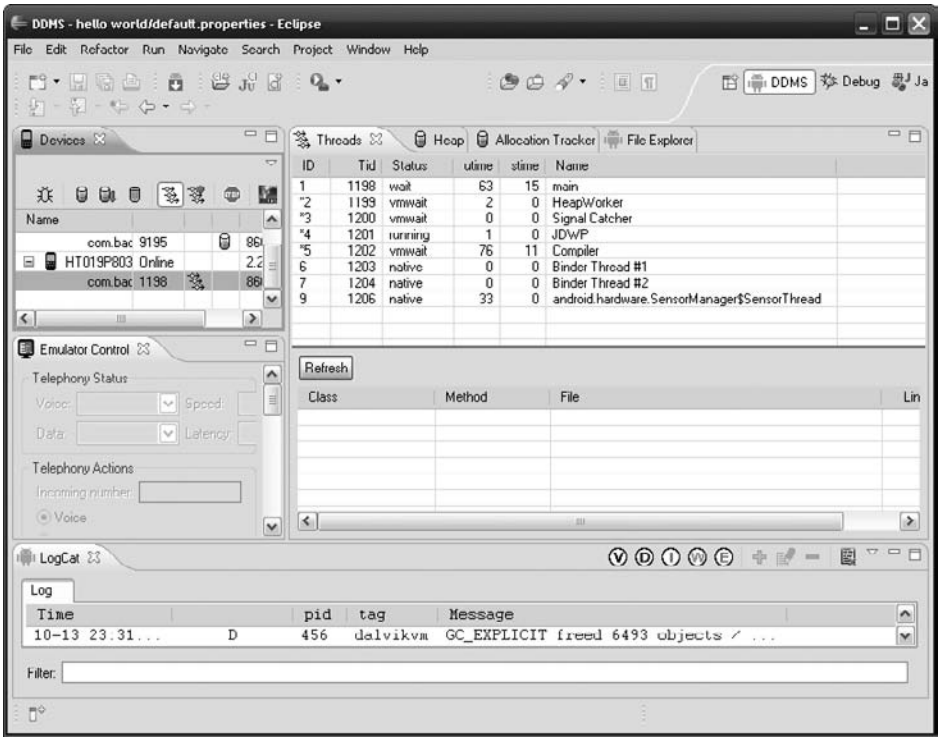

Рис. 2.13. DDMS в действии

Как и в других перспективах, в нашем распоряжении несколько специальных представлений, выполняющих для нас полезную работу. В данном случае мы хотим собирать информацию обо всех процессах, их виртуальных машинах, потоках, текущем состоянии памяти, данных от LogCat по конкретному устройству и т. д. Рассмотрим эти представления.

- Devices (Устройства) — демонстрирует все подключенные устройства и эмуляторы, а также работающие на них процессы. С помощью кнопок на панели инструментов вы можете выполнять различные действия: производить отладку выбранного процесса, записывать состояние памяти и данные от потоков, а также делать снимки экрана.
- LogCat — аналогично описанному в предыдущем разделе, с одним отличием: выводит информацию от устройства, выбранного в представлении Devices (Устройства).
- Emulator Control (Контроль эмулятора) — позволяет изменять поведение запущенного экземпляра эмулятора. Вы можете, например, заставить его гененерировать пробные GPS-координаты для тестирования.

- Threads (Потоки) — выводит данные о потоках, используемых процессом, который выбран в представлении Devices (Устройства). Информация будет показываться, только если вы включили отслеживание потоков (это можно сделать, нажав пятую слева кнопку в представлении Devices (Устройства)).
- Heap (Куча) (не показано на рис. 2.13) — снабжает информацией о статусе памяти устройства. Как и в случае с данными о потоках, необходимо явно включить отслеживание состояния памяти в представлении Devices (Устройства) нажатием второй слева кнопки.
- Allocation Tracker (Отслеживание распределений) — показывает использованные недавно классы. Это очень помогает при борьбе с утечками памяти.
- File Explorer (Проводник) — позволяет изменять файлы на подключенном Android-устройстве или экземпляре эмулятора. Вы можете перетаскивать файлы в это представление так же, как вы делаете это при работе с операционной системой.

DDMS на самом деле — отдельное приложение, интегрированное в Eclipse с помощью плагина ADT. Вы можете запускать его и отдельно из каталога $ANDROID_HOME/tools directory (%ANDROID_HOME%/tools при использовании Windows). Оно не подключается к устройствам напрямую, применяя для этого Android Debug Bridge (ADB) — еще одну утилиту, включенную в SDK. Рассмотрим ее, чтобы дополнить наши знания о среде разработке Android.

Использование ADB

ADB позволяет управлять подключенными устройствами и экземплярами эмулятора. Она состоит из трех различных компонентов.

- Клиентское приложение, запущенное на машине разработчика с помощью команды adb (эта команда будет работать, если вы правильно настроили ваши переменные окружения). Когда мы говорим об ADB, то имеем в виду как раз эту утилиту командной строки.
- Сервер, также запущенный на компьютере разработчика. Он устанавливается в виде фонового сервиса и отвечает за соединение между программой ADB и любым подключенным устройством или экземпляром эмулятора.
- Демон ADB, также являющийся фоновым процессом и запускающийся на каждом телефоне или эмуляторе. Сервер ADB использует этот демон для подключения.

Обычно мы применяем ADB через DDMS, игнорируя существование отдельной утилиты командной строки. Но иногда бывает полезно запускать его отдельно, поэтому кратко изучим некоторые его функции.

ПРИМЕЧАНИЕ

Для получения полного справочника доступных команд обратитесь к документации по ADB на сайте Android Developers (http://developer.android.com).

Весьма полезная задача, выполняемая с помощью ADB, — получение списка всех подключенных к ADB-серверу (а значит, и к компьютеру) устройств и эмуляторов. Для этого выполните следующую команду консоли (обратите внимание: символ > не является частью команды):

```
> adb devices
```

В результате на экран будет выведен список всех подключенных устройств и эмуляторов с соответствующими серийными номерами:

```
List of devices attached
HT97JL901589    device
HT019P803783    device
```

Серийный номер устройства или эмулятора используется для выполнения последующих команд. Например, следующая команда установит APK-файл `myapp.apk` с машины разработчика на устройство с серийным номером HT019P803783:

```
> adb -s HT019P803783 install myapp.apk
```

Аргумент `-s` может использоваться с любой командой ADB, выполняющей какие-либо действия с конкретным устройством.

Существуют также команды для копирования файлов между компьютером и устройством (эмулятором). Результат действия следующей команды — копирование локального файла `myfile.txt` на карту памяти устройства с серийным номером HT019P803783:

```
> adb -s HT019P803783 push myfile.txt  /sdcard/myfile.txt
```

Чтобы произвести обратное копирование `myfile.txt` с карты памяти, используйте следующий набор символов:

```
> abd pull /sdcard/myfile.txt myfile.txt
```

Если к ADB-серверу в данный момент подключено всего одно устройство или эмулятор, вы можете опустить серийный номер — `adb` определит его автоматически.

Конечно, возможности ADB не ограничиваются описанными нами функциями. Многие из них реализуются через DDMS, и в большинстве случаев мы не будем применять командную строку. Однако для небольших задач она бывает идеальным решением.

Подводя итог

Среда разработки Android иногда может вызывать страх. К счастью, для начала работы вам необходима только часть всего функционала, и последние несколько страниц этой главы должны были дать вам достаточно информации, чтобы вы могли начать писать программный код.

Самое главное, что вы должны понять после изучения этой главы, — как все это работает вместе. JDK и Android SDK служат основой всей разработки для Android. Они предлагают инструменты для компилирования, развертывания и запуска приложений на экземплярах эмулятора и устройствах. Для ускорения процесса разработки мы используем Eclipse в сочетании с плагином ADT, избавляющим нас от неудобной работы с JDK и SDK в командной строке. Сам Eclipse построен на нескольких корневых концепциях: рабочих пространствах, управляющих проектами; представлениях, предлагающих особую функциональность (например, редактирование программного кода или вывод `LogCat`); перспективах, объединяющих представления для выполнения определенных задач (например, отладки); конфигурациях запуска и отладки, позволяющих определить параметры запуска или отладки приложения.

Обязательное условие овладения всем этим богатством — практика, как бы скучно это ни звучало. В ходе изучения данной книги мы реализуем проекты, которые познакомят вас со средой разработки Android. Однако не стоит на этом останавливаться — только от вас зависит, делать ли следующие шаги.

Теперь, когда вы усвоили всю эту информацию, можете двигаться вперед к тому, ради чего все это затеяли: разработке игр.

3 Разработка игр 101

Разработка игр — непростой процесс. Не столько из-за сложных расчетов, сколько из-за объема информации, которую вам необходимо переварить, прежде чем начать создавать игру вашей мечты. Программисту нужно беспокоиться о таких обыденных аспектах, как файловый ввод-вывод, обработка ввода, работа со звуком и графикой и поддержка сети. И это только начало! Решив все эти вопросы, вам захочется создать механику игры. Это тоже требует определенного осмысления: вы должны будете решить, на каких принципах будет построен ваш игровой мир. Сможете ли вы обойтись без физического движка, самостоятельно создав простую симуляцию? Кто и как будет жить в вашем игровом мире и как все это будет отображаться на экране?

Есть еще одна проблема, о которой часто забывают новички: перед тем как вы начнете программировать, необходимо подумать о дизайне. Бесчисленное множество проектов бесславно сгинуло на этапе предварительной демоверсии из-за того, что у их авторов отсутствовали идеи, как должна выглядеть и вести себя игра. И я сейчас говорю не об основной игровой механике вашей стрелялки от первого лица (тут как раз все просто — крестовина и мышь, и дело в шляпе). Вы должны задать себе множество вопросов. Будет ли в игре приветственный экран? Во что он будет переходить? Что игрок увидит на главном экране игры? Какие элементы будут доступны ему на основном игровом экране? Что произойдет при нажатии кнопки «Пауза»? Какие параметры будут доступны в экране настроек? Как пользовательский интерфейс будет выглядеть на экранах с различным разрешением и соотношением сторон?

Что интересно, однозначного ответа на все эти вопросы не существует. Я не претендую на то, чтобы дать вам универсальный ключ к разработке игр. Вместо этого я попытаюсь проиллюстрировать обычный процесс создания игрового приложения. Вы можете применять мой опыт, что называется, «один в один» или изменить его в соответствии с вашими потребностями. Тут нет жестких правил — если вы довольны результатом, то все в порядке. Но при этом вы должны стремиться к тому, чтобы ваше решением было простым.

Жанры: на любой вкус

Начиная проект, вы обычно решаете, каким будет жанр вашей игры. Если вы не планируете заняться чем-то абсолютно новым и ранее невиданным, то шансы на то, что ваша игровая идея уложится в один из популярных в данный момент жанров, весьма велики. Большинство жанров используют уже устоявшиеся принципы игровой механики (схемы управления, определенные цели и т. д.) Отход от этих стандартов может сделать игру настоящим хитом — геймеры все время ищут что-то новенькое. Но это и большой риск, так что постарайтесь, чтобы ваши платформер/шутер/стратегия реального времени имели аудиторию.

Рассмотрим несколько примеров наиболее популярных игр с Android Market.

Казуальные игры

Вероятно, крупнейший сегмент игр на Android Market состоит из так называемых казуальных игр. Что это такое? Вопрос не имеет конкретного ответа, но казуальные игры обычно обладают рядом общих черт. Чаще всего они очень легки в освоении (так что не только игроманы разбираются в них быстро), что увеличивает круг потенциальной аудитории. Сеанс игры обычно длится несколько минут (хотя простота игр часто завораживает игроков, заставляя их проводить за ней часы). Игровая механика варьируется от почти отсутствующей (игры-головоломки) до простой (однокнопочные платформеры вроде бросания бумаги в корзину). Из-за столь размытого определения казуальных игр они имеют практически неограниченное разнообразие и возможности.

Abduction и Abduction 2 (рис. 3.1), созданные состоящей из одного человека компанией Psym Mobile, — прекрасный пример казуальной игры. Она принадлежит к поджанру «подбрось их» (по крайней мере я так его называю). Цель игры — направлять постоянно прыгающую корову от платформы к платформе для достижения верхнего уровня. На этом пути вы будете сталкиваться с ломающимися платформами, зубцами и летающими противниками. Для преодоления этих препятствий вы можете воспользоваться бонусами. Управление коровой производится тряской телефона: таким способом регулируется направление ее прыжка/падения. Интуитивное управление, ясная конечная цель и забавная графика сделали эту игру одним из первых хитов на Android Market.

Antigen (рис. 3.2) от Battery Powered Games — совсем другой вид игры. Вы играете за антитело, сражающееся с различными видами вирусов. Эта игра — гибрид головоломки и аркады. Управление антителом осуществляется экранным джойстиком и кнопками вращения, расположенными в правом верхнем углу. Антитело оснащено набором разъемов на каждой стороне, позволяющих ему подключаться к вирусам и таким образом их уничтожать — простая, но весьма привлекательная концепция. В отличие от Abduction, использующей только акселерометр в качестве механизма управления, органы управления Antigen несколько сложнее. Поскольку некоторые устройства не поддерживают мультитач, разработчики предложили несколько схем управления для всех возможных устройств (включая Zeemote).

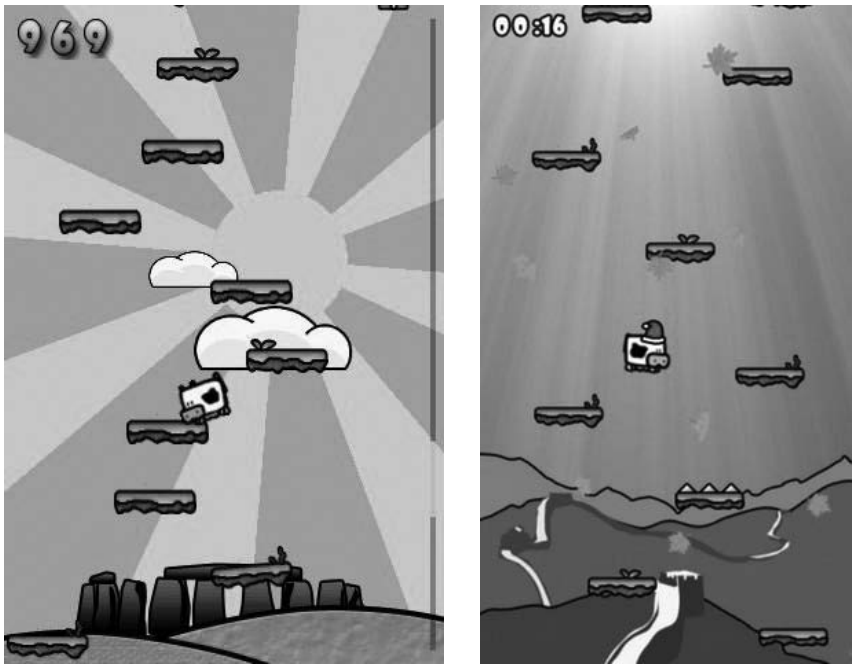

Рис. 3.1. Abduction *(слева)* и Abduction 2 *(справа)* от Psym Mobile

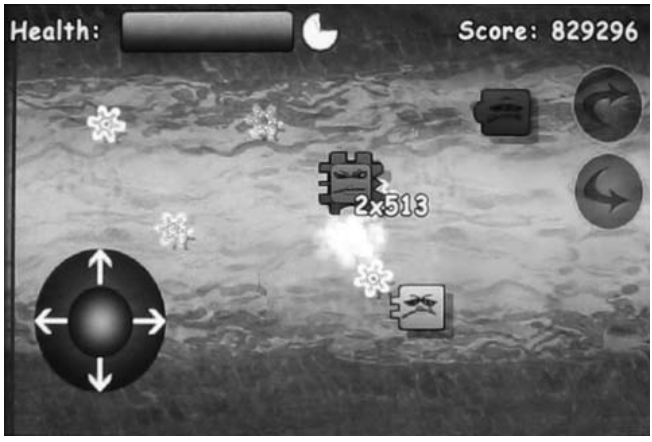

Рис. 3.2. Antigen от Battery Powered Games

Для привлечения максимальной аудитории особое внимание было уделено возможности игры даже на бюджетных аппаратах с экранами 320 × 240.

Перечисление всех возможных поджанров категории казуальных игр заняло бы большую часть этой книги. В этой сфере можно найти много инновационных концепций, и для получения представления о них лучше использовать Android Market.

Головоломки

Этот жанр не нуждается в представлении. Всем знакомы такие названия, как «Тетрис» и Bejeweled. Игры такого рода составляют довольно большую часть игрового рынка Android и крайне популярны среди всех категорий игроманов. В отличие от головоломок для PC, игры для Android отходят от классической формулы «3 в ряд» и используют более тщательную, основанную на физике концепцию.

Super Tumble — великолепный пример физической головоломки (рис. 3.3). Цель игры — убрать блоки, касаясь их, и доставить на нижнюю платформу звезду, находящуюся на вершине. Это звучит очень просто, но с каждым новым уровнем добиться цели все сложнее. Игра разработана с помощью Box2D — движка 2D-физики.

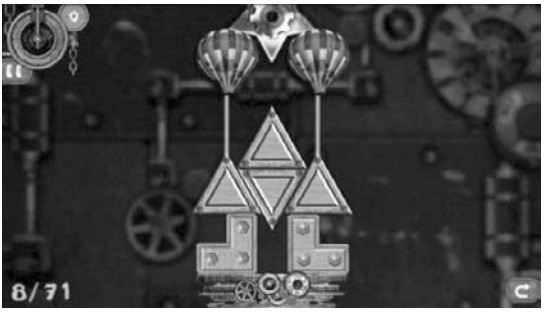

Рис. 3.3. Super Tumble от Camel Games

U Connect от BitLogik — минималистичная, но весьма занимательная задачка для мозгов (рис. 3.4). Цель игры — соединить все точки графа в одну линию. Студентам вычислительных факультетов такая задача будет весьма знакома.

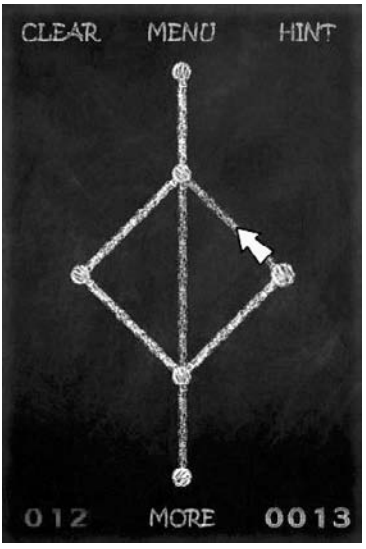

Рис. 3.4. U Connect от BitLogik

Безусловно, вы можете найти в Android Market множество клонов «Тетриса», игр «3 в ряд» и других стандартных алгоритмов. Описанные выше игры демонстрируют тот факт, что головоломка может быть больше, чем очередным клоном 20-летней идеи.

Аркады и активные игры

Аркады и активные игры обычно реализуют весь потенциал платформы Android. Многие из них предлагают ошеломляющие 3D-визуализации, демонстрирующие возможности нынешнего поколения аппаратной части. Этот жанр включает в себя множество подкатегорий (гонки, стрелялки от первого и третьего лица, платформеры).

Replica Island — возможно, самый успешный на момент написания книги платформер для Android (рис. 3.5). Он был создан инженером Google Крисом Прюэттом, чтобы продемонстрировать возможность написания высокопроизводительных игр на чистой Java под Android. Игра стремится соответствовать всем потенциальным конфигурациям устройств и методам ввода. Особое внимание было уделено работе Replica Island на бюджетных устройствах. Главный персонаж — робот, которому дано задание захватить волшебный артефакт. Механика игры напоминает старые 16-битные платформеры для приставки SNES. В стандартной конфигурации робот управляется акселерометром и двумя кнопками (одна включает двигатель для перепрыгивания препятствий, вторая предназначена для прыжка на врагов сверху). Еще один плюс — игра поставляется с открытыми кодами.

Рис. 3.5. Replica Island Криса Прюэтта

ExZeus компании HyperDevBox — классический шутер в духе Starfox для SNES с высококачественной 3D-графикой (рис. 3.6). В игре есть все: различные виды оружия, бонусы, схватки с боссами и множество предметов, в которые можно стрелять. Как и в случае с большинством 3D-игр, в ExZeus можно играть только

на устройствах высшего ценового диапазона. Управление главным персонажем осуществляется с помощью тряски и экранных кнопок — весьма интуитивный метод для данного типа игр.

Рис. 3.6. ExZeus от HyperDevBox

Deadly Chambers от Battery Powered Games — шутер от третьего лица в стиле таких классических образцов жанра, как Max Payne или Tomb Rider (рис. 3.7). Главный персонаж Dr. Chambers пытается выбраться из подземелья злобного волшебника. Battery Powered Games действует по негласному стандарту жанра — игра почти не имеет предыстории. Но кому она нужна, если вы получаете возможность, не раздумывая, убивать все на вашем пути, пользуясь для этого отличным набором эксклюзивного оружия? Управление главным героем производится через экранный джойстик. Дополнительные кнопки позволяют игроку переключаться в режим от первого лица для более точного прицеливания, выбирать вид оружия и т. д. В отличие от ExZeus, разработчик приложил немало усилий, чтобы игра была доступна и на бюджетных устройствах. Кроме того, игра предлагает различные варианты организации управления, поэтому в нее можно играть и на устройствах без мультитач. Вообще, создание данной игры можно назвать подвигом, особенно если учесть, что ее написал один человек за шесть месяцев.

Radiant компании Hexage представляет собой блестящее эволюционное развитие древней концепции Space Invaders (рис. 3.8). Отойдя от идеи статического игрового поля, игра предлагает сдвигающиеся в сторону уровни, а также большое разнообразие дизайнов уровней и противников. Вы управляете космическим

Рис. 3.7. Deadly Chambers производства Battery Powered Games

кораблем, тряся телефон, при этом у вас есть возможность улучшить ваше вооружение, покупая новое за заработанные вами очки. Полупиксельный стиль графики придает игре уникальный внешний вид, напоминая о старых добрых временах.

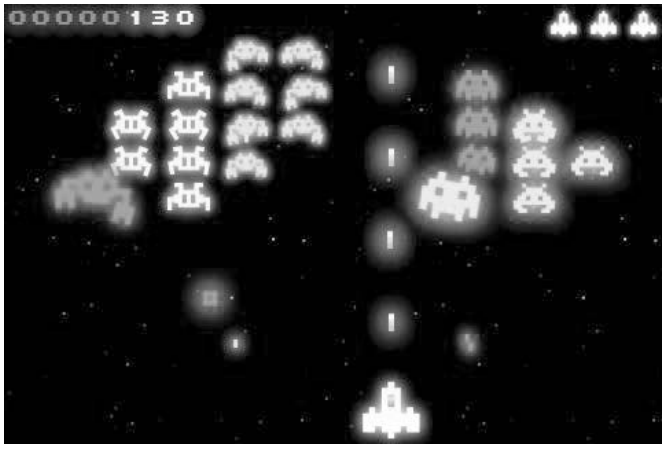

Рис. 3.8. Radiant от Hexage

«Защита башни»

Зная о большом успехе такого рода программ для платформы Android, я чувствую необходимость обсудить игры вида «защита башни» в качестве отдельного жанра. Они стали популярны как варианты десктопных стратегий реального времени, разработанные сообществом энтузиастов. Идея вскоре была реализована и для

отдельных игр, в настоящее время являющихся самыми продаваемыми на платформе Android.

В типичной игре жанра «защита башни» какая-нибудь злодейская сила насылает различных тварей для атаки на ваш замок/базу/дом/что угодно. Ваша задача — защитить это особое место на игровой карте, размещая защитные башни для стрельбы по нападающим. За каждого убитого врага вы обычно получаете некоторое количество денег или очков, которые можно инвестировать в строительство новых или обновление существующих башен. Идея крайне проста, однако достижение оптимального баланса в таких играх — весьма непростая задача.

Robo Defense от Lupis Labs Software — прародитель всех игр жанра «защита башни» для Android (рис. 3.9). Большую часть жизни платформы она занимает первое место по продажам среди игр. Игра следует стандартной для жанра формуле, безо всяких ненужных украшательств. Это очень простая и крайне привлекательная реализация с различными картами, достижениями и таблицей лучших результатов. Презентация достаточна для того, чтобы вкратце понять концепцию, но не чересчур сложна, что убеждает нас в том, что хорошо продаваемая игра необязательно должна быть сложной в визуальном и звуковом отношении.

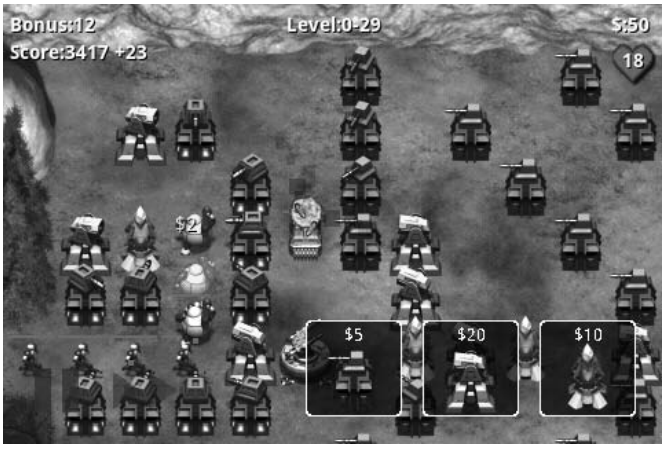

Рис. 3.9. Robo Defense от Lupis Labs Software

Что-то новенькое

Некоторые игры сложно отнести к какой-то определенной категории — они используют новые возможности и особенности устройств Android (например, камеру и GPS) для создания необычных инновационных решений. Область внимания этих игр — социальные сети и геолокация, а также некоторые элементы дополненной реальности.

SpecTrek — один из победителей второго Android Developer Challenge (рис. 3.10). Цель игры — хождение с включенным GPS для нахождения призраков и фиксации их с помощью камеры. Призраки просто накладываются на изображение видоис-

кателя камеры, и задачей игрока является держать их в фокусе и нажимать кнопку Catch (Поймать) для получения очков.

Рис. 3.10. SpecTrek от SpecTrekking.com

Теперь, когда вы знаете о том, что уже сейчас доступно для Android, я советую запустить приложение Android Market и посмотреть, какие игры появились там в последнее время. Обратите внимание на их структуру (в каком порядке идут игровые экраны, с помощью каких кнопок это делается, как игровые элементы взаимодействуют друг с другом, и т. д.). Составить впечатление о подобных вещах можно именно в процессе игры, если играть, так сказать, с позиции критика. Отбросьте на время развлекательный аспект и попытайтесь разложить игру по полочкам. Когда вам это удастся, возвращайтесь к книге и продолжайте.

Для начала нарисуем нашу простую игру на бумаге.

Дизайн игры: карандаш сильнее кода

Как я говорил ранее, очень соблазнительно сразу запустить IDE и сколотить впечатляющее демо. Вполне нормально написать код-прототип игровой механики и посмотреть на его работу. Однако сразу после этого забудьте о нем. Возьмите карандаш, пачку бумаги, садитесь на удобный стул и начинайте думать о высокоуровневых аспектах вашей игры. Не стоит пока концентрироваться на технических подробностях — вы займетесь этим позже. Прямо сейчас вам необходимо сосредоточиться на разработке пользовательского интерфейса игры. Для меня лучший способ сделать это — нарисовать эскизы главных компонентов:

- основную механику игры;
- черновой сюжет и основные персонажи;
- черновые наброски графического оформления игры;
- эскизы всех экранов, а также схема переходов между ними вместе с инициаторами переходов (например, окончание игры).

Если вы заглядывали в содержание, то уже поняли, что мы собираемся написать Android-вариант знаменитой «Змейки» (Snake). Это одна из самых популярных в истории игр для мобильных телефонов. Если вы никогда не слышали о «Змейке», поищите ее в Сети, я подожду...

Итак, возвращаемся. Теперь, когда вы знаете суть «Змейки», притворимся, что эта великолепная идея пришла в голову нам. Будем придумывать дизайн игры. Начнем с основной механики игры.

Основная механика игры

Перед тем как заняться этим важным делом, проверьте наличие всего необходимого. У вас должны быть:

- ножницы;
- что-то, чем вы будете рисовать;
- много бумаги.

На данном этапе разработки дизайна игры нам хватит и этого. Вместо того чтобы тщательно прорисовывать изображения в Paint, Gimp или Photoshop, я предлагаю создать черновые варианты на бумаге и составить из них таблицу. Мы легко сможем перемещать листы с эскизами, не мучаясь с мышкой в редакторе. Когда дизайн нас удовлетворит, мы отсканируем или сфотографируем эскизы для использования в будущем.

Начнем с создания набросков базовых компонентов. На рис. 3.11 продемонстрирована моя версия элементов, необходимых для обеспечения основной механики игры.

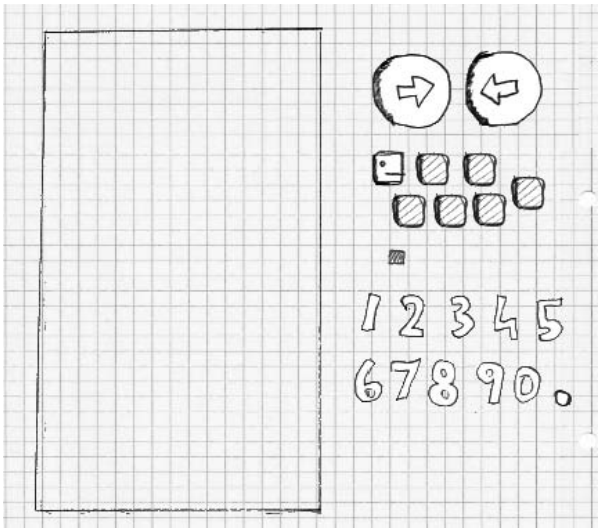

Рис. 3.11. Основные элементы дизайна

Прямоугольник слева — экран (размером примерно как у моего Nexus One). Здесь мы будем размещать все остальные элементы. Следующие компоненты — две кнопки, с помощью которых мы будем управлять змейкой. Как обойтись без ее головы, нескольких хвостовых звеньев и того, что она будет есть? Кроме того, я нарисовал (и потом вырезал) здесь несколько цифр, которые пригодятся нам для демонстрации результатов. Рисунок 3.12 иллюстрирует мое видение стартового игрового поля.

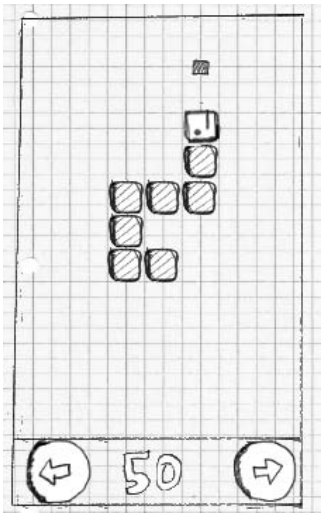

Рис. 3.12. Стартовое игровое поле

Определимся с общей механикой: змейка движется в том направлении, куда смотрит ее голова, волоча за собой хвост. Голова и хвост созданы из частей одинакового размера, лишь немного различающихся внешне. При достижении змейкой границ экрана она появляется с его противоположной стороны. При нажатии кнопки `Вправо` или `Влево` змейка поворачивается на 90° по часовой или против часовой стрелки соответственно. Если змея кусает себя за хвост, игра заканчивается. При касании элемента головой он исчезает, счет увеличивается на 10, а на не занятом змейкой участке экрана появляется новый. Змейка при этом увеличивается в размерах — ее хвост растет.

Довольно подробное описание для столь простой игры, не правда ли? Обратите внимание — я расположил элементы игры в соответствии с их сложностью, поэтому описание поведения игры при пожирании змейкой элемента оставил напоследок. Более продуманные игры, конечно, не могут быть описаны так быстро. Обычно разработчик разбивает элементы игры на несколько групп и разрабатывает их по отдельности, впоследствии соединяя в единое целое. Игра имеет явное условие завершения: она заканчивается при заполнении всего пространства экрана змейкой.

Теперь, когда наша абсолютно оригинальная игра обрела механику, попробуем придумать для нее сюжет.

Сюжет и оформление

Хотя различные эпические истории про зомби, космические корабли, гномов и взрывы будоражат наше воображение, нужно понимать, что наши возможности ограниченны. Как понятно из рис. 3.12, великим художником мне никогда не стать. Я не смогу нарисовать зомби даже под страхом смерти. Поэтому я поступаю как типичный независимый разработчик — использую стиль пятилетнего ребенка.

Итак, добро пожаловать в мир мистера Нома — бумажной змеи, очень любящей поедать капли чернил, падающие неизвестно откуда на его бумажную страну. Мистер Ном крайне эгоистичен и посвятил свою жизнь одной не слишком благородной цели — стать самой большой бумажной змеей в мире, питающейся чернилами!

Эта короткая предыстория позволяет нам определиться с некоторыми моментами, например со стилем оформления игры. Мы сможем отсканировать нарисованные нами элементы, чтобы использовать их при создании графических активов.

Поскольку мистер Ном — индивидуальность, мы слегка изменим его внешность: нарисуем ему змеиную морду. И шляпу. Поедаемые элементы будут выглядеть как набор чернильных пятен. Добавим также игре звук — каждый раз при поедании пятна мистер Ном будем хрюкать. И еще — вместо того, чтобы давать игре скучное название вроде «Рисованная змейка», назовем ее «Мистер Ном», так намного интереснее.

На рис. 3.13 мистер Ном показан во всей красе с несколькими чернильными пятнами, заменяющими безликие элементы для поедания. Я также нарисовал здесь логотип игры, который мы будем неоднократно использовать.

Рис. 3.13. Мистер Ном, его шляпа, чернильные пятна и логотип

Экраны и трансформации

Итак, у нас уже есть основные принципы, предыстория, персонажи и визуальное решение. Мы можем начать рисовать экраны и трансформации между ними. Однако для начала необходимо точно уяснить для себя, что понимается под экраном. В данном случае это неразделимый элемент, занимающий все пространство

дисплея и отвечающий за одну составляющую часть игры (например, главное меню, меню настроек или игровой экран, где все и происходит). Экран может состоять из нескольких компонентов (кнопок, элементов управления, заголовков или нарисованного игрового мира). Экран позволяет игроку взаимодействовать со своими элементами. К числу таких взаимодействий относятся переходы между экранами (например, нажатие пункта New Game (Новая игра) в главном меню делает активным игровой экран или экран выбора уровня). Вооружившись этими определениями, мы можем нарисовать все экраны игры «Мистер Ном».

Первое, что увидят пользователи, запустив нашу игру, — экран главного меню. Из чего состоит удачное главное меню? Хорошим тоном является демонстрация на главном экране названия игры, поэтому мы поместим туда наш логотип. Чтобы экран не выглядел чуждым элементом, стоит добавить фон, который мы возьмем из игрового экрана. Обычно игру запускают для того, чтобы в нее играть, поэтому нелишней будет и кнопка PLAY (Играть) (это будет первый интерактивный компонент экрана). Игроки любят отслеживать свои достижения, поэтому добавим кнопку просмотра лучших результатов (HIGHSCORES), также интерактивную. Возможно, в мире существуют еще люди, не знающие правил игры в «Змейку», — для них мы добавим кнопку вызова помощи (HELP) для перехода к соответствующему экрану. Хотя звуковое оформление нашей игры обещает быть завораживающим, некоторые предпочитают играть в тишине. Специально для таких игроков на экране будет присутствовать включатель/выключатель звука.

Как разместить все эти элементы на экране — дело вкуса. Вы можете изучить подраздел информатики, называющийся «Взаимодействие человека и компьютера» (Human Computer Interfaces, HCI), чтобы познакомиться с новейшими научными воззрениями на этот счет. Но в случае с «Мистером Номом» это выглядит как стрельба из пушек по воробьям. Мой вариант размещения элементов на экране главного меню представлен на рис. 3.14.

Рис. 3.14. Экран главного меню

Обратите внимание — все элементы (логотип, кнопки меню и т. д.) являются отдельными изображениями. Стартуя из экрана главного меню, мы можем переходить на другие экраны. Для «Мистера Нома» нам нужны еще игровой экран, экран лучших результатов и экран помощи. В данном случае можно обойтись без экрана настроек — единственная настройка (переключатель звука) выведена на главный экран.

Давайте пока проигнорируем игровой экран и сконцентрируемся для начала на лучших результатах. Волевым решением я буду хранить их локально, поэтому нам будут доступны достижения только для однопользовательского режима. Кроме того, я обеспечу хранение только пяти лучших результатов. Таким образом, экран лучших достижений будет выглядеть примерно как на рис. 3.15. Он состоит из большой надписи HIGHSCORES (Рекорды) сверху, пяти высших достижений и кнопки со стрелкой, предназначенной для возврата назад. Здесь мы вновь в качестве фона используем фон игрового экрана.

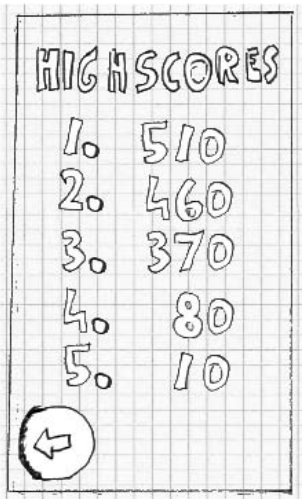

Рис. 3.15. Экран лучших результатов

Следующий экран — помощь. Он расскажет игроку предысторию и пояснит основные принципы игры. В данном случае информации слишком много для размещения на одном экране. Поэтому наш экран помощи будет разделен на несколько экранов, каждый из которых будет содержать важную часть информации для пользователя: кто такой этот мистер Ном и чего он хочет добиться; как им управлять, чтобы он мог поедать кляксы; а также чего мистер Ном не любит делать (а именно, поедать себя). Итого у нас три экрана, как мы видим на рис. 3.16. Заметьте — я добавил кнопку на каждом из них, благодаря которой игрок знает, что доступна дополнительная информация.

И вот мы подошли к главному — игровому экрану. В общем-то, мы видели его и раньше, поэтому нам осталось уточнить лишь несколько деталей. Игра не должна начинаться немедленно — игроку надо дать немного времени, чтобы подгото-

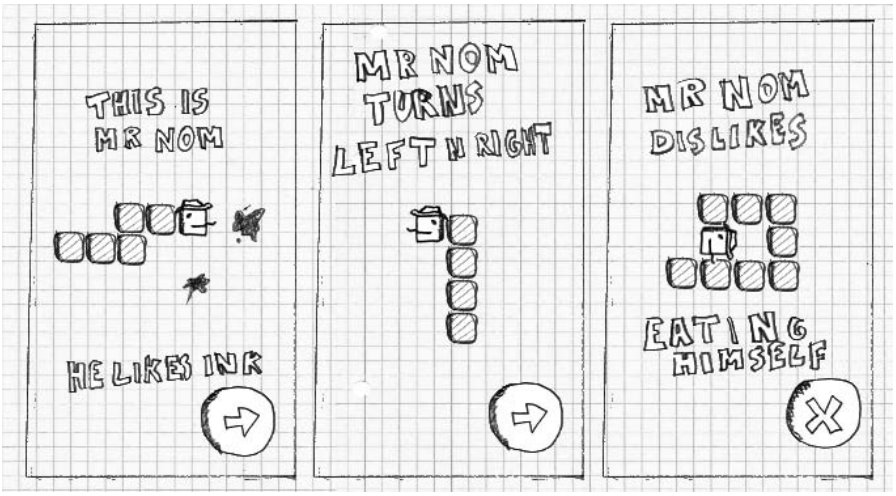

Рис. 3.16. Экран помощи

виться. Поэтому первое, что он увидит при входе в игровой экран, — предложение нажать экран для начала процесса пожирания чернил. Это не означает создания отдельного экрана; мы просто реализуем стартовую паузу на игровом экране.

Кстати, о паузах — нам нужна специальная кнопка для приостановки игрового процесса. А раз есть возможность нажать паузу — должна быть и возможность продолжить. В данном случае для обоих действий нарисуем большую кнопку Resume (Продолжить). В режиме паузы будет показана еще одна кнопка, позволяющая вернуться в экран главного меню.

Если вдруг мистер Ном укусит себя за хвост, игроку надо сообщить об окончании игры. Для этого можно нарисовать отдельный экран или просто показать большое сообщение Game Over (Игра окончена) поверх игрового. В данном случае выберем второй вариант.

Перед тем как закруглиться, покажем, какого результата добился игрок, а также добавим сюда возможность возврата в главное меню. Итого у нас получается четыре подэкрана для игрового экрана: начальной подготовки к игре, собственно для игры, паузы и конца игры. На рис. 3.17 показаны все они.

Теперь пришло время связать экраны между собой. Каждый экран обладает некоторыми интерактивными компонентами, предназначенными для переходов между собой. Из главного меню с помощью соответствующих кнопок мы можем попасть в игровой экран, экран лучших результатов и экран помощи. Из игрового экрана есть возможность вернуться в главное меню нажатием кнопки либо в режиме паузы, либо в состоянии «Игра окончена». Кнопка на экране лучших результатов возвращает нас на главный экран. Экраны помощи оснащены кнопками перехода между собой и в конечном итоге возврата в главное меню.

И это все о переходах! Неплохо, не так ли? Рисунок 3.18 наглядно демонстрирует все переходы с помощью стрелок между интерактивными компонентами. Для большей ясности я также нарисовал здесь все элементы наших экранов.

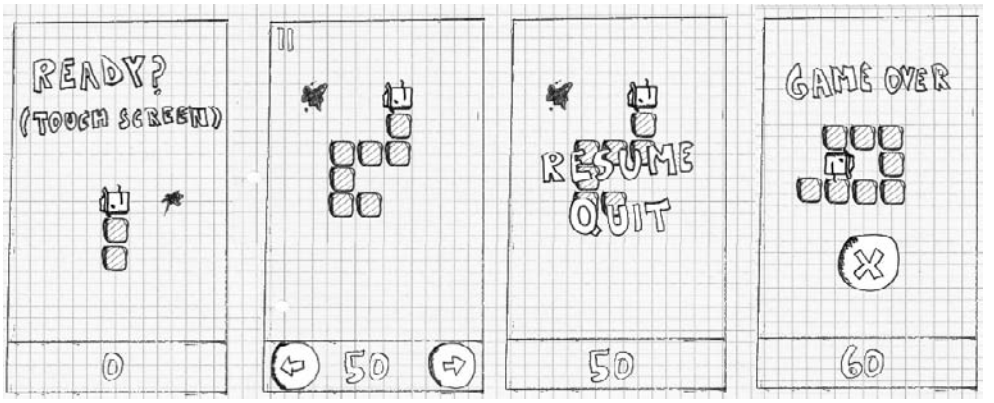

Рис. 3.17. Игровой экран в четырех состояниях

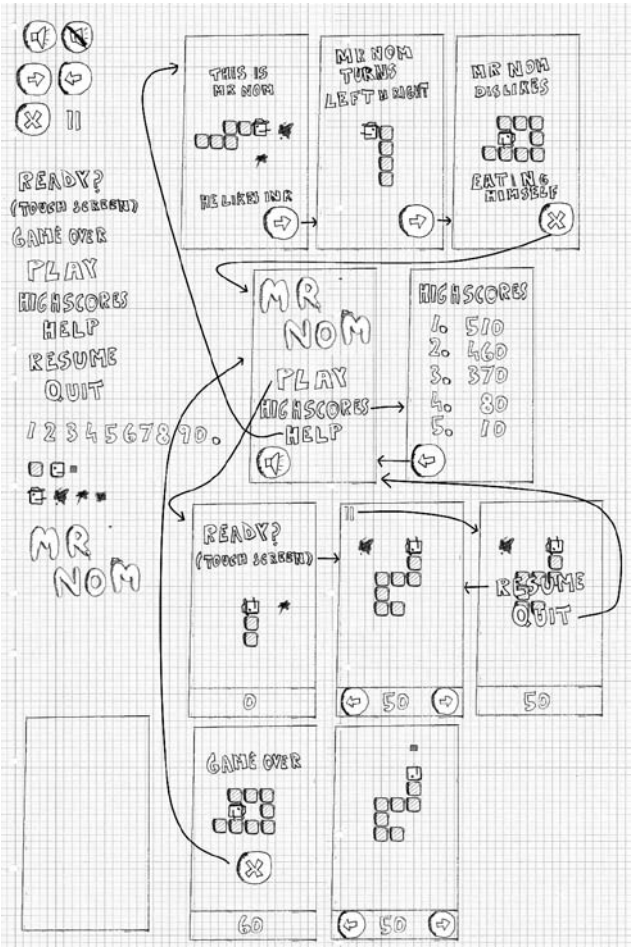

Рис. 3.18. Все элементы дизайна и переходы

Таким образом, дизайн нашей игры можно считать созданным. Осталось всего ничего — реализовать его в виде исполняемого приложения.

ПРИМЕЧАНИЕ

Метод, который мы использовали для создания дизайна, прекрасно подходит для небольших игр. Эта книга предназначена для начинающих разработчиков, и в данном случае такая технология вполне подходит. В больших проектах вы, скорее всего, будете работать в команде с четким разделением обязанностей. Хотя и в таком случае вы можете воспользоваться описанным выше методом, но вам может понадобиться его слегка подкорректировать. К тому же процесс разработки станет более интерактивным из-за того, что дизайн будет постоянно пересматриваться.

Код: скучная рутина

Итак, вы уже знаете, что Android API подходит для разработки игр. Но вам все еще неизвестно, как именно это делать. Уже есть идеи по дизайну игры, однако процесс преобразования его в исполняемый файл пока выглядит неким колдовством. В следующих подразделах я собираюсь сделать для вас обзор составляющих компьютерной игры. Я использую немного псевдокода для интерфейсов, которые мы реализуем позже с помощью Android. Интерфейсы — очень полезная штука по двум причинам: они позволяют нам сконцентрироваться на семантике, не отвлекаясь на детали реализации, а также дают возможность при необходимости менять способ этой реализации (например, вместо использования 2D-визуализации мы для демонстрации мистера Нома можем задействовать OpenGL ES).

Любая игра требует некоей базовой среды, позволяющей абстрагироваться от низкоуровневого взаимодействия с операционной системой. Обычно эта среда разбивается на несколько модулей.

- *Управление окнами*. Отвечает за создание окна и берет на себя такие вещи, как его закрытие или приостановка/возобновление работы приложения на Android.
- *Ввод*. Этот модуль связан с предыдущим и отвечает за отслеживание пользовательского ввода (касаний, клавиатурного ввода и движений акселерометра).
- *Файловый ввод/вывод*. Позволяет приложению получать ресурсы, расположенных на носителе.
- *Графика*. Пожалуй, наиболее сложная часть, если не считать собственно игру. Этот модуль ответственен за загрузку графики и прорисовку ее на экране.
- *Звук*. Загрузка и воспроизведение всего, что способно достичь наших ушей.
- *Игровой фреймворк*. Соединяет в себе все вышеперечисленное и предлагает удобную основу для написания игр.

Каждый из этих модулей состоит из одного или нескольких интерфейсов, каждый из которых должен иметь как минимум одну реализацию, использующую семантику основообразующей платформы (в нашем случае Android).

ПРИМЕЧАНИЕ

Вы не ошиблись — я намеренно не включил поддержку сети в данный список. Мы не будем рассматривать в книге вопросы создания многопользовательских игр. Это весьма сложный вопрос, ответ на который сильно зависит от типа игры. Если вам интересна данная тема, в Сети вы сможете найти много соответствующей информации (www.gamedev.net — прекрасное место для старта).

В последующем мы будем стараться как можно меньше зависеть от конкретной версии платформы — идеи будут одинаковы для всех ее реализаций.

Управление приложением и окнами

Игра, как и любая компьютерная программа, обладает пользовательским интерфейсом, который содержится в окне (если парадигма пользовательского интерфейса основана на окнах, что верно для всех известных операционных систем). Окно выступает в роли контейнера, и мы будем рассматривать его как холст, на котором будет нарисованы все составляющие игры.

Большинство ОС позволяют пользователям взаимодействовать с окнами особым образом (не считая касания пользовательской области или нажатия кнопки). На компьютерных системах вы обычно можете перетаскивать их, изменять их размер и сворачивать в каком-либо варианте Панели задач. В случае с Android изменение размера заменено на смену ориентации, а сворачивание реализовано в виде перехода приложения в фоновый режим при нажатии кнопки Home (Домой) или входящем звонке.

Модуль управления приложением и окнами решает, кроме того, задачу настройки окна и обеспечения заполнения его компонентом интерфейса, воспринимающим пользовательский ввод в виде касаний и нажатий клавиш. Этот компонент может быть визуализирован либо центральным процессором, либо с помощью аппаратного ускорения (как в случае с OpenGL ES).

Модуль управления приложением и окнами не обладает каким-то конкретным набором интерфейсов. Мы объединим его с игровым фреймворком позже. Сейчас нам необходимо запомнить состояния приложения и события окна, которыми необходимо управлять:

- Create (Создать) — возникает один раз при открытии окна (а следовательно, и приложения);
- Pause (Пауза) — появляется при приостановке работы приложения каким-либо способом;
- Resume (Возобновить) — возникает при возобновлении работы приложения и возвращения окна на передний план.

ПРИМЕЧАНИЕ

Некоторые поклонники Android могут в этот момент округлить глаза. Почему используется только одно окно («активность» на языке Android)? Почему не применять для игры более одного виджета, чтобы создавать сложные пользовательские интерфейсы? Главным образом потому, что нам необходим полный контроль над внешним видом и ощущением от игры. Кроме того, в этом случае я могу сосредоточиться на программировании игры для Android вместо программирования интерфейсов для Android. О данной теме вы можете почитать в других источниках.

Ввод

Конечно, пользователю необходимо будет взаимодействовать с игрой каким-либо образом. Именно этим занимается модуль ввода. В большинстве операционных систем события ввода (вроде касания экрана или нажатия кнопки) отсылаются

к текущему окну. Далее это окно перенаправляет данные события элементу интерфейса, на котором в данный момент находится фокус. Процесс перенаправления обычно весьма прозрачен для нас. Все, что нам необходимо, — получать события от сфокусированного компонента интерфейса. API пользовательского интерфейса операционной системы предлагает механизм внедрения в систему перенаправления событий, чтобы мы могли их легко регистрировать и сохранять. Но что нам делать с записанной информацией? Существует два варианта действий.

- *Опрашивание*. При данном способе мы лишь проверяем текущее состояние устройства ввода. Любые изменения состояния между опросами будут для нас потеряны. Данный вид обработки ввода подходит для таких вещей, как проверка факта нажатия пользователем определенной кнопки. Он не годится для отслеживания ввода текста, ведь порядок ввода символов будет утерян.
- *Обработчики событий*. Этот способ дает нам полную хронологию событий, произошедших с момента последней проверки. Он хорошо подходит для обработки текстового ввода или любой другой задачи, зависящей от порядка событий. Метод также полезен для определения первого касания пальцем экрана или поднятия его.

Какие устройства ввода нам нужно контролировать? В случае с Android существует три основных метода ввода: сенсорный экран, клавиатура/трекбол и акселерометр. Для первых двух способов могут использоваться оба метода обработки событий. Для акселерометра обычно применяется только опрашивание. Сенсорный экран способен генерировать три события:

- *касание экрана* — происходит, когда палец касается дисплея;
- *перетаскивание* — выполняется, когда палец движется по дисплею. Возникновению этого события всегда предшествует событие касания;
- *отрыв* — происходит, когда палец поднимается от дисплея.

Каждое событие касание несет дополнительную информацию: положение относительно компонентов интерфейса и индекс указателя (используется в мультитач-экранах для отслеживания касания нескольких пальцев).

Клавиатура генерирует два типа событий:

- *клавиша нажата* — происходит при нажатии кнопки;
- *клавиша отпущена* — выполняется при поднятии пальца от клавиши.

Клавиатурные события также содержат дополнительные данные. События типа «клавиша нажата» хранят код нажатой кнопки, а события типа «клавиша отпущена» включают в себя код кнопки и Юникод-символ. Между ними существует разница: во втором случае учитывается также состояние других клавиш (например, клавиши Shift). Благодаря этому у нас появляется возможность определять, ввел пользователь большую или маленькую букву. С событием нажатия клавиши нам повезло меньше — у нас нет точных данных о том, какой именно символ создается.

Наконец, есть еще акселерометр. Его состояние мы всегда получаем методом опрашивания. Акселерометр сообщает об изменении положения устройства по одному из направлений-осей, называемых обычно *x*, *y* и *z* (рис. 3.19).

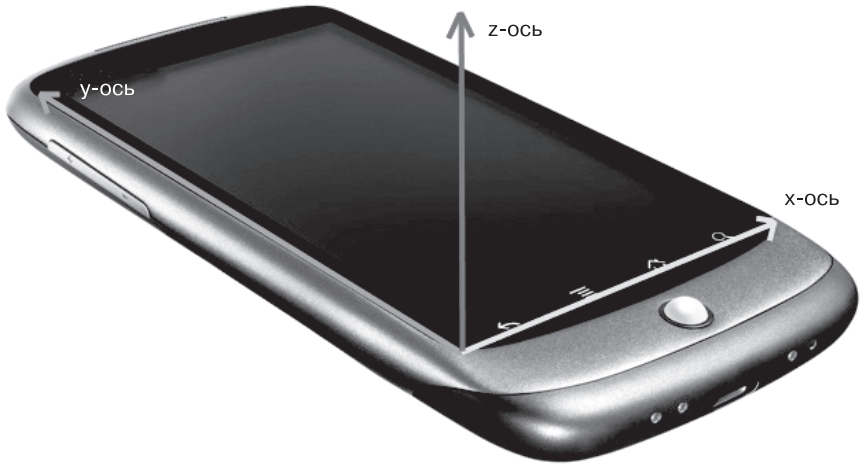

Рис. 3.19. Оси акселерометра на телефоне Android; ось *z* указывает вверх над телефоном

Ускорение по каждой из осей измеряется в метрах в секунду за секунду (м/с^2). Из уроков физики вы можете помнить, что при свободном падении любой объект движется с ускорением 9,8 м/с^2. На других планетах значение ускорения свободного падения отличается. Для простоты будем считать, что наше приложение будет работать только на планете Земля. Когда точка на оси отдаляется от центра Земли, значение ускорения будет возрастать. При обратном перемещении мы получаем отрицательную динамику ускорения. Например, если вы держите телефон вертикально в портретном режиме, значение на оси *y* будет равно 9,8 м/с^2. На рис. 3.19 такое значение будет по оси *z*, а оси *x* и *y* будут сообщать ускорение, равное нулю.

Теперь определим интерфейсы, позволяющие опрашивать события от сенсорного экрана, акселерометра и клавиатуры, а также дающие доступ к обработчикам событий от дисплея и клавиатуры (листинг 3.1).

Листинг 3.1. Интерфейс Input и классы KeyEvent и TouchEvent

```
package com.badlogic.androidgames.framework;

import java.util.List;

public interface Input {
    public static class KeyEvent {
        public static final int KEY_DOWN = 0;
        public static final int KEY_UP = 1;

        public int type;
        public int keyCode;
        public char keyChar;
    }

    public static class TouchEvent {
```

```
        public static final int TOUCH_DOWN = 0;
        public static final int TOUCH_UP = 1;
        public static final int TOUCH_DRAGGED = 2;

        public int type;
        public int x, y;
        public int pointer;
    }

    public boolean isKeyPressed(int keyCode);

    public boolean isTouchDown(int pointer);

    public int getTouchX(int pointer);

    public int getTouchY(int pointer);

    public float getAccelX();

    public float getAccelY();

    public float getAccelZ();

    public List<KeyEvent> getKeyEvents();

    public List<TouchEvent> getTouchEvents();
}
```

Наше определение начинается с двух классов — `KeyEvent` и `TouchEvent`. Класс `KeyEvent` определяет константы, кодирующие тип `KeyEvent`; класс `TouchEvent` делает то же самое. Экземпляр `KeyEvent` хранит его тип, код клавиши и Юникод-код (если тип события `KEY_UP`).

Код `TouchEvent` выполняет аналогичную функцию — хранит тип `TouchEvent`, позицию пальца относительно исходного элемента интерфейса, и ID указателя, выданный данному пальцу драйвером сенсорного экрана. Этот ID будет храниться до тех пор, пока палец будет касаться дисплея. При этом первый коснувшийся экрана палец получает ID, равный 0, следующий — 1 и т. д. Если экрана касаются два пальца и палец 0 поднят, ID второго остается равным 1 до тех пор, пока он касается экрана. Следующий палец получает первый свободный номер, который в данном случае может быть равен 0.

Следующие строки кода — методы опрашивания интерфейса `Input`, которые достаточно прозрачны и не требуют подробных объяснений. `Input.isKeyPressed()` получает `keyCode` и возвращает результат — нажата соответствующая кнопка в данный момент или нет. `Input.isTouchDown()`, `Input.getTouchX()` и `Input.getTouchY()` возвращают состояние переданного им указателя, его x- и y-координаты. Обратите внимание — значение этих координат будет не определено, если соответствующий указатель в данный момент не касается экрана.

`Input.getAccelX()`, `Input.getAccelY()` и `Input.getAccelZ()` возвращают соответствующие значения ускорения для каждой оси.

Последние два метода используются для реализации обработчиков событий. Они возвращают экземпляры `KeyEvent` и `TouchEvent`, хранящие информацию с последнего раза, когда мы вызывали эти методы. События расположены в порядке их появления — самое свежее из них размещено в конце списка.

С этим простым интерфейсом и вспомогательными классами у нас есть все, чтобы удовлетворить наши потребности во вводе. Теперь займемся файловыми операциями.

ПРИМЕЧАНИЕ

Хотя изменяющиеся классы с открытыми методами вызывают отвращение, в данном случае мы можем их оставить по двум причинам: во-первых, Dalvik все еще слишком неторопливо при вызове методов (в данном случае свойств); во-вторых, изменяемость классов событий не влияет на внутреннюю работу нашей реализации ввода. Просто запомните, что это — плохой стиль программирования, и больше не будем об этом вспоминать.

Файловый ввод-вывод

Чтение и запись файлов — весьма необходимые вещи для наших попыток в программировании игр. Учитывая, что мы находимся в стране Java, по большей части нам придется иметь дело с экземплярами классов `InputStream` и `OutputStream` — стандартными Java-механизмами чтения и записи данных в файлы. В нашем случае нас больше интересует чтение файлов из пакета нашей игры (уровней, изображений, звукозаписей). С записью файлов мы будем сталкиваться гораздо реже — она понадобится нам, только если мы захотим сохранить результаты, настройки или игру, чтобы потом продолжить с того места, где прервались. В общем, нам необходим самый простой механизм доступа к файловой системе — такой, как в листинге 3.2.

Листинг 3.2. Интерфейс FileIO

```
package com.badlogic.androidgames.framework;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

 public interface FileIO {
    public InputStream readAsset(String fileName) throws IOException;

     public InputStream readFile(String fileName) throws IOException;

     public OutputStream writeFile(String fileName) throws IOException;
}
```

Тут все просто и понятно. Мы просто определяем имя файла и возвращаем для него поток. Как обычно, в Java в случае непредвиденных событий мы вызываем

исключение IOException. Где именно мы будем читать и записывать файлы, зависит, конечно, от реализации интерфейса. Ресурсы могут быть прочитаны непосредственно из APK-файла приложения или с SD-карты (также называемой внешним хранилищем).

Возвращаемые экземпляры InputStreams и OutputStreams — старые добрые потоки Java. Естественно, после окончания использования их необходимо закрывать.

Звук

Хотя программирование звука — довольно сложная тема, мы можем использовать для него весьма простую абстракцию. Мы не будем реализовывать сложную обработку звука; достаточно будет обеспечить воспроизведение звуковых эффектов и музыки, загруженных из файлов (примерно так же, как мы будем загружать растровые изображения в графическом модуле).

Однако перед тем, как погрузиться в интерфейсы звукового модуля, сделаем паузу и получим некоторое представление о том, что такое звук и как его представить в цифровом виде.

Физика звука

Звук обычно описывают как поток волн, перемещающихся в пространстве подобно воздуху или воде. Волна — это не физически осязаемый объект, а скорее движение молекул в пространстве. Представьте себе пруд, в который бросили камень. Когда камень достигает поверхности воды, он заставляет двигаться множество молекул, которые, в свою очередь, передают свое движение соседям. В результате вы увидите круги, расходящиеся от того места, куда упал камень. Из подобных высоконаучных экспериментов, проводимых вами в детстве, вы знаете, что волны воды могут взаимодействовать друг с другом — складываться или гасить друг друга. Все это верно и для звуковых волн. Они комбинируются для получения разных тонов и мелодий, которые вы слышите как музыку. Громкость звука определяется количеством энергии, передаваемой молекулами своим соседям, пока они не достигнут вашего уха.

Запись и воспроизведение

Принцип записи и воспроизведения звука в теории довольно прост: для записи мы отслеживаем количество энергии, приложенной к определенной точке молекулами, формирующими звуковые волны. Воспроизведение — по сути, принуждение молекул воздуха, окружающих динамик, двигаться так же, как они делали это при записи.

Конечно, на практике все сложнее. Звук обычно записывается одним из двух методов: аналоговым или цифровым. В обоих случаях звуковые волны записываются с помощью микрофона, главным элементом которого является мембрана, преобразующая удары молекул в какой-либо вид сигнала. Способ обработки и хранения сигнала и составляет разницу между аналоговой и цифровой записью. Мы будем иметь дело с цифровыми записями, поэтому рассмотрим второй вариант.

При цифровой записи звука состояние мембраны микрофона измеряется за дискретный временной такт и сохраняется. В зависимости от состояния окружающих молекул мембрана может выгибаться внутрь или наружу, возвращаясь затем в нейтральное состояние. Процесс измерения и сохранения состояния микрофона называется сэмплированием, поскольку мы сохраняем сэмплы состояния мембраны за дискретные такты времени. Количество полученных сэмплов за единицу времени называется частотой дискретизации. Обычно временной интервал задается в секундах, а частота измеряется в герцах (Гц). Чем больше сэмплов записывается в секунду, тем выше качество записи. Компакт-диски воспроизводят звук с частотой дискретизации 44 100 Гц (44,1 КГц). Более низкие частоты дискретизации можно обнаружить при передаче голоса по телефону (чаще всего 8 КГц).

Частота дискретизации — отнюдь не единственный параметр, определяющий качество звука. Способ хранения положений мембраны также имеет значение и тоже является субъектом оцифровки. Напомню, что такое положение мембраны — это размер ее отклонения от нейтрального положения. Поскольку направление отклонения имеет значение, его значение сохраняется со знаком. Следовательно, положение мембраны во время определенного временного интервала — положительное или отрицательное число. Мы можем хранить это число разными способами: как целое 8-, 16- или 32-битное число со знаком или как 32-битное (и даже 64-битное) число с плавающей точкой. Каждый из этих типов данных имеет свою точность. Например, 8-битное целое число может хранить значения от –128 до 127. 32-битный тип `integer` предлагает намного больший диапазон. При хранении в виде `float` положение мембраны обычно нормализируется, чтобы попадать в диапазон от –1 до 1. При этом максимальное и минимальное значения соответствуют наибольшему отклонению мембраны от нейтрального положения в обе стороны. Положение мембраны также называется амплитудой. Оно характеризует громкость записываемого звука.

С одним микрофоном мы можем записать только одноканальный звук (моно), в котором отсутствует ощущение пространства. Имея два микрофона, мы можем измерять звуковые волны в разных точках пространства и получать таким образом так называемый стереозвук. Достичь этого можно, например, размещая микрофоны слева и справа от источника звука. Когда он потом воспроизводится одновременно из двух динамиков, мы можем услышать пространственный компонент аудиозаписи. Однако это также означает, что нам необходимо сохранять при записи вдвое больше звуковых сэмплов.

Воспроизведение — в общем-то, несложный процесс. Имея звуковые сэмплы в цифровом виде, сохраненные с конкретной частотой дискретизации в виде данных определенного типа, мы можем вывести их на наше воспроизводящее звук устройство, которое трансформирует информацию в сигналы для подключенного к нему динамика. Динамик дешифрует сигнал и преобразует его в колебания своей мембраны, которая, в свою очередь, заставляет двигаться молекулы воздуха вокруг нее и таким образом генерировать звуковые волны. Все то же самое, что и при записи, только в обратном порядке!

Качество звука и компрессия

Да уж, сплошная теория. Зачем нам вообще об этом знать? Если вы читали внимательно, то понимаете, что качество звукового файла зависит от частоты дискретизации и типа данных, использованных для его сохранения. Чем выше частота дискретизации и больше точность типа данных, тем лучше будет результат на выходе. Однако не стоит забывать об увеличивающихся требованиях к вместимости хранилища.

Представьте, что мы записали один и тот же аудиоролик длиной 60 секунд дважды: в первый раз с частотой дискретизации 8 КГц и 8 битами на сэмпл и во второй раз с частотой 44 КГц и 16 битами на сэмпл. Как вы думаете, сколько места потребуется нам для хранения в обоих случаях? Для первого ролика мы используем 1 байт на сэмпл. Умножим это значение на частоту 8000 Гц и получим 8000 байт в секунду. Для 60-секундной звукозаписи нам потребуется 480 000 байт, или примерно 0,5 Мбайт. Вторая, высококачественная запись потребует больше места: 2 байта на сэмпл, 2 раза по 44 000 байт в секунду. Итого 88 000 байт в секунду. Умножаем на 60 секунд и получаем в итоге 5 280 000 байт (чуть более 5 Мбайт). Стандартная трехминутная поп-песенка в таком качестве займет примерно 15 Мбайт (и это мы говорим о режиме моно, для стереокачества места потребуется в два раза больше). Не многовато ли места для глупой песенки?

Умные люди позаботились об этом и разработали различные способы уменьшить количество байт, необходимых для записи звука. Они изобрели довольно сложные психоакустические алгоритмы компрессии, анализирующие исходные записи и дающие на выходе их сжатые версии. Этому процессу обычно сопутствуют потери — некоторые несущественные части оригинальной записи удаляются. При использовании таких форматов, как MP3 или OGG, вы на самом деле слушаете сжатые аудиозаписи с потерями, однако их применение помогает уменьшить количество требуемого дискового пространства.

Как же эти сжатые аудиозаписи воспроизводятся? Хотя существуют специальные аппаратные чипы для декодирования различных аудиоформатов, в большинстве случаев перед тем, как «скормить» звуковой карте запись, необходимо ее сначала прочитать и декомпрессировать. Мы можем сделать это однажды и хранить все несжатые звуки в памяти или же извлекать фрагменты аудиофайла по мере необходимости.

На практике

Вы убедились, что даже трехминутные песенки могут занять много места. Когда мы воспроизводим музыку из нашей игры, то извлекаем звуковые фрагменты на лету, вместо того чтобы загружать их заранее в память. Обычно у нас воспроизводится только один звуковой поток за раз, поэтому нам необходим только однократный доступ к диску. Для коротких звуковых эффектов (например, взрывов или выстрелов) картина несколько иная — один и тот же звуковой эффект мы будем применять неоднократно и одновременно. Потоковое воспроизведение звуковых фрагментов с диска для каждого экземпляра эффекта — не лучшее решение.

Однако нам везет — короткие звуки обычно занимают мало места в памяти. Поэтому мы можем загрузить их в память, откуда будем воспроизводить по мере надобности.

Итак, у нас следующие требования. Нам необходимы способы:

- загрузки аудиофайлов для потокового воспроизведения из памяти;
- управления воспроизведением потокового звука;
- управления воспроизведением полностью загруженного в память звука.

Все это напрямую транслируется в интерфейсы `Audio`, `Music` и `Sound` (показанные в листингах 3.3–3.5 соответственно).

Листинг 3.3. Интерфейс `Audio`

```
package com.badlogic.androidgames.framework;

public interface Audio {
    public Music newMusic(String filename);

    public Sound newSound(String filename);
}
```

Интерфейс `Audio` — наш способ создавать новые интерфейсы `Music` и `Sound`. Экземпляр `Music` представляет потоковый аудиофайл, интерфейс `Sound` — короткий звуковой эффект, который мы можем хранить в памяти. Методы `Audio.newMusic()` и `Audio.newSound()` принимают имя файла в качестве аргумента и вызывают исключение `IOException` при сбое процесса загрузки (например, если нужный файл не существует или поврежден). Имена файлов соотносятся с файлами активов, хранящимися в пакете APK нашего приложения.

Листинг 3.4. Интерфейс `Music`

```
package com.badlogic.androidgames.framework;

 public interface Music {
    public void play();

    public void stop();

    public void pause();

    public void setLooping(boolean looping);

    public void setVolume(float volume);

    public boolean isPlaying();

    public boolean isStopped();

    public boolean isLooping();

    public void dispose();
}
```

Интерфейс Music чуть более сложен. Он включает в себя методы для начала воспроизведения музыкального потока, приостановки и прекращения воспроизведения, а также циклического воспроизведения (начало проигрывания звука с начала сразу после окончания). Кроме того, мы можем установить громкость в виде типа данных float в диапазоне от 0 (тишина) до 1 (максимум). Интерфейс также содержит несколько методов получения, позволяющих отследить текущее состояние экземпляра Music. После того как необходимость в объекте Music отпадет, его необходимо утилизировать — это освободит системные ресурсы, а также снимет блокировку с воспроизводимого звукового файла.

Листинг 3.5. Интерфейс Sound

```
package com.badlogic.androidgames.framework;

public interface Sound {
    public void play(float volume);

    public void dispose();
}
```

Интерфейс Sound, напротив, очень прост. Все, что нам нужно, — вызов метода play(), принимающего в качестве параметра громкость в виде числа float. Мы можем вызывать метод play() всякий раз, когда захотим (например, при каждом выстреле или прыжке персонажа). Когда необходимость в экземпляре Sound отпадет, его следует уничтожить по тем же причинам — для освобождения памяти и потенциально связанных системных ресурсов.

ПРИМЕЧАНИЕ

Хотя в этой главе мы довольно глубоко погрузились в мир звука, о программировании аудио нужно узнать еще очень много. Чтобы раздел не разросся, мне пришлось упростить некоторые вещи. Например, обычно вы не будете изменять громкость звука линейно. Однако в нашем случае можно опустить подобные детали. Просто знайте, что в этом вопросе еще многое осталось неизученным.

Графика

Последний большой модуль, который нам необходимо изучить, — графика. Как вы, вероятно, догадались, он отвечает за прорисовку изображений (также известных как растры) на нашем экране. Это может звучать просто, но если вы хотите использовать высокопроизводительную графику, то должны познакомиться для начала с основами работы с графикой. Начнем со старого доброго 2D.

Первый вопрос, который нам нужно задать себе, — каким образом вообще изображения появляются на экране? Ответ на него довольно сложен, и нам нет нужды вникать во все детали — просто бегло рассмотрим то, что происходит в нашем компьютере и на экране.

О растрах, пикселах и фреймбуферах

Сейчас все дисплеи основаны на растре — двухмерной таблице так называемых элементов изображения. Вы можете знать их под именем «пикселы», и мы будем их так называть далее. Таблица растров ограничена по ширине и высоте, которые

выражены количеством пикселов в строке и столбце. Если вы любознательны, то можете включить ваш компьютер и попытаться увидеть на экране отдельные пикселы. Сразу замечу, что не несу ответственности за те повреждения, которые могут получить ваши глаза.

У пиксела есть два атрибута: позиция в таблице и цвет. Позиция пиксела выражается двухмерными координатами в дискретной координатной системе. Дискретность в данном случае означает, что каждая координата выражена целым числом. Вообще, пикселы определяются в виде Эвклидовой координатной системы, наложенной на таблицу и начинающейся из левого верхнего угла. Значения координаты x растут слева направо, координаты y — сверху вниз. Последнее обстоятельство часто сбивает с толку. Но на это есть простая причина, и скоро мы о ней поговорим.

Игнорируя ось y, мы увидим, что из-за дискретной природы координат их начало совпадает с левым верхним углом таблицы, расположенным по адресу (0; 0). Пиксел справа от него имеет координаты (1; 0), пиксел под ним — (0; 1) и т. д. (посмотрите на левую часть рис. 3.20). Растровая таблица дисплея небезгранична, поэтому количество координат ограниченно. Координаты с отрицательными значениями находятся за пределами экрана (как и координаты, равные и превышающие границы растра). Обратите внимание — максимальное значение координаты x равно ширине таблицы минус 1, а максимальной координаты y — высота минус 1. Это происходит из-за того, что координаты начинаются с нуля, а не единицы (распространенная причина недоразумений при программировании графики).

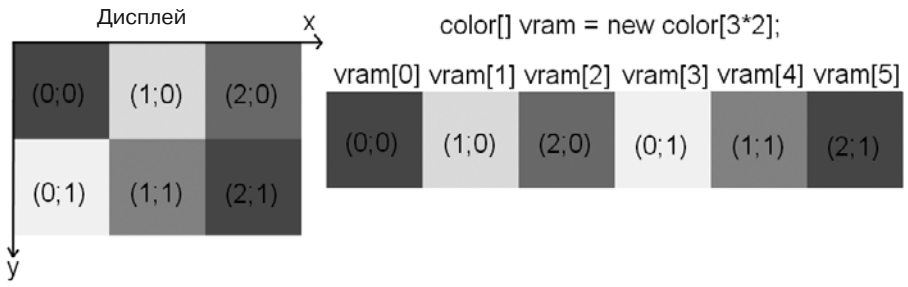

Рис. 3.20. Упрощенная схема таблицы растров и VRAM

Дисплей получает постоянный поток информации от графического процесса. Он кодирует цвет каждого пиксела экранной таблицы, как это определяется программой или операционной системой при управлении прорисовкой дисплея. Экран обновляет свое состояние несколько десятков раз в секунду (это значение называется частотой обновления, выражаемой в герцах). Частота обновления ЖК-дисплеев составляет обычно 60 Гц в секунду; ЭЛТ-мониторы и плазменные панели поддерживают большую частоту.

Графический процессор имеет доступ к специальной области памяти, также известной как видеопамять (или VRAM). Внутри VRAM, в свою очередь, зарезервирована область для хранения каждого пиксела, отображаемого на экране.

Эта область обычно называется фреймбуфером, а полное изображение экрана — фреймом. Каждому пикселу таблицы растров соответствует адрес в памяти во фреймбуфере, хранящий его цвет. Когда мы хотим что-то изменить на экране, то меняем цветовые значения в этой области VRAM.

Пришло время объяснить, почему ось y в системе координат дисплея направлена вниз. Память (будь это VRAM или обычная RAM) линейна и однонаправлена. Представляйте ее в виде одномерного массива. Как же нам разместить двухмерные координаты пиксела в одномерной ячейке памяти? На рис. 3.20 показана весьма маленькая таблица 3×2 пиксела, а также ее представление в VRAM (представим, что VRAM состоит только из фреймбуфера). Исходя из него, мы можем вывести формулу вычисления адреса памяти для пиксела с координатами x и y:

```
int address = x + y * rasterWidth;
```

Мы можем также пойти другим путем, рассчитав отдельно координаты x и y пиксела:

```
int x = address % rasterWidth;
int y = address / rasterWidth;
```

Итак, ось y направлена вниз из-за компоновки памяти цветов пикселов в VRAM. Это что-то вроде наследства, доставшегося нам от эпохи начала компьютерной графики. Мониторы тогда обновляли цвет каждого пиксела экрана, начиная с левого верхнего угла, двигаясь вправо до упора, возвращаясь влево на следующей строке и так до нижнего предела экрана. Было удобно иметь такую структуру VRAM, упрощающую передачу информации о цвете на монитор.

ПРИМЕЧАНИЕ

Если бы у нас был полный доступ к фреймбуферу, мы могли бы использовать опережающие расчеты для написания полнофункциональной библиотеки рисования пикселов, линий, прямоугольников, изображений в памяти и т. д. Современные операционные системы по различным причинам не предоставляют полный доступ к фреймбуферу. Вместо этого мы обычно рисуем в области памяти, которая затем копируется в настоящий фреймбуфер операционной системой. Тем не менее общие принципы при этом остаются неизменными. Если вам интересно, как делаются подобные низкоуровневые вещи, поищите в информацию о Брезенхэме и его алгоритмах рисования линий и окружностей.

Vsync и двойная буферизация

Итак, значения частот обновления не слишком велики, и мы можем записывать данные во фреймбуфер быстрее, чем обновляется экран. Такое вполне может произойти. Более того — мы не знаем точно, когда дисплей получает последнюю копию фрейма из VRAM, что может вызвать проблемы, если мы находимся в процессе рисования чего-нибудь. В этом случае дисплей покажет нам части старого содержимого фреймбуфера вперемешку с новыми, что крайне нежелательно. Вы можете наблюдать данный эффект во многих играх для PC, где его можно описать как размытость (на экране одновременно видны части старого и нового кадра).

Первая часть решения этой проблемы — так называемая двойная буферизация. Вместо того чтобы использовать единственный фреймбуфер, графический

процессор (GPU) управляет двумя — основным и фоновым. Из основного буфера дисплей получает информацию о цветах пикселов. Фоновый буфер нужен для прорисовки следующего кадра, пока экран переваривает содержимое основного буфера. После прорисовки текущего кадра мы приказываем GPU поменять эти буферы местами, что обычно означает команду поменять местами их адреса в памяти. В литературе по программированию графики и документации по API вы можете обнаружить такие термины, как «переворот страницы» и «обмен буферов», как раз и обозначающие описанный процесс.

Однако двойная буферизация сама по себе не решает проблему полностью: обмен тоже может произойти в тот момент, когда экран находится в процессе обновления своего содержимого. Тут нам приходит на помощь вертикальная синхронизация (также известная как vsync). При вызове метода обмена между буферами графический процессор блокируется до тех пор, пока дисплей не сообщает об окончании обновления. После получения этого сигнала GPU может безопасно менять адреса памяти буферов, и все будет хорошо.

К счастью, сейчас нам редко приходится заботиться об этих нудных подробностях. VRAM и тонкости двойной буферизации и вертикальной синхронизации безопасно скрыты от нас, поэтому мы не можем натворить с ними дел. Вместо этого нам предлагается набор API, обычно ограничивающий нас в манипуляциях содержимым нашего окна приложения. Некоторые из этих API (например, OpenGL ES) используют аппаратное ускорение, а это обычно подразумевает лишь манипулирование VRAM с помощью специальных контуров графического чипа. Так что, как видите, никакого волшебства. Причина, по которой вы должны понимать принципы работы графики (по крайней мере на высшем уровне), состоит в том, что это позволяет вам понимать законы, управляющие производительностью вашего приложения. При включенном vsync вы никогда не сможете превысить частоту обновления вашего дисплея, что может сбить с толку, если вы хотите нарисовать один-единственный пиксел.

При использовании API без применения аппаратного ускорения мы напрямую не взаимодействуем с дисплеем, вместо этого рисуя один из наших UI-компонентов в окне. В нашем случае мы имеем дело с единственным компонентом, растянутым на все окно приложения. По этой причине наша координатная система растягивается не на весь экран, а только на наш компонент. Таким образом, наш компонент пользовательского интерфейса становится заменой дисплею со своим собственным фреймбуфером. Операционная система управляет композицией содержимого всех видимых окон и обеспечивает корректность их передачи в части, соответствующие реальному фреймбуферу.

Что такое цвет?

Вы, вероятно, заметили, что до сей поры я игнорировал разговоры о цветах. Я создал тип `color` на рис. 3.20 и притворился, что все в порядке. Теперь пришло время узнать, что такое цвет на самом деле.

С точки зрения физики цвет — реакция сетчатки глаза и коры головного мозга на электромагнитные волны. Такая волна характеризуется длиной и интенсивностью. В пределах нашей видимости находятся волны длиной от 400 до 700 нм. Этот сегмент электромагнитного спектра, также известный как видимая часть спектра, заключается между фиолетовым и красным цветами (между ними находятся синий, желтый и оранжевый). Передача цветом монитора заключается в трансляции электромагнитных волн определенной частоты для каждого пиксела. Разные виды дисплеев используют различные способы достижения этой цели. Упрощенно этот процесс можно представить так: пиксел на экране состоит из трех разных светящихся частей, каждая из которых излучает один из цветов (красный, синий и зеленый). При обновлении дисплея эти части светятся по разным причинам (например, в ЭЛТ-мониторах в них попадают электроны). При этом дисплей контролирует, какие части из набора светятся. Если пиксел полностью красный, только красная его часть будет бомбардироваться электронами с полной интенсивностью. В случае, когда нам нужны цвета, отличные от базовых, это достигается смешиванием, которое, в свою очередь, реализуется управлением интенсивностью свечения частей пиксела. Электромагнитные волны по пути к сетчатке нашего глаза накладываются друг на друга, и наш мозг интерпретирует эту смесь как определенный цвет. Поэтому мы можем определить любой цвет как смешение трех базовых цветов разной интенсивности.

Цветовые модели

То, что мы обсуждали только что, называется цветовой моделью (точнее, цветовой моделью RGB). RGB означает Red (Красный), Green (Зеленый), Blue (Синий). Существует множество и других цветовых моделей (например, YUV и CMYK). Однако в большинстве API для программирования модель RGB является, по сути, стандартом, поэтому и мы будем обсуждать только ее.

Цветовая модель RGB называется аддитивной из-за того, что финальный цвет получается смешением и дополнением базовых цветов (красного, синего, зеленого). Вы, скорее всего, экспериментировали со смешением цветов еще в школе. Рисунок 3.21 демонстрирует несколько примеров смешения цветов в модели RGB.

Конечно, мы можем создавать гораздо больше цветов, чем показано на рис. 3.21, варьируя интенсивность красного, синего и зеленого цветов. Каждый компонент обладает интенсивностью от 0 до максимального значения (допустим, 1). Если мы интерпретируем каждый компонент цвета как значение одной из трех осей Эвклидовой оси координат, то сможем построить так называемый цветовой куб (рис. 3.22). При изменении интенсивности каждого компонента можно получить еще больше цветов. Цвет рассматривается как триплет (красный, зеленый, синий), где значение каждого элемента лежит в диапазоне от 0 до 1. Например, 0,0 означает отсутствие интенсивности для данного цвета, а 1,0 — полную интенсивность. Черный цвет имеет значение (0; 0; 0), белый — (1; 1; 1).

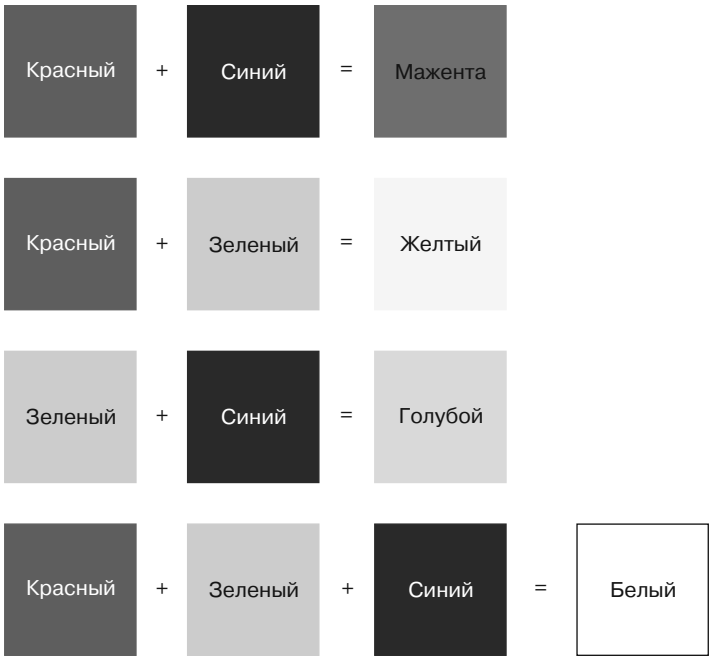

Рис. 3.21. Смешивание базовых цветов

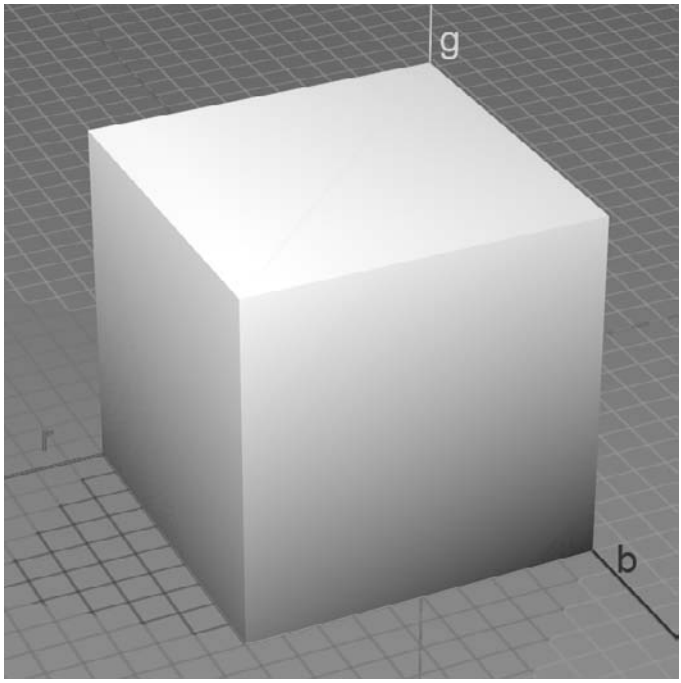

Рис. 3.22. Цветовой куб

Цифровое кодирование цветов

Как мы можем закодировать триплет цветов RGB в компьютерной памяти? Для начала необходимо определиться с типом данных, который мы будем использовать для цветовых компонентов. Можно воспользоваться числами с плавающей точкой и определить диапазон между 0,0 и 1,0. Это дало бы нам большое разнообразие вариантов цветов. К сожалению, данный способ требует много места (3 раза по 4 или 8 байт на пиксел в зависимости от того, используем ли мы 32- или 64-битные числа). Поэтому лучшим решением будет отказаться от всего возможного многообразия цветов, тем более что дисплеи обычно способны передавать ограниченное их количество. Вместо чисел с плавающей точкой мы можем применить беззнаковые целые числа. Интенсивность каждого компонента варьируется от 0 до 255. В этом случае для одного пиксела нам потребуется лишь 3 байта, или 24 бита, а значит, мы сможем закодировать таким образом 2 в 24-й степени (16 777 216) цветов. Я бы сказал, что этого вполне достаточно.

Можем ли мы сэкономить еще? Да, можем. Мы можем запаковать каждый компонент в отдельное 16-битное слово, и в этом случае каждому пикселу потребуется 2 байта для хранения. Красному цвету будет необходимо 5 бит, зеленому — 6, синий будет использовать оставшиеся 5 бит. Причина, по которой зеленому цвету нужно больше бит, в том, что наши глаза распознают больше оттенков зеленого, чем синего или красного. Все биты вместе дают 2 в 16-й степени (65 536) вариантов возможных цветов. На рис. 3.23 показаны три описанных выше способа кодирования цвета.

Рис. 3.23. Цветовое кодирование оттенка розового (на черно-белом рисунке он, увы, выглядит серым)

В случае с типом `float` мы можем использовать 32-битный тип данных Java. При 24-битном кодировании у нас возникает небольшая проблема: в Java нет 24-битного типа `integer`, поэтому нам либо придется хранить каждый компонент в типе `byte`, либо задействовать 32-битный тип `integer`, оставляя 8 его бит неиспользованными. При 16-битном кодировании мы можем либо применять два отдельных байта, либо хранить компоненты в данных типа `short`. Заметьте также, что Java не имеет беззнаковых типов данных, однако мы можем спокойно использовать знаковые типы для хранения беззнаковых значений.

Как при 16-битном, так и при 24-битном кодировании необходимо также определить порядок, в котором будут храниться три цветовых компонента в типе данных `short` или `integer`. Обычно используются два варианта: RGB и BGR. На рис. 3.23 используется RGB-кодирование. Синий компонент хранится в нижних 5 или 8 битах, зеленый компонент — в следующих 6 или 8 битах, красный — в верхних 5 или 8 битах. При BGR-кодировании порядок обратный — зеленый остается на месте,

красный и синий меняются местами. В книге мы будем пользоваться RGB-порядком, поскольку в графическом API Android применяют именно его.

Итак, подведем итоги нашей дискуссии о кодировании цветов.

- 32-битное RGB-кодирование `float` использует 12 байт на пиксел, интенсивность при этом меняется в диапазоне от 0,0 до 1,0.
- 24-битное RGB-кодирование `integer` применяет 3 или 4 байта на пиксел, интенсивность при этом варьируется от 0 до 255. Порядок компонентов может быть RGB или BGR. Иногда используется маркировка RGB888 или BGR888, где цифра 8 означает количество битов на компонент.
- 16-битное RGB-кодирование `integer` применяет 2 байта на пиксел; интенсивность красного и синего цветов меняется от 0 до 31, зеленого — от 0 до 63. Порядок компонентов может быть RGB или BGR. Иногда используется маркировка RGB565 или BGR565, где цифры 5 и 6 означают количество битов на соответствующий компонент.

Применяемый тип кодирования также называется глубиной цвета. Изображения, которые мы создаем и храним на диске или в памяти, обладают определенной глубиной цвета, то же относится и к фреймбуферу и самому дисплею. Современные экраны обычно обладают глубиной цвета по умолчанию в 24 бита, но в некоторых случаях могут быть сконфигурированы на меньшее значение. Фреймбуфер графического оборудования также достаточно гибок и может использовать различные глубины цвета. Созданные нами изображения, естественно, могут обладать совершенно разной глубиной.

ПРИМЕЧАНИЕ

Существует много способов кодирования цветовой пиксельной информации. Помимо цветов RGB мы можем применять оттенки серого, состоящие из одного компонента. Но поскольку эти способы широко не используются, мы в данный момент их проигнорируем.

Форматы изображения и сжатие

На определенном этапе процесса разработки игры наш художник предложит нам изображения, созданные с помощью специального программного обеспечения (например, Gimp, Pain.NET или Photoshop). Эти изображения могут храниться на диске в различных форматах. Для чего такое разнообразие? Разве мы не можем хранить растровые картинки на диске как простой набор байтов?

В общем-то, да, можем. Однако посмотрим, сколько это займет места. Если нам необходимо лучшее качество, то пикселы будут кодироваться в RGB888 (24 бита на пиксел). Допустим, размер изображения составляет 1024×1024 пиксела. Итого — 3 Мбайт на одну небольшую картинку! При использовании метода RGB565 вы сэкономите, но немного — размер уменьшится на 1 Мбайт.

Как и в случае со звуком, над методами уменьшения объема необходимой для хранения изображения памяти была проведена большая работа. Были разработаны алгоритмы компрессии, предназначенные для хранения изображений и сохраняющие как можно больше информации о цвете. Два наиболее популярных формата — JPEG и PNG. JPEG — формат с потерей качества. Это означает, что некоторая

часть исходных данных в процессе сжатия теряется, однако он предлагает большую степень сжатия и таким образом экономит больше места на диске. PNG относится к числу форматов без потери данных, поэтому он воспроизводит изображение со стопроцентной точностью. Выбор формата зависит, таким образом, преимущественно от ограничений емкости хранилища.

Как и в случае со звуковыми эффектами, нам придется полностью декомпрессировать изображение при загрузке его в память. Так что даже если ваша картинка занимает 20 Кбайт на диске, вам все равно понадобится место для помещения ее полной таблицы растров в памяти.

После того как изображение загружено и разархивировано, оно становится доступно в виде массива пиксельных цветов (точно таким же образом, как фреймбуфер существует в оперативной памяти). Единственные отличия — пикселы хранятся в обычной RAM, а глубина цвета может отличаться от глубины цвета фреймбуфера. Загруженное изображение имеет такую же систему координат, что и фреймбуфер — с осью x, направленной слева направо, и осью y, направленной сверху вниз.

После загрузки изображения мы можем перерисовать ее из памяти во фреймбуфер, просто передав цвета пикселов из картинки в соответствующие положения фреймбуфера. Нам не придется делать это вручную: для реализации этого функционала существует соответствующее API.

Альфа-наложение и смешивание

Перед тем как начать разработку графических интерфейсов, нам придется разобраться с еще одним вопросом: наложением изображений. Предположим, что у нас есть фреймбуфер, который мы можем использовать для прорисовки, а также набор загруженных в память картинок, которые мы будем помещать в этот фреймбуфер. На рис. 3.24 показано простое фоновое изображение, а также Боб — зомби-убийца.

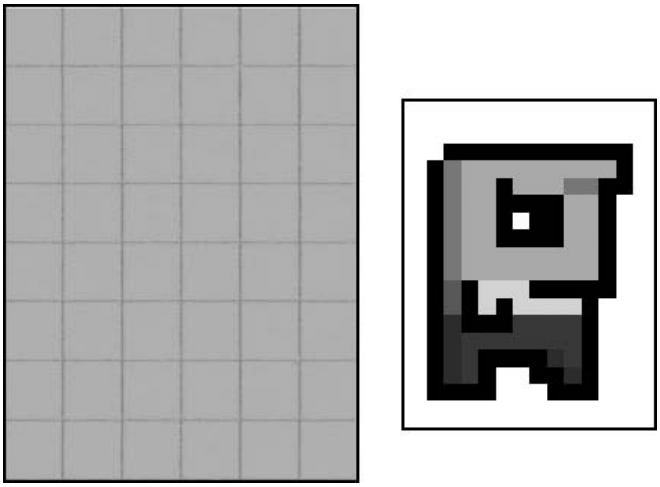

Рис. 3.24. Простой фон и Боб, хозяин Вселенной

Чтобы нарисовать мир Боба, для начала поместим фон во фреймбуфер, а потом туда же запустим поверх него нашего героя. Порядок действий важен, ведь вторая картинка перепишет текущее содержимое фреймбуфера. Итак, что мы получим в итоге? Ответ на рис. 3.25.

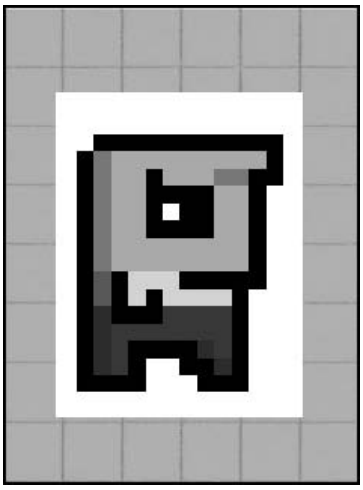

Рис. 3.25. Наложение Боба на фон во фреймбуфере

Это явно не то, что нам было нужно. Обратите внимание на рис. 3.24 — Боб окружен белыми пикселами. Когда мы помещаем его поверх фона во фреймбуфер, эти белые пикселы тоже там оказываются. Что нам надо сделать, чтобы Боб появился на фоне один, без сопутствующего ему белого фона?

Нам поможет альфа-смешивание. Вообще-то в случае с Бобом технически более верным будет говорить об альфа-маскировке (являющейся подвидом альфа-смешивания). Обычно программа по обработке изображений дает нам возможность определить не только RGB-значения пиксела, но и его полупрозрачность (будем рассматривать ее как еще один компонент цвета пиксела). Мы можем кодировать ее так же, как и красный, зеленый и синий компоненты.

Ранее я упоминал, что мы можем хранить 24-битный RGB-триплет в 32-битном типе данных `integer`. В этом случае у нас остается 8 неиспользованных бит, которые можно использовать для хранения значения альфы. Полупрозрачность пиксела может в данном случае варьироваться от 0 (полная прозрачность) до 255 (непрозрачность). Такой способ кодирования известен как ARGB8888 или BGRA8888 (в зависимости от порядка). Конечно, существуют также форматы RGBA8888 и ABGR8888.

При использовании 16-битного кодирования мы сталкиваемся с небольшой проблемой — все биты типа данных `short` заняты цветовыми компонентами. Определим некий формат ARGB4444 (по аналогии с ARGB8888), в котором 12 бит будут отведены под RGB-значения (по 4 бита на элемент).

Мы можем легко представить себе, как может работать метод прорисовки полностью прозрачных и полностью непрозрачных пикселов. В первом случае мы просто игнорируем пикселы с альфа-компонентой, равной 0, во втором — просто перерисовываем пиксел цветом. В случае промежуточного значения альфа-компоненты пиксела все несколько сложнее.

Рассматривая смешивание с формальной точки зрения, нам нужно определить несколько моментов:

- у смешивания есть два входа и один выход, каждый из которых представлен в виде RGB-триплета и альфа-значения (α);
- два входа называются источником и целью. Источник — это пиксел изображения, который мы хотим нарисовать поверх цели (например, во фреймбуфере). Цель — пиксел, который мы хотим частично перерисовать источником;
- выход — это тоже цвет, определяемый RGB-триплетом и альфа-значением. Обычно мы игнорируем альфу, для простоты в этой главе сделаем так же;
- чтобы слегка упростить нашу математику, представим значения RGB-компонента и альфа в диапазоне от 0,0 до 1,0.

Вооружившись этими определениями, мы можем создать так называемые уравнения смешивания. Простейшее из них выглядит примерно так:

```
red = src.red * src.alpha + dst.red * (1 – src.alpha)
blue = src.green * src.alpha + dst.green * (1 – src.alpha)
green = src.blue * src.alpha + dst.blue * (1 – src.alpha)
```

`src` и `dst` — пикселы источника и цели, которые мы хотим объединить. Два цвета смешиваются покомпонентно. Обратите внимание на отсутствие в этих уравнениях альфа-значения целевого пиксела. Попробуем это на примере:

```
src = (1, 0.5, 0.5), src.alpha = 0.5, dst = (0, 1, 0)
red = 1 * 0.5 + 0 * (1 – 0.5) = 0.5
blue = 0.5 * 0.5 + 1 * (1 – 0.5) = 0.75
red = 0.5 * 0.5 + 0 * (1 – 0.5) = 0.25
```

Рисунок 3.26 иллюстрирует предыдущее уравнение. Наш источник окрашен в оттенок розового, цель — в оттенок зеленого. Оба они участвуют в создании итогового цвета (некоего оттенка оливкового).

Рис. 3.26. Смешивание двух пикселов

Два достойных джентльмена с именами Портер и Дафф создали очень много уравнений смешивания. Однако нам хватит и описанного выше, поскольку его будет достаточно для большинства ситуаций. Поэкспериментируйте с ним на бумаге или в графической программе, чтобы получить более четкое представление о смешивании и наложении цветов.

ПРИМЕЧАНИЕ

Смешивание — широкое поле для деятельности. Если вы хотите использовать весь его потенциал, можете поискать в Сети работы Портера и Даффа на эту тему. Однако для игр, которые мы будем писать, достаточно будет самого простого уравнения.

Обратите внимание на большое количество операций умножения, участвующих в вышеупомянутых уравнениях (шесть, если быть точным). Умножение — ресурсоемкая операция, и ее необходимо по возможности избегать. При смешивании мы можем избавиться от трех операций умножения, предварительно умножив RGB-значения пиксела-источника на его же альфа-значение. Большинство программ по обработке графики поддерживают предварительное умножение значений RGB на соответствующее значение альфы. Если поддержка таких операций отсутствует, вы можете сделать это в процессе загрузки картинки в память. Однако при использовании графического API для прорисовки изображений со смешиванием, нам необходимо будет убедиться в использовании правильного уравнения смешивания. Наша картинка все еще будет содержать альфа-значения, поэтому предыдущее уравнение смешивания выдаст неправильные результаты. Альфа источника не должна умножаться на цвет источника. К счастью, все графические API для Android позволяют нам полностью определить, как мы хотим смешивать наши изображения.

В случае с Бобом мы с помощью графического редактора просто устанавливаем альфа-значения всех белых пикселов равными 0, загружаем изображение в формате ARGB8888 или ARGB4444 (возможно, предварительно перемножив) и используем метод прорисовки, производящий альфа-смешивание с правильным уравнением. Результат можно увидеть на рис. 3.27.

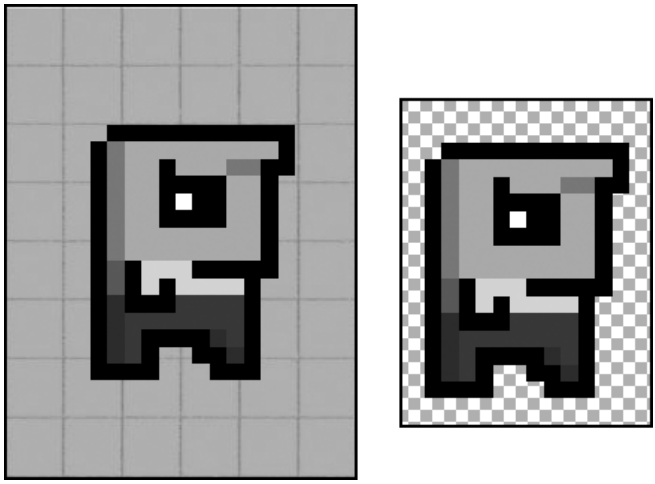

Рис. 3.27. Слева Боб уже смешан; справа Боб в редакторе Paint.NET; шахматная доска демонстрирует, что альфа-значение пикселов белого фона равно нулю

ПРИМЕЧАНИЕ

Формат JPEG не поддерживает хранение альфа-значений для пикселов. Используйте для таких случаев формат PNG.

На практике

Вооружившись всей этой информацией, мы можем, наконец, начать разрабатывать интерфейсы нашего графического модуля. Определим для начала их функциональность. Обратите внимание: когда я говорил о фреймбуфере, то на самом деле имел в виду виртуальный фреймбуфер компонента создаваемого нами пользовательского интерфейса. Мы просто делаем вид, что применяем реальный фреймбуфер. Нам необходимо, чтобы наш модуль выполнял следующие действия:

- загрузка изображений с диска и хранение их в памяти для дальнейшей прорисовки;
- очистка фреймбуфера с цветом для удаления содержимого его прошлого кадра;
- установка определенного цвета для определенного положения пиксела во фреймбуфере;
- прорисовка линий и прямоугольников во фреймбуфере;
- прорисовка ранее загруженных изображений во фреймбуфере. Было бы неплохо иметь возможность рисовать не только изображения целиком, но и его части. Вам также необходимо уметь рисовать картинки со смешиванием и без него;
- получение размеров фреймбуфера.

Я предлагаю два простых интерфейса: `Graphics` и `Bitmap`. Начнем с интерфейса `Graphics` (листинг 3.6).

Листинг 3.6. Интерфейс Graphics

```
package com.badlogic.androidgames.framework;

public interface Graphics {
    public static enum PixmapFormat {
        ARGB8888, ARGB4444, RGB565
    }

    public Pixmap newPixmap(String fileName, PixmapFormat format);

    public void clear(int color);

    public void drawPixel(int x, int y, int color);

    public void drawLine(int x, int y, int x2, int y2, int color);

    public void drawRect(int x, int y, int width, int height, int color);

    public void drawPixmap(Pixmap pixmap, int x, int y, int srcX, int srcY,
            int srcWidth, int srcHeight);

    public void drawPixmap(Pixmap pixmap, int x, int y);

    public int getWidth();

    public int getHeight();
}
```

Начинаем с открытого статичного перечисления PixmapFormat, необходимого для кодирования различных поддерживаемых нами форматов. Далее у нас несколько методов интерфейса.

- Graphics.newPixmap() — загружает изображение в JPEG- или PNG-формате. Мы определяем желаемый формат для результирующего Pixmap, который будет использоваться механизмом загрузки. Мы делаем так, чтобы иметь возможность каким-то образом управлять слепком памяти наших изображений (то есть загружая изображения форматов RGB888 или ARGB8888 в форматы RGB565 или ARGB4444). Имя файла определяет ресурс, содержащийся в APK-файле нашего приложения.
- Graphics.clear() — полностью очищает фреймбуфер с соответствующим цветом. Все цвета в нашем маленьком фреймворке определены в виде 32-битных значений формата ARGB8888 (конечно, Pixmaps может иметь и другой формат).
- Graphics.drawPixel() — устанавливает значение пиксела фреймбуфера с координатами (x; y) в определенный цвет. Координаты за пределами экрана будут игнорироваться (это называется клиппингом).
- Graphics.drawLine() — аналогичен Graphics.drawPixel(). Определяем начальную и конечную точку линии, а также ее цвет. Любая часть линии, выходящая за пределы фреймбуфера, будет игнорироваться.
- Graphics.drawRect() — рисует прямоугольник во фреймбуфере. Координаты (x; y) определяют позицию его левого верхнего угла. Аргументы width и height задают количество пикселов размера прямоугольника. Аргумент color указывает цвет его заполнения.
- Graphics.drawPixmap() — рисует прямоугольные участки Pixmap во фреймбуфере. Координаты (x; y) определяют позицию левого верхнего угла расположения цели Pixmap во фреймбуфере. Аргументы srcX и srcY (выраженные в координатной системе Pixmap) обозначают соответствующий левый верхний угол участка прямоугольника, используемого Pixmap. Параметры srcWidth и srcHeight означают размер участка, который мы извлекаем из Pixmap.
- Graphics.getWidth() и Graphics.getHeight() — возвращают ширину и высоту фреймбуфера в пикселах.

Все методы прорисовки, кроме Graphics.clear(), автоматически выполняют смешивание каждого пиксела, с которым они работают (как описано в предыдущем выше). Мы можем отключить смешивание для каждого отдельного пункта для некоторого ускорения прорисовки, однако это усложнит нашу реализацию. Обычно в простых играх вроде «Мистера Нома» можно оставить смешивание включенным.

Интерфейс Pixmap описан в листинге 3.7.

Листинг 3.7. Интерфейс Pixmap

```
package com.badlogic.androidgames.framework;

import com.badlogic.androidgames.framework.Graphics.PixmapFormat;

public interface Pixmap {
```

```
    public int getWidth();

    public int getHeight();

    public PixmapFormat getFormat();

    public void dispose();
}
```

Мы делаем его очень простым и неменяющимся, поскольку все наложение выполняется во фреймбуфере.

- `Pixmap.getWidth()` и `Pixmap.getHeight()` — возвращают ширину и высоту объекта `Pixmap` в пикселах.
- `Pixmap.getFormat()` — возвращает формат `PixelFormat`, используемый для хранения `Pixmap` в оперативной памяти.
- `Pixmap.dispose()` — экземпляры `Pixmap` применяют память и потенциально другие системные ресурсы. Если они нам больше не нужны, их следует уничтожить с помощью данного метода.

С этим простым графически модулем мы сможем в дальнейшем реализовать «Мистера Нома». Закончим эту главу обсуждением самого игрового фреймворка.

Игровая среда

После выполнения всей черновой работы мы можем поговорить о том, как, собственно, будем создавать игру.

- Игра разделяется на экраны, выполняющие одни и те же задачи: обработка пользовательского ввода, соответствующее изменение состояния экрана, прорисовка игровой сцены. Некоторые экраны могут не нуждаться в поддержке пользовательского ввода, однако реализуют переход на другой экран после некоторого промежутка времени (например, экран заставки).
- Экранами необходимо каким-либо образом управлять (например, нам нужно отслеживать текущий экран и иметь возможность переходить с него на другой, что на самом деле означает уничтожение первого экрана и установку второго экрана в качестве основного).
- Игра должна предоставить экранам доступ к различным модулям (для графики, звука, ввода и т. д.), чтобы они могли загружать ресурсы, отслеживать действия пользователя, прорисовывать фреймбуфер и т. д.
- Поскольку все наши игры будут проходить в режиме реального времени (что означает постоянное движение и обновление компонентов), нам необходимо обеспечить как можно более частое обновление состояния текущего экрана. Обычно мы делаем это внутри цикла, называемого основным. Данный цикл прерывается, когда пользователь выходит из игры. Одна итерация цикла называется кадром. Количество кадров в секунду (fps), которое мы можем рассчитать, называется частотой обновления.

- Говоря о времени, нам также необходимо отслеживать промежуток времени, прошедший со времени появления последнего кадра. Это нужно для кадронезависимого движения (которое мы скоро обсудим).
- В игре также необходимо отслеживать состояние окна (например, поставлена игра на паузу или нет) и информировать текущий экран об этих событиях.
- Игровая среда будет иметь дело с настройкой окна и созданием компонента пользовательского интерфейса, который мы будем прорисовывать и от которого будем получать пользовательский ввод.

Давайте сделаем из всего этого немного псевдокода, пока игнорируя такие аспекты управления окном, как постановка на паузу и возврат из нее:

```
createWindowAndUIComponent();

Input input = new Input();
Graphics graphics = new Graphics();
Audio audio = new Audio();
Screen currentScreen = new MainMenu();
Float lastFrameTime = currentTime();
while( !userQuit() ) {
   float deltaTime = currentTime() – lastFrameTime;
   lastFrameTime = currentTime();

   currentScreen.updateState(input, deltaTime);
   currentScreen.present(graphics, audio, deltaTime);
}

cleanupResources();
```

Мы начали с создания окна для нашей игры и компонента пользовательского интерфейса для прорисовки и получения пользовательского ввода. Далее мы инициализируем все необходимые для низкоуровневой работы модули. Мы создаем наш стартовый экран и делаем его текущим, а также записываем текущее время. Далее происходит вход в главный цикл, прерываемый в тот момент, когда пользователь демонстрирует свое желание выйти из игры.

Внутри основного цикла мы рассчитываем так называемый дельта-интервал — время, которое прошло с начала прошлого кадра. Затем записываем время начала текущего кадра. Дельта-интервал и текущее время обычно измеряются в секундах. В случае с экраном дельта-интервал показывает, как много времени прошло с момента последнего его обновления. Эта информация необходима нам для кадронезависимого движения (о котором мы очень скоро поговорим).

Наконец, мы просто обновляем состояние текущего экрана и демонстрируем его пользователю. Это обновление зависит от дельта-интервала и состояния; следовательно, мы передаем эти данные экрану. Представление состоит из визуализации состояния экрана во фреймбуфере, а также из воспроизведения любого запрашиваемого экраном аудиоресурса (например, звука выстрела). Метод представления иногда также должен знать, сколько времени прошло с момента его последнего вызова.

При прерывании работы главного цикла мы можем очистить и освободить все ресурсы, после чего закрыть окно.

Именно таким образом практически каждая игра ведет себя на высоком уровне — обработка ввода, обновление состояния, представление его пользователю и повторение всего этого до бесконечности (или до того момента, когда игроку надоест).

Приложения с пользовательским интерфейсом на современных операционных системах обычно не работают в режиме реального времени. Они используют событийную парадигму, при которой операционная система оповещает приложение о наступающих событиях ввода, а также о необходимости визуализации. Это достигается при помощи функций обратного вызова, которые приложение регистрирует в операции при старте. Это они отвечают за обработку поступающих оповещений о событиях. Все это происходит в рамках так называемого пользовательского потока — главного потока приложения. Это вообще хорошая идея — возвращать ответы от функций обратного вызова как можно чаще, поэтому нам не стоит реализовывать наш главный цикл в одной из них.

Вместо этого мы размещаем главный цикл игры в отдельном потоке, создаваемом при старте игры. Это означает, что нам необходимо соблюсти некоторые меры предосторожности при получении событий из пользовательского потока (таких как ввод или события окна). Но этими деталями мы займемся позже, когда будем реализовывать игровой фреймворк на Android. Просто помните, что нам необходимо будет синхронизировать пользовательский поток и главный цикл программы в определенные моменты.

Игровые и экранные интерфейсы

Вооружившись полученными знаниями, попытаемся разработать интерфейс игры. Он должен реализовывать следующие функции:

- создание окна и компонента пользовательского интерфейса, а также функций обратного вызова для обработки экранных и пользовательских событий;
- запуск потока главного цикла программы;
- отслеживание текущего экрана, обновление и представление его в каждой итерации главного цикла (то есть в кадре);
- отслеживание любых событий окна (например, постановки на паузу и возобновления игры) из потока пользовательского интерфейса и передача их экрану для соответствующего изменения состояния;
- предоставление доступа ко всем ранее разработанным модулям: `Input`, `FileIO`, `Graphics` и `Audio`.

Как разработчики игр мы равнодушны к тому, в каком потоке работает наш главный цикл и нужно ли ему синхронизироваться с интерфейсным потоком. Нам бы просто хотелось создать различные игровые экраны с небольшой помощью низкоуровневых модулей и оповещений об оконных событиях. Поэтому мы создадим очень простой интерфейс `Game`, скрывающий от нас все сложные подробности,

а также абстрактный класс Screen, который будет использоваться для реализации наших экранов (код интерфейса показан в листинге 3.8).

Листинг 3.8. Интерфейс Game

```
package com.badlogic.androidgames.framework;

public interface Game {
    public Input getInput();

    public FileIO getFileIO();

    public Graphics getGraphics();

    public Audio getAudio();

    public void setScreen(Screen screen);

    public Screen getCurrentScreen();

    public Screen getStartScreen();
}
```

Как вы, вероятно, и ожидали, здесь описаны несколько методов-получателей, возвращающих экземпляры низкоуровневых модулей (которые наша реализация Game будет создавать и отслеживать). Метод Game.setScreen() позволяет установить для игры текущий экран. Эти методы будут реализованы один раз (одновременно с созданием внутренних потоков, управлением окном и главным циклом для обновления и представления экрана). Метод Game.getCurrentScreen() возвращает текущий активный экран.

Для реализации интерфейса Game мы позже будем использовать абстрактный класс AndroidGame, реализующий все методы, кроме Game.getStartScreen() — он останется абстрактным. Когда мы создадим экземпляр AndroidGame для нашей игры, то унаследуем и реализуем в нем метод Game.getStartScreen(), возвратив экземпляр первого экрана игры.

Чтобы вы получили некоторое впечатление о легкости, с которой будет создаваться наша игра, посмотрите на пример (представьте, что мы уже реализовали класс AndroidGame):

```
public class MyAwesomeGame extends AndroidGame {
    public Screen getStartScreen () {
        return new MySuperAwesomeStartScreen(this);
    }
}
```

Впечатляет, не правда ли? Все, что нам необходимо сделать, — реализовать экран, с которого должна начинаться наша игра, а дальше AndroidGame, от которого мы наследовались, сделает все остальное. Заглядывая вперед — он же будет заставлять экран MySuperAwesomeStartScreen обновляться и прорисовываться в потоке

главного цикла. Обратите внимание — в нашей реализации Screen мы передали конструктору MyAwesomeGame сам экземпляр.

ПРИМЕЧАНИЕ

Если у вас возник вопрос, что же на самом деле создает наш класс MyAwesomeGame, я вам подскажу: AndroidGame будет наследоваться от Activity, автоматически создаваемого операционной системой Android при запуске пользователем игры.

Последним элементом нашей головоломки будет абстрактный класс Screen. Мы реализуем его в виде класса, а не интерфейса, поскольку мы уже проделали для него некоторую черновую работу. В этом случае нам придется писать меньше шаблонного кода в реальных реализациях Screen. Абстрактный класс Screen показан в листинге 3.9.

Листинг 3.9. Класс Screen

```
package com.badlogic.androidgames.framework;

public abstract class Screen {
    protected final Game game;

    public Screen(Game game) {
        this.game = game;
    }

    public abstract void update(float deltaTime);

    public abstract void present(float deltaTime);

    public abstract void pause();

    public abstract void resume();

    public abstract void dispose();
}
```

Тут выясняется, что предварительная работа нам пригодится. Конструктор получает экземпляр Game и хранит его в члене с модификатором final, доступном всем подклассам. Благодаря использованию этого механизма мы достигаем двух целей:

- получаем доступ к низкоуровневым модулям Game для воспроизведения звука, прорисовки экрана, получения реакции пользователя, а также чтения и записи файлов;
- можем устанавливать новый текущий экран, вызывая при необходимости метод Game.setScreen() (например, при нажатии кнопки перехода на другой экран).

Первый пункт достаточно очевиден: нашей реализации Screen необходим доступ к этим модулям, чтобы делать действительно важные вещи (вроде прорисовки стада бешеных единорогов).

Вторая возможность позволяет нам легко реализовать переходы между экранами (экземплярами Screen). Каждый экземпляр может принимать решение о переходе на другой определенный экземпляр Screen, основываясь на своем состоянии (например, при нажатии кнопки).

Предназначение методов Screen.update() и Screen.present() вполне понятно: они обновляют состояние экрана и представляют его пользователю соответственно. Экземпляр Game вызывает их один раз при каждой итерации главного цикла.

Методы Screen.pause() и Screen.resume() вызываются при постановке игры на паузу и выходе из нее. Эти действия также производятся экземпляром Game и относятся к текущему активному экрану.

Метод Screen.dispose() вызывается экземпляром Game при вызове Game.setScreen(). В результате текущий экземпляр Screen освобождает системные ресурсы (например, графические активы, хранящиеся в Pixmaps), чтобы получить свободное место в памяти для нового экрана. Кроме того, вызов Screen.dispose() — последний шанс для экрана убедиться, что вся необходимая информация сохранена.

Простой пример

Продолжим с нашим примером MySuperAwesomeGame — вот весьма простая реализация класса:

MySuperAwesomeStartScreen:

```
public class MySuperAwesomeStartScreen extends Screen {
    Pixmap awesomePic;
    int x;

    public MySuperAwesomeStartScreen(Game game) {
        super(game);
        awesomePic = game.getGraphics().newPixmap("data/pic.png",
                PixmapFormat.RGB565);
    }

    @Override
    public void update(float deltaTime) {
        x += 1;
        if (x > 100)
            x = 0;
    }

    @Override
    public void present(float deltaTime) {
        game.getGraphics().clear(0);
        game.getGraphics().drawPixmap(awesomePic, x, 0, 0, 0,
                awesomePic.getWidth(), awesomePic.getHeight());
    }

    @Override
    public void pause() {
```

```
        // тут не надо ничего делать
    }

    @Override
    public void resume() {
        // тут тоже ничего не надо делать
    }

    @Override
    public void dispose() {
        awesomePic.dispose();
    }
}
```

Рассмотрим, что этот класс делает в связке с `MySuperAwesomeGame`.

1. При создании `MySuperAwesomeGame` он создает окно, компонент интерфейса (который мы прорисовываем и от которого получаем события пользовательского ввода), функции обратного вызова для обработки событий, а также поток главного цикла. Наконец, этот класс вызывает собственный метод `MySuperAwesomeGame.getStartScreen()`, возвращающий экземпляр класса `MySuperAwesomeStartScreen()`.
2. В конструкторе `MySuperAwesomeStartScreen()` загружаем растровое изображение с диска и храним его в переменной-члене. Таким образом, завершается установка нашего экрана, и управление возвращается классу `MySuperAwesomeGame`.
3. Поток главного цикла теперь постоянно будет вызывать методы `MySuperAwesomeStartScreen.update()` и `MySuperAwesomeStartScreen.render()` только что созданного нами экземпляра.
4. В методе `MySuperAwesomeStartScreen.update()` увеличиваем значение переменной `x` на каждом новом кадре. Эта переменная хранит координату *x* изображения, которое мы хотим прорисовывать. Ее значение обнуляется, когда становится равным больше 100.
5. В методе `MySuperAwesomeStartScreen.render()` очищаем фреймбуфер, заполняя его черным цветом (0x00000000 = 0), после чего прорисовываем объект `Pixmap` в позиции (*x*; 0).
6. Поток главного цикла повторяет шаги 3–5 до тех пор, пока пользователь не выйдет из игры, нажав на устройстве кнопку «Назад». В этом случае экземпляр `Game` вызовет метод `MySuperAwesomeStartScreen.dispose()`, очищающий `Pixmap`.

И вот она, наша первая игра! Все, что пользователь увидит на ее экране, — картинку, движущуюся слева направо. Не слишком впечатляющий геймплей, но мы продолжим над ним работать. Заметьте, что в Android игра может быть приостановлена и возобновлена в любой момент. Наша реализация `MyAwesomeGame` вызывает в этих случаях методы `MySuperAwesomeStartScreen.pause()` и `MySuperAwesomeStartScreen.resume()`, что приводит к приостановке выполнения главного цикла программы на время паузы.

Последняя проблема, которую нам сейчас необходимо обсудить, — кадронезависимое движение.

Кадронезависимое движение

Представим, что пользователь запустил нашу игру на устройстве, поддерживающем частоту обновления 60 кадров в секунду. Поскольку мы увеличиваем значение `MySuperAwesomeStartScreen.x` на 1 в каждом кадре, наш `Pixmap` переместится на 100 пикселов за 100 кадров. При частоте обновления 60 кадров в секунду (fps) достижение положения (100; 0) займет примерно 1,66 секунды.

Теперь допустим, что другой пользователь играет в нашу игру на другом устройстве, обеспечивающем частоту обновления 30 кадров в секунду. В данном случае наш `Pixmap` будет перемещаться в секунду на 30 пикселов, поэтому точка с координатами (100,0) будет достигнута через 3,33 секунды.

Это плохо. В нашей простой игре это может не иметь значения, но представьте на месте `Pixmap` Супер Марио и подумайте, что может для него значить такая зависимость от аппаратных возможностей. Например, мы нажимаем стрелку «Вправо», и Марио бежит в ту же сторону. В каждом кадре он продвигается на 1 пиксел (как и `Pixmap`). На устройстве с частотой кадров 60 кадров в секунду Марио будет бежать вдвое быстрее, чем на телефоне с частотой 30 кадров в секунду! Таким образом, характеристики аппарата могут полностью изменять показатели производительности игры. Нам необходимо это исправить.

Решение этой проблемы — кадронезависимое движение. Вместо перемещения нашего объекта `Pixmap` (или Марио) на фиксированную величину за один кадр, мы определим скорость его движения в юнитах за секунду. Допустим, мы хотим, чтобы наш `Pixmap` перемещался на 50 пикселов в секунду. Помимо этого значения нам также нужна информация о времени, прошедшем с последнего перемещения `Pixmap`. И это как раз тот момент, когда вступает в игру тот странный дельта-интервал. Он показывает нам, сколько времени прошло с последнего обновления. Таким образом, наш метод `MySuperAwesomeStartScreen.update()` должен выглядеть примерно так:

```
@Override
public void update(float deltaTime) {
    x += 50 * deltaTime;
    if(x > 100)
        x = 0;
}
```

Теперь, если наша игра проходит при частоте 60 кадров в секунду, передаваемый методу дельта-интервал всегда будет равен примерно 0,016 секунды (1/60). Таким образом, в каждом кадре будет продвижение на 0,83 (50 · 0,016) пиксела, а за секунду — около 100 пикселов (60 · 0,85)! Проверим результаты при частоте 30 кадров в секунду: 1,66 пиксела (50 · 1 / 30). Умножая на 30, снова получаем 100 пикселов в секунду. Итак — неважно, на каком устройстве запускается наша игра, вся анимация и движение в ней будут всегда соответствовать текущему времени.

Однако если вы попробовали проверить эти расчеты на предшествующем коде, то увидели, что наш `Pixmap` на самом деле вообще не двигается при частоте 60 кадров в секунду. Так происходит из-за ошибки в нашем коде. Попробуйте угадать, где именно. Это довольно малозаметная, но распространенная ловушка, часто подстерегающая разработчиков игр. Переменная `x`, которую мы увеличиваем в каждом кадре, определена как `integer`. Прибавление 0,83 к типу `integer` не дает никакого эффекта. Для исправления этой неприятности нужно просто заменить тип данных переменной `x` на `float`. Кроме того, необходимо добавить сумму при вызове `Graphics.drawPixmap()`.

ПРИМЕЧАНИЕ

Хотя расчеты значений с плавающей точкой в Android обычно осуществляются медленнее, чем для целочисленных значений, эта разница обычно ничтожна, поэтому мы можем не обращать на нее внимания.

И на этом все о нашей игровой среде. Мы можем напрямую преобразовать экраны нашего мистера Нома в классы и интерфейсы фреймворка. Конечно, следует заняться еще кое-какими деталями реализации, но мы оставим это для последующих глав. Пока же вы можете собой гордиться — вы дочитали эту главу до конца и теперь готовы стать разработчиком игр для платформы Android (и не только для нее).

Подводя итог

Изучив к этому моменту столько насыщенных и информативных страниц книги, вы должны получить неплохое представление о том, из каких этапов состоит процесс создания игры. Мы изучили несколько самых популярных на Android Market жанров и сделали некоторые выводы. Мы решили нарисовать игру с нуля, используя только ножницы, ручку и несколько листов бумаги. Наконец, мы исследовали теоретические основы процесса разработки игр и даже создали набор интерфейсов и абстрактных классов, которые будем использовать в книге далее для реализации наших идей, основанных на теоретических идеях. Если вы думаете, что вам необходимо продвинуться дальше пределов, обозначенных в этой главе, обратитесь за помощью к Сети — все ключевые слова вы знаете. Понимание принципов — ключ к разработке стабильных и высокопроизводительных приложений. Теперь давайте реализовывать нашу игровую среду для Android!

4 Android для разработчиков игр

Фреймворк Android весьма обширен и иногда может запутать. Для каждой возможной решаемой вами задачи присутствует собственный API. Конечно, сначала вам необходимо изучить эти API. К счастью для нас, разработчиков игр, необходим предельно ограниченный их набор. Все, что нам нужно, — окно с единственным компонентом пользовательского интерфейса для прорисовки и получения пользовательского ввода, а также возможность фонового воспроизведения звука. Этого будет достаточно для реализации придуманной нами в предыдущей главе игровой среды. Вы изучите необходимый минимум API Android для превращения мистера Нома в реальность и будете удивлены тем, как мало вам на самом деле нужно знать об этих API для достижения этой цели. Напомню, какие именно ингредиенты нам нужны:

- управление окном;
- пользовательский ввод;
- файловый ввод/вывод;
- звук;
- графика.

Для каждого из этих модулей существует эквивалент среди API фреймворка приложения. Мы найдем и выберем необходимые нам API для обработки данных модулей, обсудим их содержание и, наконец, реализуем соответствующие интерфейсы игровой среды, спроектированной нами в прошлой главе.

Однако перед погружением в процесс управления окном в платформе Android нам необходимо еще раз вернуться к вопросу, затронутому в главе 2: определению нашего приложения через файл манифеста.

Определение приложения Android: файл манифеста

Приложение Android может состоять из большого количества различных компонентов.

- Активности — компоненты представления пользовательского интерфейса и взаимодействия с ним.

- Сервисы — процессы, работающие в фоновом режиме и не имеющие видимого интерфейса. Например, сервис может отвечать за получение почтовым сервером новых писем.
- Поставщики содержимого — делают части вашего приложения доступными для других приложений.
- Намерения — это сообщения, создаваемые системой или самими приложениями. Они могут оповещать нас о системных событиях (например, извлечение карты памяти или подключение USB-кабеля), а также используются системой для запуска компонентов нашего приложения (например, активностей). Мы также можем вызывать собственные намерения, чтобы попросить другие приложения выполнить необходимые нам действия (например, открыть фотогалерею или сделать фотографию с помощью стандартного приложения `Камеры`).
- Приемники вещания — реагируют на особые намерения и могут выполнять действия: запустить определенную активность или сгенерировать другое намерение для операционной системы.

Приложение Android не имеет одной точки входа, к чему мы привыкли при разработке для настольных систем (например, в виде метода `main()` в случае с Java). Вместо этого компоненты приложения Android запускаются или выполняют определенные действия при получении соответствующих намерений.

Набор компонентов и намерений, на которые эти компоненты реагируют, определяется в файле манифеста приложения. Система Android использует этот файл манифеста, чтобы знать, из чего состоит приложение, а также о том, какая активность запускается при его старте.

ПРИМЕЧАНИЕ

В этой книге нас интересуют только активности, поэтому из всего файла манифеста мы изучим только то, что касается этого вида компонента. Если вы хотите копнуть поглубже, то можете получить более подробную информацию о файле манифеста на сайте Android Developers.

Файл манифеста выполняет намного больше задач, чем просто определение компонентов приложения. Следующий список показывает соответствующие части файла манифеста в контексте разработки игр:

- версия приложения, отображаемая и используемая на Android Market;
- версии Android, на которых наше приложение может работать;
- профили оборудования, требуемые приложением (например, наличие мультитач, определенные разрешения дисплея, поддержка OpenGL ES 2.0);
- разрешения на использование определенных компонентов (запись на SD-карту, доступ к сети и т. д.).

В следующих подразделах мы создадим шаблонный манифест, который сможем повторно использовать во всех проектах, которые разработаем в процессе изучения книги. Для этого пройдемся по соответствующим XML-тегам, необходимым для определения приложения.

Элемент <manifest>

Тег <manifest> является корневым элементом файла AndroidManifest.xml. Вот простой пример:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.helloworld"
      android:versionCode="1"
      android:versionName="1.0"
      android:installLocation="preferExternal">
...
</manifest>
```

Если ранее вы уже имели дело с XML-файлами, то вам должно быть все понятно про первую строчку. Тег <manifest> определяет пространство имен android, используемое во всем файле манифеста. Атрибут package задает корневой пакет нашего приложения. Позже мы соотнесем определенные классы нашего приложения с этим именем пакета.

Атрибуты versionCode и versionName определяют версию нашего приложения в двух формах. versionCode — целое число, которое необходимо увеличивать каждый раз, когда мы публикуем новую версию приложения. Оно используется в Android Market для отслеживания нашей версии приложения. versionName показывается пользователям Android Market при посещении страницы приложения. Здесь мы можем применять любое строковое значение.

Атрибут installLocation доступен нам, только если мы в Eclipse опеределили в качестве цели нашего приложения версию Android 2.2 и выше. Он определяет, где должно быть установлено приложение. Строка preferExternal сообщает системе, что приложение должно устанавливаться на карту памяти. Такое возможно только с версии Android 2.2, для более старых прошивок параметр будет игнорироваться. Начиная с Android 2.2 приложение всегда будет устанавливаться во внутреннее хранилище, если существует такая возможность.

Все атрибуты XML-элементов в файле манифеста имеют префикс, означающий принадлежность к пространству имен android. Для краткости далее я буду опускать этот префикс, говоря об определенном атрибуте.

Внутри элемента <manifest> мы определяем компоненты приложения. разрешения, профили оборудования и поддерживаемые версии Android.

Элемент <application>

Как и в случае с элементом <manifest>, обсудим элемет <application> на примере:

```
<application android:icon="@drawable/icon" android:label="@string/app_name"
android:debuggable="true">
...
</application>
```

Кое-что тут выглядит странно. Что означают строки @drawable/icon и @string/app_name? При разработке стандартного приложения Android мы обычно создаем

много XML-файлов, каждый из которых определяет некоторую часть нашего приложения. Чтобы иметь возможность точно идентифицировать эти части, нам нужна возможность обращаться к ресурсам, не определенным в файлах XML (например, к изображениям или интернационализированным строкам). Эти ресурсы располагаются в подпапках каталога `/res` (мы обсуждали это в главе 2, когда рассматривали проект `hello world` в Eclipse).

Для обращения к этим ресурсам мы используем указанную выше нотацию. Знак `@` указывает, что нам необходим внешний по отношению к манифесту ресурс. Строка после `@` определяет тип ресурса, с которым мы связываемся, что напрямую связывает его с одним из каталогов или файлов в папке `/res`. Последняя часть определяет имя ресурса — в примере выше это изображение с названием `icon` и строка `app_name`. В случае с изображением имя ресурса — это название файла, расположенного в каталоге `res/drawable/`. Обратите внимание — имя файла указано без расширения (PNG или JPG). Android добавит его автоматически, просканировав папку `res/drawable/`. Строка `app_name` определена в файле `res/values/strings.xml`, содержащем строки, используемые приложением. Имя строки определено в этом файле.

ПРИМЕЧАНИЕ

Обработка ресурсов в Android очень гибка, но одновременно сложна. В данной книге я решил почти полностью пропустить эту тему по двум причинам: отсутствие необходимости в этом для написания игр и желание иметь полный контроль над нашими ресурсами. У Android есть привычка изменять ресурсы, размещенные в каталоге res/ (особенно изображения в drawables). Как разработчикам игр нам это совсем не нужно. Единственная вещь, которую я бы предложил, — использовать систему управления ресурсами Android для интернационализированных строк. В нашей книге это не понадобится; вместо этого мы будем применять более подходящий для игровых приложений инструмент — дружественную папку assets/, в которой наши ресурсы никто не будет трогать и которая позволит нам создавать внутри собственную иерархию.

Теперь значение атрибутов элемента должно стать более понятным. Атрибут `icon` определяет изображение из папки `res/drawable/`, используемое в качестве значка нашего приложения. Этот значок будет показываться на Android Market, а также на устройстве. Кроме того, этот значок применяется по умолчанию для всех активностей, определенных внутри элемента `<application>`.

Атрибут `label` задает строку, которая будет показываться для нашего приложения в программе для запуска приложений. В примере это строка в файле `res/values/string.xml`, соответствующая тому, что мы определили при создании проекта в Eclipse. Мы можем установить ему любое значение, например `My Super Awesome Game`. Эта метка также является меткой по умолчанию для всех активностей, определенных внутри элемента `<application>`, она будет показываться в строке заголовка нашего приложения.

Атрибут `debuggable` определяет, может ли наше приложение подвергаться отладке. В процессе разработки лучше установить ему значение `true` (иначе отладка в Eclipse будет невозможна), но при размещении готовой программы на Android Market желательно задать значение `false`.

Мы обсудили очень маленькую часть атрибутов, которые вы может установить для элемента <application>. Однако для наших целей этого вполне достаточно. Если вы хотите узнать побольше, то сможете найти полную документацию на сайте Android Developers.

Элемент <application> содержит определения всех компонентов приложения, включая активности и сервисы, а также любые используемые дополнительные библиотеки.

Элемент <activity>

Чем дальше, тем интереснее. Вот гипотетический пример для нашей игры «Мистер Ном»:

```
<activity android:name=".MrNomActivity"
          android:label="Mr. Nom"
          android:screenOrientation="portrait">
          android:configChanges="keyboard|keyboardHidden|orientation">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Для начала взглянем на атрибуты тега <activity>.

- name — определяет имя класса активности, которое связано с атрибутом package, установленным нами в элементе <manifest>. Вы также можете ввести здесь полное имя класса.
- label — такой же атрибут мы уже определили ранее в теге <application>. Эта строка отражается в строке заголовка активности (если она присутствует). Кроме того, данная метка используется в качестве текста, показываемого в программе для запуска приложений, если данная активность является точкой вхождения для приложения. Если не определить данный атрибут, будет использоваться соответствующий атрибут элемента <application>. Заметьте, что в данном случае я применил саму строку, а не ссылку на нее в файле strings.xml.
- screenOrientation — определяет ориентацию экрана, используемую данной активностью. Для нашего «Мистера Нома» я определил портретную ориентацию (только в ней она и будет использоваться). При желании можно определить и ландшафтную ориентацию. Оба варианта заставят активность использовать определенную ориентацию в течение всего ее жизненного цикла вне зависимости от того, как на самом деле ориентировано устройство. Если этот атрибут не будет определен, активность будет применять текущую ориентацию дисплея, основываясь на данных акселерометра. Это также означает, что при любом изменении ориентации активность уничтожается и создается заново — в случае с играми это не очень желательно. Поэтому режим ориентации обычно фиксируется в одном из вариантов (портретном или ландшафтном).

- `configChanges` — переориентация устройства или сдвиг клавиатурного блока рассматриваются как изменение конфигурации. В случае подобных изменений Android уничтожает и перезагружает наше приложение для применения изменений. В случае с играми это не лучший вариант поведения. Атрибут `configChanges` элемента `<activity>` помогает справиться с этой проблемой. Он дает возможность определить, какие изменения конфигурации мы хотим обрабатывать сами, без пересоздания активности. Множественные изменения конфигурации могут быть заданы с использованием символа | для склеивания нескольких вариантов. В примере выше мы сами обрабатываем клавиатуру, `keyboardHidden` и изменение ориентации.

Как и в случае с элементом `<application>`, на самом деле атрибутов для элемента `<activity>`, конечно, намного больше. Но для разработки игр мы ограничимся описанными выше.

Вы, должно быть, заметили, что элемент `<activity>` не пуст — он содержит в себе другой элемент, а тот, в свою очередь, содержит два. Для чего они нужны?

Как я указывал ранее, не существует единой точки входа в приложение Android. Вместо этого у нас есть несколько таких точек в виде активностей и сервисов, запускающихся при получении намерений от системы или сторонних приложений. Нам каким-то образом необходимо сообщить Android, какие активности и сервисы (и как именно) будут реагировать на определенные намерения. Именно для этого нужен атрибут `<intent-filter>`.

В рассматриваемом примере мы определили два типа фильтров намерений: `<action>` и `<category>`. Элемент `<action>` сообщает системе, что наша активность является главной точкой входа в программу. Элемент `<category>` указывает, что мы хотим, чтобы активность была добавлена в программу для запуска приложений. Оба элемента вместе позволяют Android сделать вывод, что при нажатии значка приложения в программе для запуска приложений должна запускаться именно эта активность. Значение элементов `<action>` и `<category>` — имя, определяющее намерение, на которое следует реагировать. `android.intent.action.MAIN` — специальное намерение, которое система Android использует для запуска главной активности приложения. Намерение `android.intent.category.LAUNCHER` используется для оповещения операционной системы о том, должна ли данная активность иметь возможность запуска через программу для запуска приложений (application launcher).

В нашем примере есть только одна активность с двумя этими фильтрами. Однако стандартное приложение Android почти всегда имеет несколько активностей, и все они также должны быть определены в файле `manifest.xml`. Вот пример определения подобной субактивности:

```
<activity android:name=".MySubActivity"
          android:label="Sub Activity Title"
          android:screenOrientation="portrait">
          android:configChanges="keyboard|keyboardHidden|orientation"/>
```

Как видите, здесь не определены фильтры намерений — только четыре атрибута, которые мы рассмотрели ранее. При подобном определении активности она

становится доступна только нашему приложению. Запускается такая активность специальным видом намерения (например, при нажатии соответствующей кнопки в другой активности). Чуть позже в этом разделе мы рассмотрим вопрос программного запуска активности.

Подводя итог, можно сказать, что у нас есть одна активность, в которой определены два фильтра намерений (что делает ее главной точкой вхождения для нашего приложения). Для всех остальных активностей мы опускаем определения фильтров намерений, поэтому они являются внутренними для нашего приложения и поэтому запускаются программно.

ПРИМЕЧАНИЕ

Как было сказано ранее, в наших играх будет только одна активность. Она будет обладать тем же определением фильтров намерений, что и в данном примере. Причина, по которой я описал определение множественных активностей, в том, что у нас в планах значится создание специального тестового приложения с множественными активностями. Не волнуйтесь, это будет совсем несложно.

Элемент <uses-permission>

Теперь оставим элемент `<application>` и вернемся к элементам, определенным нами в качестве дочерних для элемента `<manifest>`. Один из них — `<uses-permission>`.

Android обладает тщательно проработанной моделью обеспечения безопасности. Каждое приложение работает в рамках собственного процесса и виртуальной машины, под собственным пользователем Linux и группой и не может влиять на другие приложения. Кроме того, система ограничивает применение своих ресурсов (сетевой доступ, карта памяти, диктофон и т. д.). Если нашему приложению необходимо их использовать, мы должны запросить на это разрешение с помощью элемента `<uses-permission>`.

Разрешение всегда запрашивается в следующей форме (строка определяет название необходимого нам разрешения):

```
<uses-permission android:name="string"/>
```

Вот несколько названий разрешений, которые нам могут понадобиться:

- `android.permission.RECORD_AUDIO` — получение доступа к оборудованию для записи звука;
- `android.permission.INTERNET` — получение доступа ко всем сетевым API, что позволяет, например, получить изображение из Интернета или загрузить лучшие результаты;
- `android.permission.WRITE_EXTERNAL_STORAGE` — дает возможность читать и записывать файлы на внешний носитель (обычно на SD-карту);
- `android.permission.WAKE_LOCK` — позволяет осуществлять так называемую защиту от блокировки. С ее помощью мы можем препятствовать переходу устройства в спящий режим при отсутствии нажатий экрана в течение некоторого времени. Такое может произойти, например, при управлении игрой с помощью акселерометра.

Для получения доступа к сетевым API мы определяем следующий элемент в качестве дочернего для <mainfest>:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Для всех дополнительных разрешений мы просто добавляем еще элементы <uses-permission>.

Вы можете определить гораздо больше разрешений — я снова адресую вас к документации Android. Нам сейчас нужны только те, которые мы только что рассмотрели.

Если вы забудете добавить на что-то разрешение (например, на доступ к карте SD), сообщение об этом появится в LogCat, но его там не всегда легко обнаружить. Подумайте хорошо о разрешениях, необходимых вашему приложению, и определите их при первоначальном создании проекта.

Еще одна вещь, о которой необходимо сказать, — когда пользователь инсталлирует приложение, его первым делом попросят просмотреть все разрешения, требуемые программой. Многие просто пропускают этот текст и доверяют автору. Однако некоторые более ответственно относятся к своим решениям и внимательно изучают разрешения. Если вы просите что-то подозрительное (например, возможность отсылать платные СМС или получать координаты пользователя), то можете получить неодобрительную реакцию от них в разделе Комментарии на Android Market. Если вам необходимо одно из таких проблемных разрешений, сообщите пользователю, зачем именно оно вам нужно, в описании программы. Но лучше всего вообще избегать подобных разрешений.

Элемент <uses-feature>

Если вы сами — пользователь Android и владеете устройством со старой версией ОС (например, 1.5), то, видимо, заметили, что некоторые отличные приложения не показываются вам на Android Market. Причина этого — использование элемента <uses-feature> в файле манифеста приложения.

Android Market фильтрует все доступные приложения, исходя из вашего профиля оборудования. С помощью элемента <uses-feature> приложение может определить, какие аппаратные возможности ему необходимы (например, мультитач или поддержка OpenGL ES 2.0). Любое устройство, не обладающее указанными возможностями, не будет видеть приложение.

Элемент <uses-feature> обладает следующими атрибутами:

```
<uses-feature android:name="string" android:required=["true" | "false"]
android:glEsVersion="integer" />
```

Атрибут name определяет саму возможность (функцию). Атрибут required сообщает фильтру, необходима ли нам эта возможность для работы программы или просто было бы хорошо ее иметь. Последний атрибут опционален и используется только в случае, если для работы обязательно требуется определенная версия OpenGL ES.

Для разработчиков игр наиболее важными являются следующие требования:

- `android.hardware.touchscreen.multitouch` — требует от устройства наличия мультитач-экрана, поддерживающего некоторые распространенные возможности. Такие типы дисплеев могут иметь проблемы с отслеживанием независимых касаний, поэтому вам нужно подумать, достаточно ли этих возможностей для вашей игры;
- `android.hardware.touchscreen.multitouch.distinct` — старший брат предыдущей возможности. Он запрашивает полный набор мультитач-способностей, позволяющих реализовать все доступные функции.

Более подробно мы рассмотрим мультитач позже в этой книге. На данный момент нам необходимо помнить, что, если наша игра требует дисплей с мультитач, мы можем отфильтровать все устройства, не поддерживающие эту возможность, указав элемент `<uses-feature>` с одной из нескольких возможностей, например, так:

```
<uses-feature android:name="android.hardware.touchscreen.multitouch"
android:required="true"/>
```

Другая полезная вещь для разработчиков игр — определение необходимой версии OpenGL ES. В нашей книге мы имеем дело с OpenGL ES 1.0 и 1.1. В этом случае нам нет необходимости задавать соответствующий элемент `<uses-feature>`, поскольку две вышеупомянутые версии не слишком друг от друга отличаются. Однако любое устройство, обладающее поддержкой OpenGL ES 2.0, намного опережает по своим графическим возможностям предшественников. Если наша игра визуально сложна и требует много вычислительной мощности, мы можем затребовать версию OpenGL ES 2.0, чтобы приложение было видно только аппаратам, поддерживающим эту библиотеку. Заметьте, что мы не используем OpenGL ES 2.0, а просто фильтруем устройства таким образом, чтобы наш код на OpenGL ES 1.x получил достаточно вычислительной мощности. Сделать это можно так:

```
<uses-feature android:glEsVersion="0x00020000" required="true"/>
```

Теперь наша игра будет видна только устройствам с поддержкой OpenGL ES 2.0, что гарантирует нам использование мощного графического процессора.

ПРИМЕЧАНИЕ

Эта возможность на некоторых устройствах обрабатывается некорректно, что делает приложение невидимым для вполне подходящих устройств. Используйте данную возможность с осторожностью.

Учтите — каждое новое требование к аппаратной составляющей потенциально уменьшает количество устройств, которым будет видно ваше приложение на Android Market (а это напрямую влияет на его продажи). Подумайте дважды, прежде чем добавлять их. Например, если стандартный режим игры требует мультитач, но вы хотите рассмотреть возможность играть в нее и на обычных дисплеях, вам может потребоваться разделить код на отдельные ветки, чтобы охватить больший сегмент рынка.

Элемент <uses-sdk>

Последний элемент, который мы поместим в наш файл манифеста, — <uses-sdk>, дочерний для <manifest>. Мы неявно уже задавали его при создании проекта Hello World в главе 2, когда определяли минимальную версию SDK в диалоговом окне New Android Project (Новый проект Android). Итак, для чего нужен этот элемент? Рассмотрим пример:

```
<uses-sdk android:minSdkVersion="3" android:targetSdkVersion="9"/>
```

Как обсуждалось в главе 2, каждая версия Android имеет некий целочисленный параметр, также известный как версия SDK. Элемент <uses-sdk> определяет минимальную версию SDK, которую будет поддерживать наше приложение, то есть его целевую версию.

Этот элемент позволяет развертывать приложение, использующее API из более новых версий, на устройствах с более старой версией SDK. Показательный пример — API для мультитач, поддерживаемые начиная с 5-й версии SDK (Android 2.0). При создании проекта в Eclipse мы используем цель сборки, поддерживающую этот API, например SDK 5 или новее (я обычно задействую новейшую версию, 9 на момент написания). Если мы хотим, чтобы наша игра запускалась также на устройствах с версией SDK 3, необходимо определить значение атрибута minSdkVersion в файле манифеста. Конечно, нам нужно будет позаботиться о том, чтобы не использовать API, недоступные в более старых версиях SDK (по крайней мере на устройствах с версией 1.5). На аппаратах с более новой версией новые функции из новых API останутся доступными.

Описанная выше конфигурация обычно вполне подходит для большинства игр (только если вы не хотите разделить ветки программного кода для разных версий API — в этом случае будет правильным установить значение атрибута minSdkVersion равным минимальной версии SDK, которую вы поддерживаете).

Настройка проекта игры на Android за 10 простых шагов

Теперь скомбинируем всю полученную к этому моменту информацию и разработаем простой пошаговый метод создания нового игрового проекта для Android в среде Eclipse. Вот что нам нужно от этого проекта.

- Он должен быть способен использовать возможности последней версии SDK, сохраняя при этом совместимость с более старыми версиями, которые до сих пор применяют многие устройства. Это означает, что нам необходима поддержка Android 1.5 и выше.
- Он должен инсталлироваться на карту памяти (при наличии такой возможности), чтобы не заполнять внутреннюю память устройства.
- Он должен быть доступен для отладки.

- Он должен иметь одну главную активность, обрабатывающую все изменения конфигурации самостоятельно, чтобы не пересоздаваться при сдвиге клавиатуры или изменении ориентации устройства.
- Активность должна быть зафиксирована в портретном или ландшафтном режиме.
- Проект должен иметь доступ к SD-карте.
- Он должен быть способен реализовывать защиту от блокировки.

Этих нескольких простых целей нам необходимо достичь. Для этого мы должны предпринять следующие шаги.

1. Создайте новый проект Android в Eclipse, открыв диалог New Android Project (Новый проект Android) (как описано в главе 2).
2. В диалоговом окне New Android Project (Новый проект Android) определите название проекта и установите цель сборки равной последней версии SDK.
3. В том же окне введите название вашей игры, пакета (в котором будут храниться все ваши классы), а также название главной активности. Далее установите минимальную версию SDK равной 3. Нажмите Finish (Готово) для создания проекта.
4. Откройте файл `AndroidManifest.xml`.
5. Чтобы устанавливать игру на карту памяти всегда, когда это возможно, добавьте атрибут `installLocation` в элемент `<manifest>` и установите его значение равным `preferExternal`.
6. Чтобы сделать доступной отладку игры, добавьте атрибут `debuggable` в элемент `<application>` и установите его значение равным `true`.
7. Для фиксации ориентации активности добавьте атрибут `screenOrientation` в элемент `<activity>` и определите его значение (`portrait` или `landscape`).
8. Чтобы сообщить Android о том, что мы хотим обрабатывать изменения конфигурации, вызванные событиями клавиатуры, `keyboardHidden` и изменением ориентации дисплея, установите атрибуту `configChanges` элемента `<activity>` значение `keyboard|keyboardHidden|orientation`.
9. Добавьте два элемента `<uses-permission>` в элемент `<manifest>` и определите атрибуты `name android.permission.WRITE_EXTERNALSTORAGE` и `android.permission.WAKE_LOCK`.
10. Добавьте атрибут `targetSdkVersion` в элемент `<uses-sdk>` и определите вашу целевую версию SDK. Она должна быть той же, что вы задали для целевой сборки в шаге 2.

Вот и все. Десять простых шагов — и в результате мы имеем приложение, устанавливаемое на SD-карту (в случае с Android 2.2 и выше), способное к отладке, обладающее фиксированной ориентацией экрана и не перезагружаемое при изменении конфигурации. Оно имеет доступ к карте памяти, препятствует автоматической блокировке и работает на всех версиях Android начиная с 1.5. Вот итоговый файл `AndroidManifest.xml`, получившийся после осуществления всех шагов:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.badlogic.awesomegame"
      android:versionCode="1"
      android:versionName="1.0"
      android:installLocation="preferExternal">
    <application android:icon="@drawable/icon"
                 android:label="Awesomnium"
                 android:debuggable="true">
        <activity android:name=".GameActivity"
                  android:label="Awesomnium"
                  android:screenOrientation="landscape"
                  android:configChanges="keyboard|keyboardHidden|orientation">
            <intent-filter>
              <action android:name="android.intent.action.MAIN" />
              <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission
           android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <uses-sdk android:minSdkVersion="3" android:targetSdkVersion="9"/>
</manifest>
```

Как видите, я избавился от `@string/app_name` в атрибутах метки элементов `<application>` и `<activity>`. Это необязательно, но мне нравится, когда все определения приложения собраны в одном месте. Теперь пришло время кода! Или...

Назначение значка для вашей игры

Когда вы развернете вашу игру на устройстве и откроете вашу программу для запуска приложений, то увидите, что ей назначен красивый, но отнюдь не уникальный значок Android. Он же будет показываться для вашего приложения и на Android Market. Как изменить его на наш значок?

Еще раз взглянем на элемент `<application>`. Здесь мы определили атрибут `icon`, связанный с изображением, расположенным в каталоге `res/drawable` и имеющим название `icon`. Теперь наши действия должны стать очевидными — нам необходимо заменить файл `icon` в каталоге `drawable` нашим собственным.

Если вы изучите каталог `res/`, то увидите не одну, а несколько папок, начинающихся с `drawable` (рис. 4.1).

Итак, у нас снова классическая проблема курицы и яйца. В главе 2 была только одна папка `res/drawable` для нашего проекта `hello world`. Это произошло из-за того, что мы определили в качестве цели сборки версию SDK 3, которая поддерживает только один размер экрана. Начиная с версии 1.6 (SDK версии 4) положение изменилось. В главе 1 мы видели, что устройства могут иметь разные дисплеи, но не обсуждали, как Android их поддерживает. Оказывается, существует продуманный механизм, позволяющий вам определять ваши графические ресурсы для набора

так называемых плотностей. Плотность экрана — это комбинация физического размера экрана и количества пикселов на нем. Мы изучим данный вопрос подробнее позже, пока же достаточно знать, что Android определяет три плотности: `ldpi` для экранов с низкой плотностью, `mdpi` для стандартной плотности и `hdpi` для дисплеев высокой плотности. Для экранов с малой плотностью мы обычно используем изображения меньшего размера, для более плотных — ресурсы высокого разрешения.

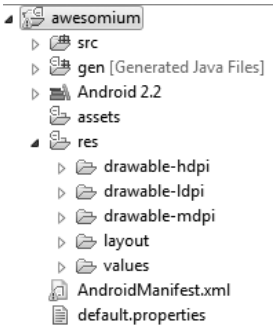

Рис. 4.1. Что случилось с каталогом res/ folder?

Итак, в случае с нашим значком нам необходимо предложить три его версии, по одной для каждой плотности. Но каков должен быть размер этих версий? К счастью, у нас уже есть значки по умолчанию в каждой папке `res/drawable`, из которых мы можем определить необходимые размеры для наших собственных значков. Файл `icon` в `res/drawable-ldpi` имеет размер 36×36 пикселов аналогичный файл в `res/drawable-mdpi` — 48×48 пикселов и, наконец, `icon` в `res/drawable-hdpi` — 72×72 пиксела. Все, что нам нужно, — создать версии нашего значка с соответствующими размерами и заменить файлы `icon.png` в каждом каталоге на наши собственные с тем же именем. Обратите внимание — ссылки на файлы в манифесте чувствительны к регистру. Чтобы избежать потенциальных проблем, всегда используйте нижний регистр.

Чтобы добиться настоящей совместимости с Android 1.5, нам необходимо добавить в проект каталог `res/drawable/` и скопировать в него файл `icon.png` из `res/drawable-mdpi/`. Android 1.5 ничего не знает о других каталогах `drawable`, поэтому может не найти нужный нам файл.

И вот теперь уже точно мы можем заняться написанием кода под Android.

Основы Android API

В оставшейся части главы мы сосредоточимся на работе с теми API для Android, которые необходимы нам для игровых нужд. Для этого мы сделаем тестовый проект, содержащий все наши маленькие примеры для разных используемых нами API. Итак, начнем.

Создание тестового проекта

Из предыдущего раздела вы уже знаете, как создавать проекты. Поэтому для начала выполним те 10 шагов, о которых говорилось ранее. Я так и сделал, создав в итоге проект под названием `ch04-android-basics` с одной главной активностью `AndroidBasicsStarter`. Мы планируем использовать как старые, так и новые API, поэтому я установил параметр, задающий минимальную версию SDK, равным 3 (что соответствует Android 1.5), а целью сборки сделал версию SDK 9 (Android 2.3). Теперь нам необходимо создать реализации активности, каждая из которых демонстрирует различные части Android API.

Однако помните — у нас есть только одна главная активность. На что она будет похожа? Нам нужен удобный способ добавлять новые активности, а также возможность легко их запускать. Должно быть понятно, что главная активность должна каким-то образом позволять нам запускать определенную тестовую активность. Главная активность будет определена в файле манифеста как точка входа в приложение (как было описано ранее). Каждая дополнительная активность, которую мы добавим, будет определяться там же без дочерних элементов `<intent-filter>`. Она будет запускаться программно из главной активности.

Активность AndroidBasicsStarter

Android API предлагает специальный класс `ListActivity`, наследуемый от класса `Activity` (использованного нами в проекте `Hello World`). `ListActivity` — это специальный тип активности. Ее единственное назначение — демонстрировать список элементов (например, строк). Он понадобится нам для отображения названий наших тестовых активностей. При касании элемента списка запускается соответствующая активность. В листинге 4.1 приведен код нашей основной активности `AndroidBasicsStarter`.

Листинг 4.1. `AndroidBasicsStarter.java`, наша главная активность, ответственная за демонстрацию списка и запуск тестовых активностей

```
package com.badlogic.androidgames;

import android.app.ListActivity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;

public class AndroidBasicsStarter extends ListActivity {
    String tests[] = { "LifeCycleTest", "SingleTouchTest", "MultiTouchTest",
            "KeyTest", "AccelerometerTest", "AssetsTest",
            "ExternalStorageTest", "SoundPoolTest", "MediaPlayerTest",
            "FullScreenTest", "RenderViewTest", "ShapeTest", "BitmapTest",
            "FontTest", "SurfaceViewTest" };

    public void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setListAdapter(new ArrayAdapter<String>(this,
                android.R.layout.simple_list_item_1, tests));
    }

    @Override
    protected void onListItemClick(ListView list, View view, int position,
            long id) {
        super.onListItemClick(list, view, position, id);
        String testName = tests[position];
        try {
            Class clazz = Class
                    .forName("com.badlogic.androidgames." + testName);
            Intent intent = new Intent(this, clazz);
            startActivity(intent);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Выбранное мною название пакета — `com.badlogic.androidgames`. Назначение импортируемых элементов должно быть очевидно — это просто все классы, которые мы собираемся использовать в нашем коде.

Наш класс `AndroidBasicsStarter` наследуется от класса `ListActivity` — все пока очевидно. Переменная `tests` представляет собой строковый массив, хранящий названия всех тестовых активностей, которые должно демонстрировать наше приложение. Обратите внимание — названия в этом массиве являются точными названиями классов активностей Java, которые мы позже реализуем.

Следующий кусок кода должен быть вам знаком. Это метод `onCreate()`, который необходимо реализовывать для каждой нашей активности. Данный метод будет вызываться при создании активности. Помните, что первым делом при вызове этого метода для активности мы должны вызывать метод `onCreate()` базового класса, иначе активность не будет отображаться на экране.

После этого нашей следующей задачей является вызов метода `setListAdapter()`, предлагаемого базовым классом `ListActivity`. Он позволит нам определить элементы списка, которые мы хотим отобразить. Эти элементы передаются методу в виде экземпляра класса, реализующего интерфейс `ListAdapter`. Для этого мы используем удобный `ArrayAdapter`. Конструктор этого класса принимает три аргумента: нашу активность, о втором я расскажу чуть позже и массив элементов, которые должна отображать `ListActivity`. Ранее мы уже определили переменную-массив `tests` для хранения этих элементов, поэтому можем использовать ее в качестве третьего аргумента.

Итак, что же это за второй параметр, передаваемый конструктору `ArrayAdapter`? Чтобы объяснить это, я должен рассказать о содержимом API Android для пользовательского интерфейса (который мы, вообще говоря, не будем использовать в этой

книге). Поэтому вместо того, чтобы тратить множество страниц на объяснение того, что нам на самом деле не понадобится, я дам вам простое объяснение: каждый элемент в списке демонстрируется с помощью объекта `View`. Второй аргумент конструктора определяет компоновку каждого объекта `View`, а также его тип. Его значение — `android.R.layout.simple_list_item_1` — является предопределенной константой, предлагаемой API пользовательского интерфейса. Эта константа означает, что мы используем стандартный список для отображения текста. Вообще говоря, `View` — это Android-виджет (как кнопка, текстовое поле или полоса прокрутки). Мы обсуждали этот вопрос в главе 2 при создании активности `HelloWorld`.

Если мы запустим активность методом `onCreate()`, то увидим нечто похожее на рис. 4.2.

Рис. 4.2. Наша тестовая стартовая активность выглядит неплохо, но еще ничего не делает

Теперь настало время сделать так, чтобы при нажатии элемента списка что-то происходило. Нам необходимо запускать соответствующую активность при нажатии пункта списка.

Программный запуск активностей

Класс `ListActivity` содержит защищенный метод `onListItemClick()`, вызываемый при нажатии элемента списка. Нам нужно переопределить этот метод в нашем классе `AndroidBasicsStarter`. Именно это мы и делали в листинге 4.1.

Аргументами этого метода являются объект ListView (который ListActivity использует для отображения элементов), View (содержащийся в ListView элемент, на который производится касание), position (позиция нажатого элемента в списке) и id, который нас не слишком интересует. На самом деле главное для нас — это аргумент position.

Метод onListItemClicked() довольно законопослушен и первым делом вызывает метод базового класса. При переопределении методов активности это всегда является хорошей практикой. Далее мы получаем имя класса из массива tests, основываясь на аргументе position. Это первый элемент головоломки.

Ранее мы говорили о том, что можем запускать активности, определенные в файле манифеста, программно посредством Intent. Класс Intent обладает простым и удобным конструктором, принимающий два аргумента: экземпляр Context и экземпляр Class, представляющий Java-класс активности, которую мы хотим запустить. Context — это интерфейс, предлагающий нам общую информацию о нашем приложении. Он реализуется классом Activity, поэтому мы просто передаем конструктору Intent ссылку this.

Чтобы получить экземпляр Class, представляющий запускаемую нами активность, мы используем небольшое отражение, с которым знакомы те, кто работал с Java. Статический метод Class.forName() принимает строку, содержащую полное имя класса, чей экземпляр мы хотим получить. Все тестовые активности (которые мы реализуем позже) содержатся в пакете com.badlogic.androidgames. Соединяя имя пакета с названием класса, полученного из массива tests, мы получаем полное имя класса запускаемой активности, которое передаем методу Class.forName(), и получаем таким образом хороший экземпляр Class, который можно передать конструктору Intent.

После создания экземпляра Intent мы можем вызывать метод startActivity(), определенный в интерфейсе Context. Поскольку наша активность реализует этот интерфейс, мы просто вызываем эту реализацию. Вот и все!

Как же будет вести себя наше приложение? Сначала будет показана стартовая активность. Каждый раз, когда мы нажимаем элемент списка, запускается соответствующая активность. При этом основная активность переходит в паузу и уходит на задний план. Новая активность будет создана намерением, которые мы послали, и заменит стартовую на экране. При нажатии кнопки «Назад» на телефоне активность будет уничтожена, на экране восстановится стартовая активность.

Создание тестовых активностей

При создании новой тестовой активности нам необходимо выполнить следующие шаги.

1. Создать соответствующий класс Java в пакете и реализовать его логику.
2. Добавить запись о нем в файл манифеста, используя все необходимые атрибуты (например, android:configChanges или android:screenOrientation). Обратите внимание — мы не определяем элемент <intent-filter>, поскольку запускаем активность программно.
3. Добавить имя класса активности в массив tests класса AndroidBasicsStarter.

После осуществления этих шагов обо все остальном будет заботиться логика, реализованная нами в классе `AndroidBasicsStarter`. Новая активность автоматически появится в списке и запустится при нажатии ее названия.

Вопрос, который вас может заинтересовать, — запускается ли каждая тестовая активность в отдельном процессе и с собственной виртуальной машиной? Ответ — нет. Приложение, состоящее из активностей, обладает так называемым стеком активностей. При каждом запуске новой активности она помещается в этот стек. При ее закрытии последняя активность, помещенная в стек, появляется на экране, становясь новой действующей активностью.

Данный механизм имеет свои сложности. Во-первых, все активности приложения (содержащиеся в стеке на паузе и активная на экране) делят одну виртуальную машину, во-вторых — помещаются в одной области памяти. Это может быть как положительным, так и отрицательным моментом. Если активность содержит статические поля, они получат свою область памяти при ее запуске. При этом, будучи статическими, они переживут уничтожение самой активности и последующую сборку мусора. Как вы понимаете, при неосторожном обращении со статическими полями может произойти утечка памяти. Подумайте дважды, прежде чем использовать статические поля.

Однако, как я уже неоднократно замечал, в наших играх мы будем использовать только одну активность. Стартовая активность, созданная нами выше, — исключение из правила, которое делает нашу жизнь легче. Но не беспокойтесь, у нас будет достаточно возможностей нарваться на неприятности даже с одной активностью.

ПРИМЕЧАНИЕ

Все зависит от глубины погружения в программирование пользовательских интерфейсов для Android. Мы всегда будем использовать один объект View в активности для вывода и получения ввода. Если вы хотите изучить такие возможности, как компоновка, группировка представлений, а также другие свистелки-звенелки, предлагаемые библиотекой Android UI, могу предложить вам изучить отличное руководство пользователя на сайте Android Developers.

Жизненный цикл активности

Первый вопрос, который мы должны выяснить при разработке для Android, — как ведет себя наша активность. В случае с ОС Android этот процесс называется жизненным циклом активности и представляет собой описание состояний и переходов между этими состояниями в течение всего времени существования активности. Сначала обсудим теоретические аспекты этой темы.

В теории

Активность может находиться в одном из трех состояний.

- *Работает*. В этом состоянии активность является текущей, занимает экран и напрямую взаимодействует с пользователем.
- *Приостановлена*. Состояние наступает, когда активность все еще видна на экране, но частично закрыта либо другой прозрачной активностью, либо диалогом, либо экран заблокирован. Приостановленная активность может быть уничтожена системой в любой момент (например, при нехватке памяти). Обратите

внимание — экземпляр приостановленной активности не уничтожается; он продолжает храниться в куче VM, ожидая возвращения в работающее состояние.

- *Остановлена*. Состояние наступает, когда активность полностью закрыта другой активностью и поэтому больше не видна на экране. Например, наша активность `AndroidBasicsStarter` перейдет в данное состояние, если мы запустим одну из тестовых активностей. Кроме того, переход в данное состояние наступает при нажатии пользователем кнопки Home (Домой) для перехода на домашний экран ОС. Система также может уничтожить остановленную активность и выгрузить ее из памяти при ее нехватке.

Как видите, и в приостановленном, и в остановленном состояниях система Android может уничтожить активность в любой момент. Это может происходить вежливо, с предварительным информированием активности (вызовом метода `finished()`), или грубо, молчаливым уничтожением процесса.

Активность может быть возвращена в работающее состояние из режимов паузы или остановки. Снова замечу — активность в любом состоянии является одним и тем же Java-экземпляром в памяти, поэтому все ее содержимое остается в том же состоянии, что и перед остановкой или приостановкой.

У активности есть несколько защищенных методов, которые мы можем переопределить для получения информации об изменениях состояния.

- `Activity.onCreate()` — метод вызывается при первом запуске активности. В нем мы устанавливаем компоненты интерфейса и обращаемся к системе ввода. Этот метод вызывается всего один раз в течение жизненного цикла активности.
- `Activity.onRestart()` — метод вызывается при возвращении активности из остановленного состояния. Ему должен предшествовать вызов метода `onStop()`.
- `Activity.onStart()` — метод вызывается после метода `onCreate()` или когда активность возвращается из остановленного состояния. Во втором случае ему предшествует вызов метода `onRestart()`.
- `Activity.onResume()` — метод вызывается после вывода метода `onStart()` или когда активность возвращается из приостановленного состояния (например, при разблокировке экрана).
- `Activity.onPause()` — метод вызывается при переходе активности в приостановленное состояние. Это может оказаться последним полученным нами оповещением, поскольку ОС может решить закрыть наше приложение без уведомления. Поэтому в этом методе нам необходимо сохранить все важные для нас параметры приложения.
- `Activity.onStop()` — метод вызывается при переходе активности в режим остановки. Ему предшествует вызов метода `onPause()` — перед переходом активности в остановленное состояние она всегда сначала проходит режим паузы. Как и в случае с `onPause()`, это может быть последний момент, когда мы получаем информацию от приложения перед его закрытием операционной системой. Здесь мы также можем сохранить текущее состояние. Однако система может решить не вызывать этот метод и просто уничтожить активность. Поэтому, поскольку перед `onStop()` и молчаливым закрытием приложения всегда вызывается `onPause()`, сохранение важной для нас информации лучше всего поручить методу `onPause()`.

- `Activity.onDestroy()` — метод вызывается в конце жизненного цикла активности, когда она окончательно уничтожается. Это последняя точка, в которой мы можем получить и сохранить какие-то данные, которые хотим получить обратно при пересоздании нашей активности. Обратите внимание — этот метод может никогда не вызываться, если активность была уничтожена системой в неявном режиме после вызова `onPause()` или `onStop()`.

Рисунок 4.3 иллюстрирует жизненный цикл активности и порядок вызова методов.

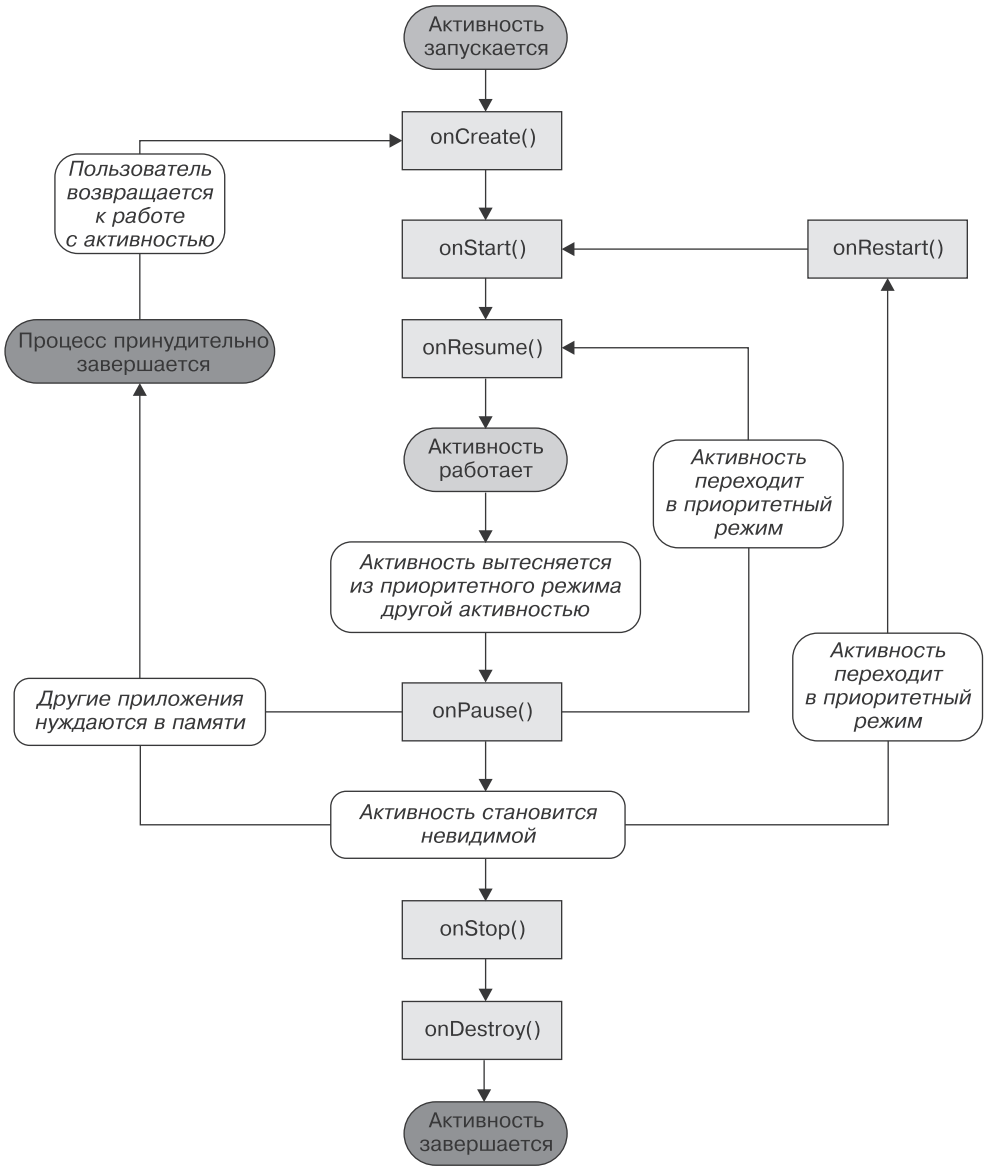

Рис. 4.3. Мощный и запутанный жизненный цикл активности

Итак, из теоретической части нам необходимо извлечь три важных урока.

- Перед переходом нашей активности в работающее состояние всегда вызывается метод `onResume()` вне зависимости от того, возвращаемся мы из приостановленного или остановленного состояния. Поэтому мы спокойно можем игнорировать методы `onRestart()` и `onStart()`. Нам не нужно заботиться о том, из какого состояния возвращается активность. В случае с нашими играми нам следует лишь знать, что сейчас активность запущена, для чего вполне достаточно сигнала от метода `onResume()`.
- Активность может быть уничтожена без оповещения после вызова метода `onPause()`. Поэтому мы никогда не должны полагаться на обязательный вызов методов `onStop()` или `onDestroy()`. Мы также знаем, что `onPause()` всегда предшествует вызову `onStop()`, поэтому мы вполне можем игнорировать методы `onStop()` и `onDestroy()` — нам достаточно будет переопределить метод `onPause()`. В нем мы можем обеспечить сохранение всей важной для нас информации (например, текущего уровня или лучших результатов), записав их на внешний носитель (карту памяти). После вызова `onPause()` прием ставок заканчивается, и мы не будем знать, запустится ли наше приложение.
- Мы знаем, что метод `onDestroy()` может никогда не быть вызван, если система решит уничтожить активность после вызова методов `onStop()` или `onPause()`. Тем не менее нам было бы неплохо знать, готовится ли наша активность к уничтожению. Как же мы узнаем об этом, если `onDestroy()` не будет вызван? Класс `Activity` включает в себя метод `Activity.isFinishing()`, который мы можем вызывать в любое время, чтобы проверить, планируется ли уничтожение нашей активности. Нам гарантирован вызов по крайней мере метода `onPause()` перед уничтожением активности. Все, что нам нужно, — вызвать `isFinishing()` внутри метода `onPause()`, чтобы понять, планирует ли система уничтожение активности после вызова `onPause()`.

Все это несколько упрощает дело. Нам необходимо переопределить лишь следующие методы:

- `onCreate()` — создаем наше окно и визуализируемые компоненты интерфейса, а также получаем пользовательский ввод;
- `onResume()` — стартуем (или возобновляем) наш поток главного цикла (рассмотренный в прошлой главе);
- `onPause()` — ставим на паузу поток основного цикла и, если `Activity.isFinishing()` возвращает `true`, сохраняем на диск важную информацию.

Многих напрягает работа с жизненным циклом активности, но если следовать этим простым правилам, игра будет способна обрабатывать постановку на паузу, возобновление и очистку.

На практике

Напишем наш первый тестовый пример, демонстрирующий жизненный цикл активности. Нам будет нужен некоторый вывод на экран, показывающий изменения состояния. Сделаем это двумя способами.

- Создадим отдельный TextView — компонент интерфейса для отображения в активности. Он нужен для вывода текста (мы уже использовали его неявно для демонстрации элементов списка в нашей стартовой активности). Каждый раз при переходе в новое состояние мы добавляем строку к элементу TextView, который в результате сможет показать нам всю историю изменений состояний приложения.
- Поскольку у нас не будет возможности показать событие разрушения нашей активности в TextView (оно исчезнет с экрана слишком быстро), мы также отследим изменения состояний в LogCat. Это будет реализовано с помощью класса Log, предлагающего набор статических методов для добавления сообщений в LogCat.

Помните, что нам необходимо добавить тестовую активность в наше тестовое приложение. Во-первых, мы определяем ее в файле манифеста в виде элемента <activity>, являющегося дочерним по отношению к <application>:

```
<activity android:label="Life Cycle Test"
          android:name=".LifeCycleTest"
          android:configChanges="keyboard|keyboardHidden|orientation" />
```

Далее мы добавляем новый Java-класс LifeCycleTest в пакет com.badlogic.androidgames. Наконец, нам необходимо добавить название этого класса в массив tests класса AndroidBasicsStarter, определенного нами ранее. Нам придется повторить все эти шаги для всех тестовых активностей, которые мы будем создавать далее. Для краткости я больше не буду о них упоминать. Заметьте также, что я не определил ориентацию экрана для активности LifeCycleTest. В данном примере она может быть как портретной, так и ландшафтной, в зависимости от расположения устройства. Это сделано для того, чтобы вы могли увидеть влияние изменения ориентации на жизненный цикл активности (вообще это зависит от значения атрибута configChanges). Листинг 4.2 демонстрирует код активности.

Листинг 4.2. LifeCycleTest.java, демонстрирующий жизненный цикл активности

```
package com.badlogic.androidgames;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;

public class LifeCycleTest extends Activity {
    StringBuilder builder = new StringBuilder();
    TextView textView;

    private void log(String text) {
        Log.d("LifeCycleTest", text);
        builder.append(text);
        builder.append('\n');
```

```
        textView.setText(builder.toString());
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        textView = new TextView(this);
        textView.setText(builder.toString());
        setContentView(textView);
        log("created");
    }

    @Override
    protected void onResume() {
        super.onResume();
        log("resumed");
    }

    @Override
    protected void onPause() {
        super.onPause();
        log("paused");

        if (isFinishing()) {
            log("finishing");
        }
    }
}
```

Быстро пробежимся по коду. Как и ожидалось, класс наследуется от `Activity`. В нем определены два члена — `StringBuilder`, хранящий все созданные нами сообщения, и `TextView`, который используется для отражения этих сообщений в активности.

Далее мы определяем маленький вспомогательный метод, записывающий текст в журнал **LogCat**, добавляющий текст в `StringBuilder` и обновляющий `TextView`. Для вывода в **LogCat** используется статический метод `Log.d()`, в качестве первого аргумента принимающий тег, а в качестве второго — текст сообщения.

В методе `onCreate()` мы, как всегда, вызываем сначала метод базового класса. Далее мы создаем объект `TextView` и устанавливаем его в качестве просмотрщика содержимого для нашей активности. Этот элемент займет все пространство нашей активности. Наконец, мы записываем созданное сообщение в **LogCat** и обновляем значение текста в `TextView` с помощью ранее определенного метода `log()`.

Затем мы переопределяем метод `onResume()`. Как и со всеми переопределяемыми методами активности, сначала вызываем метод базового класса. После этого все, что мы делаем, — вновь вызываем метод `log` с текстом `resumed` в качестве аргумента.

Переопределенный метод `onPause()` выглядит очень похожим на `onResume()`. Мы записываем в журнал строку `paused` (остановлено). Нам также необходимо знать, собирается ли система уничтожить активность после вызова `onPause()`, поэтому мы вызываем метод `Activity.isFinishing()`. Если он возвращает значение `true`, мы также пишем об этом в журнал. Конечно, в этом случае мы не сможем увидеть обновленный

текст в TextView — активность будет уничтожена до того, как изменение отобразится на экране. Поэтому-то мы и выводим сообщения в журнал LogCat.

Запустите приложение и поиграйтесь немного с тестовой активностью. Вот последовательность действий, которую вы можете выполнить.

1. Запустить тестовую активность из стартовой активности.
2. Заблокировать экран.
3. Разблокировать экран.
4. Нажать кнопку Home (Домой) (которая вернет вас на стартовый экран ОС).
5. В экране Home (Стартовый экран) удерживать кнопку Home (Домой), пока на экране не появится список запущенных приложений. Выберите приложение Android Basics Start для его возобновления (это вернет вас на экран тестовой активности).
6. Нажать кнопку «Назад» (это вернет вас на стартовую активность).

Если система не решит уничтожить вашу активность без уведомления в любой момент, когда она находится в приостановленном состоянии, вы увидите на экране нечто, похожее на рис. 4.4 (если, конечно, еще не вышли из активности кнопкой «Назад»).

Рис. 4.4. Работающая активность LifeCycleTest

При запуске активности вызывается метод `onCreate()`, далее следует `onResume()`. При блокировке дисплея вызывается `onPause()`, при разблокировке — `onResume()`. Нажав кнопку Home (Домой), мы запускаем метод `onPause()`. Возврат к активности вновь влечет за собой вызов `onResume()`. Те же сообщения показаны в LogCat — вы можете увидеть их в представлении LogCat в Eclipse. На рис. 4.5 изображено все, что мы записали в LogCat при выполнении рассмотренной выше последовательности действий (плюс нажатие кнопки «Назад»).

Log (32) LifeCycleTest

Time		pid	tag	Message
11-10 17:03...	D	2243	LifeCycleTest	created
11-10 17:03...	D	2243	LifeCycleTest	resumed
11-10 17:03...	D	2243	LifeCycleTest	paused
11-10 17:03...	D	2243	LifeCycleTest	resumed
11-10 17:03...	D	2243	LifeCycleTest	paused
11-10 17:03...	D	2243	LifeCycleTest	resumed
11-10 17:03...	D	2243	LifeCycleTest	paused
11-10 17:03...	D	2243	LifeCycleTest	finishing

Рис. 4.5. Вывод в LogCat активности LifeCycleTest

Как видите, нажатие кнопки «Назад» активизирует метод `onPause()`. Он также уничтожает активность — проверка в `onPause()` сообщает нам, что это последнее сообщение, которые мы увидим от активности.

И это все о жизненном цикле активности, упрощенном нами для разработки игр. Теперь мы легко можем обрабатывать события постановки на паузу и возврата из нее. А также гарантируем себе оповещение об уничтожении активности.

Поддержка устройств ввода

Как обсуждалось в предыдущих главах, в Android мы получаем информацию от различных устройств ввода. В этом разделе мы обсудим три наиболее важных метода ввода и работу с ними: сенсорный экран, клавиатуру и акселерометр.

Обработка событий касания и множественных касаний

Сенсорный экран — вероятно, самый важный способ получения данных от пользователя. До версии Android 2.0 API поддерживал обработку только одиночных касаний. Мультитач был представлен в Android 2.0 (версия SDK 5). Его обработка была встроена в API для одиночных касаний, что дало неоднозначные результаты с точки зрения удобства. Для начала мы изучим обработку одиночных касаний, доступную во всех версиях Android.

Обработка одиночных касаний. Когда в главе 2 мы обрабатывали нажатия кнопки, то видели, что интерфейсы слушателя являются тем инструментом, с помощью которого Android сообщает нам о наступлении событий. То же самое и с касаниями — эти события передаются реализации интерфейса `OnTouchListener`, зарегистрированного нами с `View`. Интерфейс `OnTouchListener` включает в себя лишь один метод:

```
public abstract boolean onTouch (View v, MotionEvent event)
```

Первый аргумент — объект View, к которому относятся события касания. Второй аргумент — это то, что нам необходимо разобрать для получения этого события.

OnTouchListener может быть зарегистрирован в любой реализации View с помощью метода View.setOnTouchListener() и будет вызываться перед тем, как MotionEvent будет отправлен к View. Мы можем сообщить View в нашей реализации метода onTouch(), что событие касания уже обработано, возвращая из метода true. При возврате значения false объект View сам будет обрабатывать это событие. Экземпляр MotionEvent обладает несколькими методами, нам интересны три из них.

- MotionEvent.getX() и MotionEvent.getY() — возвращают координаты x и y точки касания внутри View. Как вы уже знаете, координаты определяются от левого верхнего угла — слева направо по оси x и сверху вниз по оси y. Значения координат возвращаются в пикселах. При этом тип возвращаемых этими методами значений — float, поэтому координаты имеют субпиксельную точность.
- MotionEvent.getAction() — возвращает тип события касания. Это целочисленное значение, соответствующее одному из элементов списка MotionEvent.ACTION_DOWN, MotionEvent.ACTION_MOVE, MotionEvent.ACTION_CANCEL и MotionEvent.ACTION_UP.

Звучит довольно просто, и это впечатление не обманчиво. Событие MotionEvent.ACTION_DOWN возникает, когда палец касается экрана. При движении пальцем по дисплею возникают события MotionEvent.ACTION_MOVE. Обратите внимание — вы всегда будете получать события MotionEvent.ACTION_MOVE, поскольку не сможете держать палец неподвижно, чтобы их избежать.

При поднятии пальца от экрана срабатывает событие MotionEvent.ACTION_UP.

События MotionEvent.ACTION_CANCEL чуть более загадочны. В документации говорится, что они возникают при отмене текущего жеста. Но в реальной работе я никогда не наблюдал возникновения этого события. Тем не менее мы будем обрабатывать и его (представив, что это событие MotionEvent.ACTION_UP) при создании нашей первой игры.

Создадим простую тестовую активность и посмотрим, как это работает в коде. Активность должна показывать текущую позицию пальца на дисплее, а также тип наступившего события. Листинг 4.3 демонстрирует, что я имею в виду.

Листинг 4.3. SingleTouchTest.java; тестирование обработки одиночных касаний

```
package com.badlogic.androidgames;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.MotionEvent;
import android.view.View;
import android.view.View.OnTouchListener;
import android.widget.TextView;

public class SingleTouchTest extends Activity implements OnTouchListener {
    StringBuilder builder = new StringBuilder();
    TextView textView;

    public void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        textView = new TextView(this);
        textView.setText("Touch and drag (one finger only)!");
        textView.setOnTouchListener(this);
        setContentView(textView);
    }

    @Override
    public boolean onTouch(View v, MotionEvent event) {
        builder.setLength(0);
        switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            builder.append("down, ");
            break;
        case MotionEvent.ACTION_MOVE:
            builder.append("move, ");
            break;
        case MotionEvent.ACTION_CANCEL:
            builder.append("cancel, ");
            break;
        case MotionEvent.ACTION_UP:
            builder.append("up, ");
            break;
        }
        builder.append(event.getX());
        builder.append(", ");
        builder.append(event.getY());
        String text = builder.toString();
        Log.d("TouchTest", text);
        textView.setText(text);
        return true;
    }
}
```

Наша активность реализует интерфейс `OnTouchListener`. В ней также есть два члена, один для `TextView`, второй — `StringBuilder`, который мы будем использовать для создания строк для событий.

Метод `onCreate()` не требует длинных пояснений. Единственное новшество — вызов `TextView.setOnTouchListener()`, в котором мы регистрируем нашу активность в `TextView`, чтобы она могла получать `MotionEvents`.

Все, что осталось, — собственно реализация метода `onTouch()`. Мы игнорируем аргумент типа `View`, потому что знаем, что это наш объект `TextView`. Нас интересует получение типа события, добавление его к объекту `StringBuilder`, получение координат и обновление текста `TextView`. Вот и все. Кроме того, мы записываем сообщения в **LogCat**, чтобы видеть порядок появления событий — ведь в `TextView` мы увидим только последнее обработанное событие (`StringBuilder` очищается при каждом вызове `onTouch()`).

Последняя небольшая подробность о методе `onTouch` — выражение `return`, в котором мы возвращаем `true`. Обычно мы придерживаемся концепции слушателя

и возвращаем `false`, чтобы не связываться лишний раз с процессом обработки событий. Если мы сделаем так же в нашем примере, то не увидим никаких событий, кроме `MotionEvent.ACTION_DOWN`. Поэтому мы сообщаем `TextView`, что мы только что использовали событие. В разных реализациях `View` поведение может различаться. К счастью, в нашей книге будут применяться еще всего три вида `View`, и мы сможем использовать любое событие по нашему желанию.

Если мы запустим это приложение на эмуляторе или подключенном устройстве, то увидим, что `TextView` всегда показывает тип последнего события и координаты, вычисленные в методе `onTouch()`. Кроме того, все эти сообщения вы можете увидеть в LogCat.

Я не определял явно ориентацию активности в файле манифеста. Если вы повернете устройство так, чтобы активность перешла в ландшафтный режим, то координатная система, конечно, изменится. Рисунок 4.6 показывает активность в портретном и ландшафтном режимах. В обоих случаях я постарался касаться экрана в середине. Обратите внимание — координаты x и y как будто поменялись местами. На рисунок также добавлены оси и точка касания. В обоих случаях координаты начинаются в левом верхнем углу `TextView`, ось x движется вправо, ось y — вниз.

Рис. 4.6. Касание экрана в портретном *(слева)* и ландшафтном *(справа)* режимах

В зависимости от ориентации меняются максимально возможные значения x и y. Для примера мы использовали Nexus One (разрешение 480×800 пикселов в портретном режиме и 800×480 в ландшафтном). Поскольку координаты касания даны относительно `View` (которое занимает не весь экран устройства), максимальное значение координаты x будет всегда меньше разрешения по высоте. Позже вы узнаете, как включить полноэкранный режим, чтобы убрать строку состояния и заголовок.

К сожалению, существуют некоторые проблемы с событиями касания на старых версиях Android и устройствах первого поколения.

- *Лавина событий касания.* Драйвер сообщает о максимально возможном количестве событий, когда палец находится на экране, — на некоторых устройствах оно измеряется сотнями за секунду. Мы можем частично справиться с этой проблемой, поместив вызов `Thread.sleep(16)` в метод `onTouch()` — после каждого события поток обработки будет приостанавливаться на 16 миллисекунд. В этом случае мы получим максимально 60 событий в секунду, чего более чем достаточно для нормального игрового процесса. Эта проблема касается только устройств на базе Android 1.5.
- *Касание экрана загружает процессор.* Даже включив паузу в методе `onTouch()`, мы должны понимать, что система обрабатывает процессы в своем ядре. На старых устройствах (например, Hero или G1) эти действия могут захватывать до 50 % мощности процессора, что оставляет меньше пространства для потока главного цикла. Как следствие, падает частота кадров — иногда до такой степени, когда игровой процесс становится невозможным. На устройствах второго поколения проблема встречается намного реже, и о ней можно забыть. К сожалению, для старых аппаратов какого-то универсального решения не существует.

В общем, для уверенности стоит помещать вызов `Thread.sleep(16)` во все ваши методы `onTouch()`. На новых устройствах это не даст никакого эффекта; на более древних это по крайней мере убережет вас от лавины сообщений о касаниях.

Учитывая, что аппараты первого поколения все же постепенно уходят с рынка, проблема становится менее значимой с течением времени. Однако она по-прежнему может огорчать разработчиков игр. Попытайтесь оправдаться перед пользователями тем, что ваша игра работает как черепаха потому, что какой-то драйвер расходует все ресурсы процессора. Результат вполне предсказуем — пользователей это не будет волновать.

Обработка множественных касаний. Это были цветочки — пришло время главной головной боли. API для работы с мультитач было встроено в класс `MotionEvent`, изначально предназначенный для обработки отдельных касаний. Поэтому при попытке обработки множественных касаний возникают некоторые недоразумения. Попробуем их разрешить.

ПРИМЕЧАНИЕ

API для мультитач, похоже, вызывает некоторые вопросы даже у инженеров Android, создавших его. Он был сильно модернизирован в SDK версии 8 (Android 2.2) — новые методы, новые и даже переименованные константы. Эти изменения должны сделать работу с множественными касаниями немного проще, но доступны они только начиная с SDK версии 8. Для поддержки всех версий Android с мультитач (начиная с 2.0) придется использовать SDK версии 5.

Обработка множественных касаний довольно сильно похожа на обработку событий однократного касания. Мы реализуем тот же интерфейс `OnTouchListener`, что и для однократных касаний. Нам также необходим экземпляр `MotionEvent`, из кото-

рого будут считываться данные. Мы будем обрабатывать те же события, что и раньше (например, MotionEvent.ACTION_UP), и добавим к ним еще несколько.

Указатели ID и индексы. Различия начинаются, когда нам необходим доступ к координатам касания. MotionEvent.getX() и MotionEvent.getY() возвращают координаты одного пальца на экране. При обработке событий множественных касаний нам необходимо использовать перегруженные версии этих методов для получения так называемого номера указателя. Это должно выглядеть так:

```
event.getX(pointerIndex);
event.getY(pointerIndex);
```

Теперь мы можем ожидать, что pointerIndex напрямую связан с одним из пальцев, касающихся экрана (например, первый палец на экране получает pointerIndex, равный 0, следующий — 1 и т. д.). К сожалению, это не так.

pointerIndex — номер во внутренних массивах MotionEvent, хранящих координаты события для определенного пальца, который касается дисплея. Реальный идентификатор пальца называется идентификатором указателя. Существует отдельный метод MotionEvent.getPointerIdentifier(int pointerIndex), возвращающий идентификатор указателя, базирующийся на номере указателя. Идентификатор указателя для пальца не изменится, пока тот не оторвется от экрана (но это не обязательно верно для номера указателя).

Рассмотрим, как мы можем получить номер указателя для события. Пока проигнорируем тип события.

```
int pointerIndex = (event.getAction() & MotionEvent.ACTION_POINTER_ID_MASK) >>
MotionEvent.ACTION_POINTER_ID_SHIFT;
```

Вы, видимо, подумали сейчас то же, что подумал я, когда впервые написал этот код. Однако прежде, чем окончательно потерять веру в человечество, попробуем расшифровать, что тут происходит. Мы получаем тип события из MotionEvent через MotionEvent.getAction(). Отлично, это мы уже освоили. Далее мы выполняем битовую операцию AND, используя полученное от метода MotionEvent.getAction() значение и константу MotionEvent.ACTION_POINTER_ID_MASK. Теперь начинается самое веселое.

Константа имеет значение 0xff00, поэтому мы, по существу, делаем все биты равными 0, кроме битов с 8 по 15, хранящих номер указателя события. Нижние 8 бит числа, возвращенного методом event.getAction(), хранят значение типа события (например, MotionEvent.ACTION_DOWN и т. д.). Этой битовой операцией мы, проще говоря, стираем данные о типе события.

Теперь сдвиг приобретает больший смысл. Мы осуществляем его с помощью константы MotionEvent.ACTION_POINTER_ID_SHIFT (равной 8), то есть перемещаем биты с 8 по 15 в биты с 0 по 7, получая актуальный номер указателя на событие. Обратите внимание — наши волшебные константы называются XXX_POINTER_ID_XXX, а не XXX_POINTER_INDEX_XXX (что имело бы больше смысла — ведь нам необходим номер указателя, а не его идентификатор). Что ж, инженеры Android тоже ошибаются. В SDK версии 8 они убрали эти константы и представили взамен новые,

названные XXX_POINTER_INDEX_XXX и имеющие те же значения, что и убранные. Для обеспечения работы старых приложений, написанных на SDK 5, старые константы тоже сохранены.

Итак, мы знаем, как получить этот загадочный номер указателя, с которым мы можем запросить координаты и идентификатор указателя события.

Маска операции и другие типы событий. Далее нам необходимо получить настоящий тип события минус дополнительный номер указателя, закодированный в числе, возвращенном MotionEvent.getAction(). Нам лишь необходимо исключить номер указателя:

```
int action = event.getAction() & MotionEvent.ACTION_MASK;
```

Да, это было просто. Увы, вы поймете это, только если знаете, что это номер указателя и что он зашифрован в действии.

Осталось декодировать тип события, как мы делали это ранее. Я уже упоминал, что у нас появились новые типы событий, поэтому рассмотрим их.

- MotionEvent.ACTION_POINTER_DOWN — возникает для каждого пальца (начиная со второго), касающегося экрана. Первый палец все еще вызывает событие MotionEvent.ACTION_DOWN.
- MotionEvent.ACTION_POINTER_UP — аналог предыдущего действия. Событие возникает, когда палец отрывается от экрана и при этом экрана касаются более одного пальца. Последний палец при отрыве вызывает событие MotionEvent.ACTION_UP. Это не обязательно будет первый палец, коснувшийся экрана.

К счастью, мы можем представить, что два этих новых типа событий ничем не отличаются от старых добрых MotionEvent.ACTION_UP и MotionEvent.ACTION_DOWN.

Последнее отличие — тот факт, что один MotionEvent может обладать данными для множества событий. Да, вы все верно прочитали. По этой причине соединенные вместе события должны иметь одинаковый тип. В реальности это происходит только для события MotionEvent.ACTION_MOVE, и нам приходится мириться с этим фактом. Чтобы проверить, сколько событий содержится в одном MotionEvent, мы используем метод MotionEvent.getPointerCount(), сообщающий нам о количестве пальцев, касающихся нашего дисплея. Далее мы можем получить идентификатор указателя и координаты для номеров указателя от 0 до MotionEvent.getPointerCount() – 1 с помощью методов MotionEvent.getX(), MotionEvent.getY() и MotionEvent.getPointerId().

На практике

Напишем пример для данного API. Нам хотелось отслеживать в нем касания всех 10 пальцев (на данный момент не существует устройств, способных отслеживать большее их количество, так что нам ничего не грозит). Android присвоит этим пальцам идентификаторы указателей от 0 до 9 в том порядке, в котором они касаются дисплея. Итак, мы будем хранить координаты каждого идентификатора и его состояние (касается или нет), а также выводить эту информацию на экран с помощью TextView. Назовем нашу тестовую активность MultiTouchTest. Полный код показан в листинге 4.4.

Листинг 4.4. MultiTouchTest.java: тестируем Multitouch API

```
package com.badlogic.androidgames;

import android.app.Activity;
import android.os.Bundle;
import android.view.MotionEvent;
import android.view.View;
import android.view.View.OnTouchListener;
import android.widget.TextView;

public class MultiTouchTest extends Activity implements OnTouchListener {
    StringBuilder builder = new StringBuilder();
    TextView textView;
    float[] x = new float[10];
    float[] y = new float[10];
    boolean[] touched = new boolean[10];

    private void updateTextView() {
        builder.setLength(0);
        for(int i = 0; i < 10; i++) {
            builder.append(touched[i]);
            builder.append(", ");
            builder.append(x[i]);
            builder.append(", ");
            builder.append(y[i]);
            builder.append("\n");
        }
        textView.setText(builder.toString());
    }

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        textView = new TextView(this);
        textView.setText("Touch and drag (multiple fingers supported)!");
        textView.setOnTouchListener(this);
        setContentView(textView);
    }

    @Override
    public boolean onTouch(View v, MotionEvent event) {
        int action = event.getAction() & MotionEvent.ACTION_MASK;
        int pointerIndex = (event.getAction() &
                            MotionEvent.ACTION_POINTER_ID_MASK) >>
                            MotionEvent.ACTION_POINTER_ID_SHIFT;
        int pointerId = event.getPointerId(pointerIndex);

        switch (action) {
        case MotionEvent.ACTION_DOWN:
        case MotionEvent.ACTION_POINTER_DOWN:
            touched[pointerId] = true;
            x[pointerId] = (int)event.getX(pointerIndex);
```

```
            y[pointerId] = (int)event.getY(pointerIndex);
            break;

        case MotionEvent.ACTION_UP:
        case MotionEvent.ACTION_POINTER_UP:
        case MotionEvent.ACTION_CANCEL:
            touched[pointerId] = false;
            x[pointerId] = (int)event.getX(pointerIndex);
            y[pointerId] = (int)event.getY(pointerIndex);
            break;

        case MotionEvent.ACTION_MOVE:
            int pointerCount = event.getPointerCount();
            for (int i = 0; i < pointerCount; i++) {
                pointerIndex = i;
                pointerId = event.getPointerId(pointerIndex);
                x[pointerId] = (int)event.getX(pointerIndex);
                y[pointerId] = (int)event.getY(pointerIndex);
            }
            break;
        }

        updateTextView();
        return true;
    }
}
```

Мы реализовали интерфейс `OnTouchListener`, как и раньше. Для отслеживания координат и состояния касания 10 пальцев мы добавили три члена-массива, в которых будет храниться нужная нам информация. Массивы `x` и `y` хранят координаты каждого ID указателя, массив `touched` содержит данные о том, касается ли данный ID экрана или нет.

Далее я создал небольшой вспомогательный метод, выводящий в `TextView` текущее состояние пальцев. Он просто пробегает по всем 10 состояниям пальцев и склеивает их в одну строку с помощью `StringBuilder`. Конечный текст выводится в `TextView`.

Метод `onCreate()` устанавливает нашу активность и регистрирует ее в качестве `OnTouchListener` в `TextView`. Эта часть кода вам уже должна быть знакома.

Теперь самая тяжелая часть — метод `onTouch()`. Начинаем с получения типа события на основе целого числа, возвращенного методом `event.getAction()`. Далее получаем номер указателя и соответствующий ему идентификатор из `MotionEvent`, как уже обсуждалось ранее. Ядро метода `onTouch()` — большое путаное выражение `switch`, которое мы уже использовали в усеченном виде при обработке одиночных касаний. Мы группируем события в три высокоуровневые категории.

- Произошло касание экрана (`MotionEvent.ACTION_DOWN`, `MotionEvent.ACTION_PONTER_DOWN`). Для таких идентификаторов указателей мы устанавливаем состояние касания в `true`, а также сохраняем текущие координаты этого указателя.

- Произошел отрыв указателя от экрана (`MotionEvent.ACTION_UP`, `MotionEvent.ACTION_POINTER_UP`, `MotionEvent.CANCEL`). Состояние касания для таких идентификаторов устанавливается в `false`, сохраняются их последние известные координаты.
- Один или несколько пальцев двигаются по экрану (`MotionEvent.ACTION_MOVE`). Мы проверяем количество событий, находящихся в данный момент в `MotionEvent`, и затем обновляем координаты для номеров указателей от 0 до `MotionEvent.getPointerCount() – 1`. Для каждого события мы получаем соответствующий идентификатор указателя и обновляем его координаты.

При выполнении события мы обновляем `TextView` с помощью вызова метода `updateView()`, определенного нами ранее. Наконец, возвращаем значение `true` — знак того, что мы обработали событие касания.

Рисунок 4.7 демонстрирует вывод активности после того, как я коснулся экрана моего Nexus One двумя пальцами, а потом немного подвигал ими.

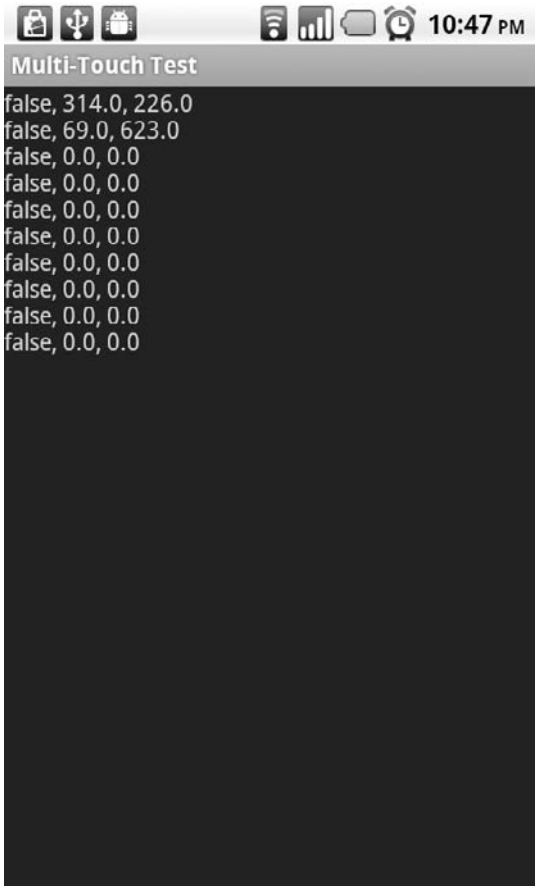

Рис. 4.7. Результат использования мультитач

Осталось несколько вопросов, которые стоит обсудить при запуске этого примера.

- Если мы запустим его на эмуляторе с версией Android ниже 2.0, то получим ужасное исключение, поскольку мы используем отсутствующее в старых версиях API. С этим можно бороться, определяя в процессе работы приложения текущую версию Android и применяя в случае с Android 1.5 и 1.6 API для обработки одиночных касаний, а в случае с Android 2.0 и позже — мультитач API. Мы вернемся к этой теме в следующей главе.
- На эмуляторе нет поддержки мультитач. Мы можем создать его с использованием версии Android 2.0 и выше, но это ничего не даст — мышь-то у нас всего одна. И даже если бы их было две, это ничего бы не изменило.
- Коснитесь двумя пальцами дисплея, поднимите первый и снова коснитесь им экрана. Второй палец сохранит свой идентификатор касания после того, как первый палец поднимается. Когда первый палец вновь касается дисплея, он получает первый свободный идентификатор указателя (в данном случае 0). Каждый новый палец, касающийся экрана, получает первый незанятый идентификатор. Это правило, которое стоит запомнить.
- Если вы протестируете данный пример на Nexus One или Droid, то можете обратить внимание на несколько странное поведение при пересечении двух пальцев на одной оси. Это происходит из-за того, что экраны данных устройств не поддерживают в полной мере отслеживание отдельных пальцев. Это на самом деле большая проблема, но мы с ней справимся, разрабатывая наш интерфейс с некоторой оглядкой. Еще раз мы обратим внимание на этот казус в одной из последующих глав. Пока просто держите это в голове.

Итак, вот таким образом осуществляется обработка множественных касаний на Android. Сложно, конечно, но, разобравшись один раз и примирившись с некоторой неуклюжестью реализации, этот функционал вполне можно использовать.

ПРИМЕЧАНИЕ

Простите, что взорвал вам мозг. Этот пункт книги был действительно трудным. К сожалению, официальная документация для этого API невероятно лаконична, и большинство людей «изучают» эти возможности, кромсая API на практике. Я предлагаю вам поработать с вышеприведенным кодом, пока вы не будете понимать точно все, что в нем делается.

Обработка событий клавиатуры

После безумия прошлого пункта стоит сделать передышку и заняться чем-нибудь попроще. Добро пожаловать в мир обработки нажатий клавиш!

Для отслеживания событий клавиш мы реализуем другой интерфейс, названный `OnKeyListener`. У него есть лишь один метод `onKey()` со следующей сигнатурой:

```
public boolean onKey(View view, int keyCode, KeyEvent event)
```

`View` определяет вид, получающий событие нажатия клавиши; аргумент `keyCode` — одна из констант, определенных в классе `KeyEvent`; последний аргумент — собственно код клавиши с дополнительной информацией.

Что такое код клавиши? Каждая клавиша экранной (или физической) клавиатуры и все системные кнопки имеют присвоенный им уникальный номер. Эти коды клавиш определены в классе KeyEvent как статические, открытые, финальные, целочисленные переменные. Например, код клавиши KeyEvent.KEYCODE_A назначен клавише A. Тут все ясно — при нажатии клавиши на клавиатуре в текстовом поле появляется символ, имеющий определенный код.

Класс KeyEvent похож на класс MotionEvent. У него есть два необходимых нам метода.

- KeyEvent.getAction() — возвращает значения KeyEvent.ACTION_DOWN, KeyEvent.ACTION_UP и KeyEvent.ACTION_MULTIPLE. Для наших целей мы можем игнорировать последнее из указанных типов событий. Другие два события генерируются, когда клавиша нажата и отпущена соответственно.
- KeyEvent.getUnicodeChar() — возвращает символ в кодировке Юникод, который появляется в текстовом поле. Это может быть похоже на событие с кодом клавиши KeyEvent.KEYCODE_A, но в Юникод. Мы можем использовать данный метод, если хотим осуществлять текстовый ввод самостоятельно.

Для получения событий от клавиатуры на View должен быть установлен фокус. Это можно сделать принудительно, вызвав следующие методы:

```
View.setFocusableInTouchMode(true);
View.requestFocus();
```

Первый метод гарантирует, что на View может быть установлен фокус. Второй запрашивает получение фокуса определенным представлением.

Создадим небольшую тестовую активность, чтобы понаблюдать за комбинацией этих методов. Нам необходимо будет получать события от клавиш и показывать последнее полученное нами в TextView. В качестве информации будет выводиться тип клавиатурного события, а также код клавиши и ее Юникод-символ (если он будет сгенерирован) Обратите внимание — некоторые клавиши не создают символы Юникод сами по себе, а только в комбинации с другими символами. Листинг 4.5 показывает способ достижения всех этих целей в нескольких строках кода.

Листинг 4.5. KeyTest.Java; тестирование API обработки клавиатурных событий

```
package com.badlogic.androidgames;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.KeyEvent;
import android.view.View;
import android.view.View.OnKeyListener;
import android.widget.TextView;

public class KeyTest extends Activity implements OnKeyListener {
    StringBuilder builder = new StringBuilder();
    TextView textView;

    public void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        textView = new TextView(this);
        textView.setText("Press keys (if you have some)!");
        textView.setOnKeyListener(this);
        textView.setFocusableInTouchMode(true);
        textView.requestFocus();
        setContentView(textView);
    }

    @Override
    public boolean onKey(View view, int keyCode, KeyEvent event) {
        builder.setLength(0);
        switch (event.getAction()) {
        case KeyEvent.ACTION_DOWN:
            builder.append("down, ");
            break;
        case KeyEvent.ACTION_UP:
            builder.append("up, ");
            break;
        }
        builder.append(event.getKeyCode());
        builder.append(", ");
        builder.append((char) event.getUnicodeChar());
        String text = builder.toString();
        Log.d("KeyTest", text);
        textView.setText(text);

        if (event.getKeyCode() == KeyEvent.KEYCODE_BACK)
            return false;
        else
            return true;
    }
}
```

Начинаем с декларации реализации нашей активностью интерфейса `OnKeyListener`. Далее определяем два члена, с которыми мы уже знакомы: `StringBuilder` для создания строки, выводимой в `TextView`, и самого `TextView` для показа текста.

В методе `onCreate()` удостоверяемся в том, что фокус установлен в `TextView` (иначе он не сможет получать события клавиатуры). Кроме того, регистрируем активность в качестве `OnKeyListener` методом `TextView.setOnKeyListener()`.

Метод `onKey()` довольно прямолинеен. Мы обрабатываем два типа событий в операторе `switch`, добавляя соответствующую строку в `StringBuilder`. После этого мы добавляем код клавиши (а также Юникод-символ) из `KeyEvent` и выводим его в `LogCat` так же, как и в `TextView`.

Последнее выражение `if` весьма интересно: если нажата клавиша `back`, мы возвращаем `false` из метода `onKey()`, заставляя `TextView` обрабатывать событие. В другом случае мы возвращаем `true`. Для чего нужно это разделение?

Вернув `true` при нажатии клавиши `back`, мы слегка вмешаемся в жизненный цикл активности. Она не будет закрыта, поскольку мы взяли на себя обработку клавиши

back. Конечно, существуют сценарии, в которых нам как раз необходимо перехватывать нажатие back, чтобы активность не закрывалась. Однако я настоятельно рекомендую не делать этого без необходимости.

Рисунок 4.8 иллюстрирует вывод активности при удерживании клавиш Shift+A на клавиатуре моего Droid.

Рис. 4.8. Одновременное нажатие Shift+A

Стоит отметить еще пару вещей.

- Если вы посмотрите на вывод LogCat, то заметите, что обработка одновременного нажатия нескольких клавиш не составляет никаких проблем.
- Нажатие джойстика и повороты трекбола воспринимаются как клавиатурные события.
- Как и в случае с событиями касания, события обработки клавиатуры отъедают немалый процент мощностей процессора на старых версиях Android и устройствах первого поколения.

Это было довольно расслабляющее чтиво по сравнению с предыдущим пунктом, не так ли?

ПРИМЕЧАНИЕ

На самом деле API по обработке клавиатуры немного сложнее, чем я вам рассказал. Но для наших игровых объектов полученной в этом разделе информации более чем достаточно. Если вам хочется чего-нибудь посложнее, обратитесь к официальной документации на сайте Android Developers.

Чтение состояния акселерометра. Акселерометр — очень интересный способ ввода для игр. Все устройства Android должны оснащаться трехосевым акселерометром. Мы немного говорили об этой функции в прошлой главе. Все, что нам будет нужно для ее реализации, — получить текущее состояние акселерометра.

Как же получить эту информацию? Вы угадали — регистрируя слушателя. Интерфейс, который нам необходимо реализовать, называется `SensorEventListener`. Он заключает в себе два метода:

```
public void onSensorChanged(SensorEvent event);
public void onAccuracyChanged(Sensor sensor, int accuracy);
```

Первый метод вызывается при возникновении события акселерометра. Второй срабатывает при изменении точности акселерометра (для наших целей мы спокойно можем его игнорировать).

Как же мы зарегистрируем `SensorEventListener`? Для этого сначала необходимо проделать некоторую предварительную работу. Нужно проверить, имеется ли вообще в аппарате акселерометр. Только что я говорил вам, что все устройства на Android должны им оснащаться. Это все еще соответствует истине, но вдруг в будущем что-то изменится? Нам необходимо быть на 100 % уверенными, что этот метод ввода нам доступен.

Первое, что нам необходимо сделать, — получить экземпляр так называемого `SensorManager`. Эта штука расскажет нам, установлен ли акселерометр, а еще именно в нем мы зарегистрируем наш слушатель. Для получения `SensorManager` мы используем метод интерфейса `Context`:

```
SensorManager manager =
(SensorManager)context.getSystemService(Context.SENSOR_SERVICE);
```

`SensorManager` — это так называемый системный сервис, предоставляемый системой. Android состоит из множества системных сервисов, каждый из которых предлагает различные сегменты информации каждому, кто об этом вежливо попросит.

Теперь, когда у нас есть сервис, мы можем проверить доступность акселерометра:

```
boolean hasAccel = manager.getSensorList(Sensor.TYPE_ACCELEROMETER).size() > 0;
```

Этот кусочек кода запрашивает у менеджера все установленные датчики типа «акселерометр». Предполагается, что в устройстве может быть несколько акселерометров, хотя в реальности метод всегда возвращает одну запись.

Если акселерометр установлен, мы можем получить его данные от `SensorManager` и зарегистрировать наш `SensorEventListener`:

```
Sensor sensor = manager.getSensorList(Sensor.TYPE_ACCELEROMETER).get(0);
boolean success = manager.registerListener(listener, sensor,
SensorManager.SENSOR_DELAY_GAME);
```

Аргумент `SensorManager.SENSOR_DELAY_GAME` определяет частоту получения состояния акселерометра. Это специальная константа, разработанная для игр, поэтому было бы глупо ее не использовать. Обратите внимание — метод `SensorManager.registerListener()` возвращает значение `boolean`, по которому можно судить, успеш-

но ли завершился процесс регистрации. Это значит, что нам необходимо проверять это значение для уверенности в том, что мы получаем события от акселерометра.

После регистрации слушателя мы можем получать события SensorEvents в методе SensorEventListener.onSensorChanged(). Имя этого метода[1] намекает, что он вызывается только при изменении состояния датчика. Это может смутить — ведь состояние акселерометра изменяется постоянно. При регистрации слушателя мы определяем частоту, с которой хотим получать обновления состояния датчика.

Как же мы обрабатываем SensorEvent? Довольно просто. SensorEvent включает в себя открытый массив значений типа float, названный SensorEvent.values. В нем хранятся текущие значения для каждой из трех осей акселерометра. SensorEvent.values[0] содержит значение по оси *x*, SensorEvent.values[1] — по оси *y* и, наконец, SensorEvent.values[2] — по оси *z*. Смысл этих значений мы обсуждали в главе 3 (если забыли, вернитесь к ее подразделу «Ввод» раздела «Код: скучная рутина»).

Вооружившись всей этой информацией, мы можем приступить к созданию простой тестовой активности. Все, что нам нужно, — выводить значения акселерометра по каждой оси в элементе TextView. В листинге 4.6 написано, как это делается.

Листинг 4.6. AccelerometerTest.java: тестирование API акселерометра

```
package com.badlogic.androidgames;

import android.app.Activity;
import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.widget.TextView;

public class AccelerometerTest extends Activity implements
  SensorEventListener {
    TextView textView;
    StringBuilder builder = new StringBuilder();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        textView = new TextView(this);
        setContentView(textView);

        SensorManager manager = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
        if (manager.getSensorList(Sensor.TYPE_ACCELEROMETER).size() == 0) {
            textView.setText("No accelerometer installed");
```

[1] Примерный перевод с англ.: «Слушатель событий сенсора, срабатывающий при возникновении изменения». — *Примеч. ред.*

```
        } else {
            Sensor accelerometer = manager.getSensorList(
                    Sensor.TYPE_ACCELEROMETER).get(0);
            if (!manager.registerListener(this, accelerometer,
                    SensorManager.SENSOR_DELAY_GAME)) {
                textView.setText("Couldn't register sensor listener");
            }
        }
    }
    @Override
    public void onSensorChanged(SensorEvent event) {
        builder.setLength(0);
        builder.append("x: ");
        builder.append(event.values[0]);
        builder.append(", y: ");
        builder.append(event.values[1]);
        builder.append(", z: ");
        builder.append(event.values[2]);
        textView.setText(builder.toString());
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // тут ничего не делаем
    }
}
```

Мы начали с проверки доступности датчика акселерометра. После успешного тестирования получаем датчик от `SensorManager` и пытаемся зарегистрировать нашу активность, реализующую интерфейс `SensorEventListener`. Если какая-то из этих операций заканчивается неудачей, в `TextView` отображается соответствующее сообщение.

Метод `onSensorChanged()` просто считывает значения координат по разным осям от `SensorEvent` и обновляет соответствующим образом содержимое `TextView`.

Метод `onAccuracyChanged()` присутствует здесь только потому, что входит в реализуемый нами интерфейс `SensorEventListener`. Других реальных целей у него здесь нет.

На рис. 4.9 показано, какие значения осей принимает акселерометр в портретном и ландшафтном режимах (телефон находится перпендикулярно земле).

Два финальных замечания по акселерометру.

- Как вы видите в правой части рис. 4.9, значения акселерометра иногда могут выходить за рамки определенного для них диапазона. Это происходит из-за небольших неточностей в работе сенсора; вам необходимо будет учитывать это, чтобы полученные вами значения были как можно более точными.
- Значения координат осей акселерометра всегда выводятся в одном и том же порядке вне зависимости от текущей ориентации устройства.

Вот теперь мы обсудили все необходимые для разработки игр классы Android API, связанные с обработкой ввода.

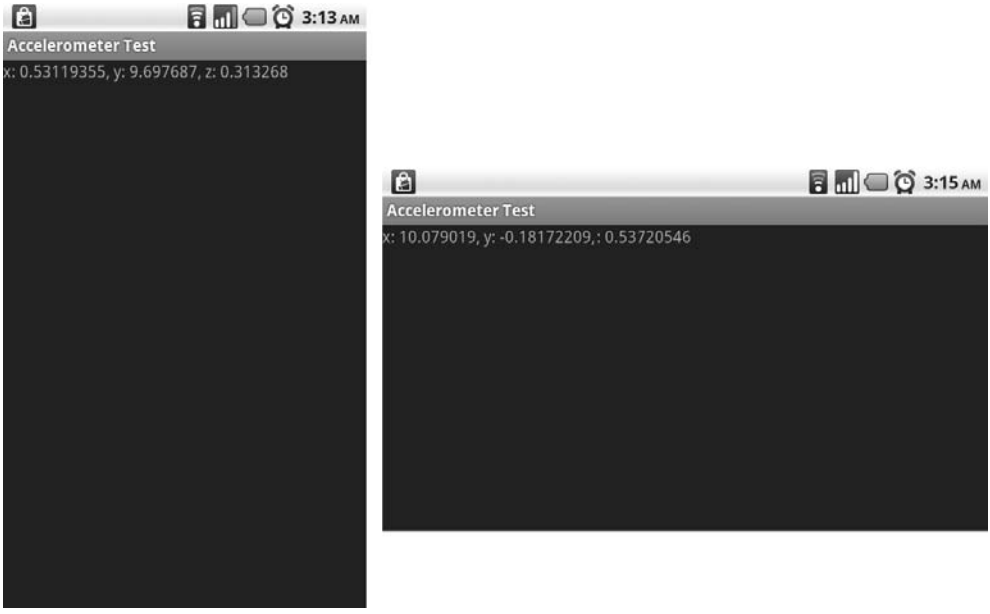

Рис. 4.9. Значения осей акселерометра в портретном *(слева)* и ландшафтном *(справа)* состоянии

ПРИМЕЧАНИЕ

Как понятно из названия, класс SensorManager[1] может предоставить доступ и к другим датчикам (например, компасу и сенсору освещенности). Если вы хотите придумать что-то новое, то можете попробовать написать игру, использующую их, — обработка таких событий весьма похожа на работу с акселерометром. Документация на сайте Android Developers даст вам больше информации.

Поддержка файловой системы

Android предлагает нам несколько способов чтения и записи файлов. В этом разделе мы рассмотрим вопросы работы с ресурсами и доступ к внешнему хранилищу (которым чаще всего является SD-карта). Начнем с ресурсов.

Работа с ресурсами

В главе 2 мы изучили набор каталогов, входящих в проект Android, и определили папки `assets/` и `res/` как контейнеры для наших файлов, поставляемых вместе с нашим приложением. При обсуждении файла манифеста я говорил, что мы не будем использовать каталог `res/`, поскольку это накладывает ограничения на внутреннюю структуру. Папка `assets/` — это место, куда будем помещать все наши файлы с той иерархией каталогов, которая нам нравится.

Доступ к файлам из `assets/` осуществляется с помощью класса `AssetManager`. Получить ссылку на этот класс можно так:

```
AssetManager assetManager = context.getAssets();
```

[1] С англ. — «диспетчер сенсоров». — *Примеч. пер.*

С интерфейсом Context мы уже ранее сталкивались — он реализуется классом Activity. В реальной жизни мы будем получать AssetManager как раз от активности.

Теперь, когда у нас есть AssetManager, мы можем начать открывать файлы примерно так:

```
InputStream inputStream = assetManager.open("dir/dir2/filename.txt");
```

Этот метод возвращает старый добрый Java InputStream, с помощью которого можно читать любой тип файла. Единственный аргумент метода AssetManager.open() — имя файла, связанное с папкой активов. В предыдущем примере у нас есть два каталога в папке assets/, при этом второй (dir2/) является дочерним для первого (dir/). В проекте Eclipse файл будет расположен в assets/dir/dir2/.

Создадим простую тестовую активность для проверки этого функционала. Нам нужно загрузить текстовый файл myawesometext.txt из подкаталога texts директории assets/. Содержимое этого файла будет показываться в TextView. Листинг 4.7 демонстрирует исходный код этой потрясающей активности.

Листинг 4.7. AssetsTest.java, демонстрирующий чтение файла активов

```
package com.badlogic.androidgames;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;

import android.app.Activity;
import android.content.res.AssetManager;
import android.os.Bundle;
import android.widget.TextView;

public class AssetsTest extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView textView = new TextView(this);
        setContentView(textView);

        AssetManager assetManager = getAssets();
        InputStream inputStream = null;
        try {
            inputStream = assetManager.open("texts/myawesometext.txt");
            String text = loadTextFile(inputStream);
            textView.setText(text);
        } catch (IOException e) {
            textView.setText("Couldn't load file");
        } finally {
            if (inputStream != null)
                try {
                    inputStream.close();
                } catch (IOException e) {
```

```
                textView.setText("Couldn't close file");
            }
        }
    }

    public String loadTextFile(InputStream inputStream) throws IOException {
        ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
        byte[] bytes = new byte[4096];
        int len = 0;
        while ((len = inputStream.read(bytes)) > 0)
            byteStream.write(bytes, 0, len);
        return new String(byteStream.toByteArray(), "UTF8");
    }
}
```

Никаких сюрпризов, за исключением того, что загрузка простого текста из `InputStream` в Java довольно многословна. Я написал маленький метод `loadTextFile()`, выдавливающий все байты из `InputStream` и возвращающий их в виде строки. В данном случае я подразумеваю, что кодировка файла — UTF-8. Остаток кода — отлавливание и обработка различных исключений. Рисунок 4.10 показывает вывод этой маленькой активности.

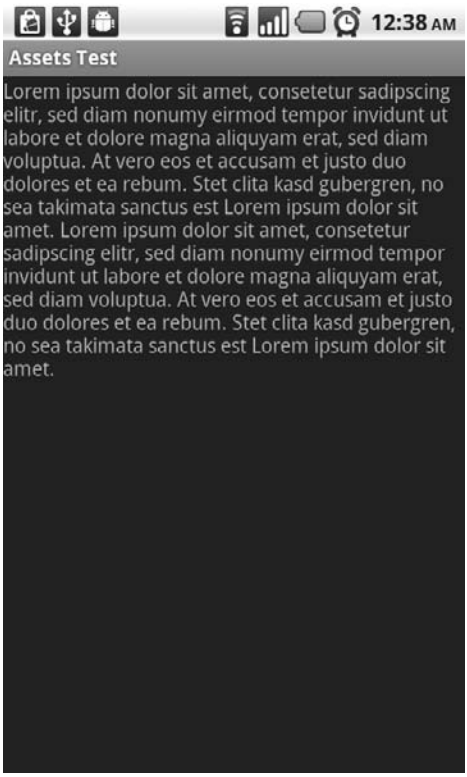

Рис. 4.10. Вывод текста

Из этого раздела вы должны вынести следующие моменты.

- Загрузка текстового файла из `InputStream` в Java несколько запутанна. Обычно мы будем осуществлять эту операцию с помощью других методов, например Apache IOUtils. Я оставлю это вам в качестве домашнего задания.
- Мы можем только читать ресурсы, но не изменять их.
- Мы можем легко изменить метод `loadTextFile()`, чтобы он загружал не текстовые, а двоичные данные. Для этого необходимо лишь возвращать вместо строки массив байтов.

Доступ к внешнему хранилищу

Хотя ресурсы прекрасно подходят для хранения изображений и звуков нашего приложения, иногда необходимо сохранять информацию и позже загружать ее вновь. Простой пример — работа с таблицей лучших результатов.

Android предлагает различные способы для реализации этого; вы можете использовать общие настройки приложения, маленькую базу данных SQLite и т. д. У всех этих возможностей есть общая черта — они не обрабатывают большие двоичные файлы. Для чего нам это может понадобиться? Хотя мы можем указать системе, чтобы она устанавливала приложение во внешнее хранилище (и таким образом не тратить память внутреннего хранилища), это будет работать только в версиях Android начиная с 2.2. В более старых версиях ОС все данные приложения будут храниться во внутренней памяти устройства. Теоретически мы могли бы включить код нашего приложения в APK-файл и загружать все ресурсы с сервера на карту памяти SD при первом запуске программы. Многие известные игры для Android так и делают.

Существуют и другие сценарии, в которых нам необходим доступ к карте памяти (которая, в общем-то, является синонимом термина «внешнее хранилище» для современных устройств). Мы могли бы позволить нашим пользователям создавать собственные уровни во внутреннем редакторе. После создания нам будет необходимо их где-то хранить, и SD-карта прекрасно для этого подходит. Итак, мы не будем использовать необычные механизмы, предлагаемые Android для хранения настроек приложения, и рассмотрим вместо этого механизм реализации чтения и записи файлов на карте SD.

Первое, что нам необходимо сделать, — запросить разрешение на доступ к внешнему хранилищу. Это делается в файле манифеста с помощью уже рассмотренного нами элемента `<user-permission>`.

Далее нужно проверить, доступно ли нам в данный момент внешнее хранилище. Например, при работе с AVD у вас есть возможность обойтись без эмулирования наличия карты памяти — тогда приложение ничего не сможет туда записывать. Другая причина не получать доступ к SD-карте — его занятость другим процессом (например, просмотра его пользователем через USB). Вот так мы проверяем состояние карты памяти:

```
String state = Environment.getExternalStorageState();
```

В результате мы получаем строку. Класс `Environment` определяет набор констант, одна из которых называется `Environment.MEDIA_MOUNTED` (ее значение — тоже строка). Если вышеуказанный метод возвращает именно эту константу, это значит, что мы имеем полный доступ (чтение и запись) к внешнему хранилищу. Обратите внимание — на самом деле вам необходимо использовать метод `equals()` для сравнения двух строк; оператор равенства в таких случаях не всегда дает верные результаты.

Итак, мы узнали, что обладаем полным доступом к внешнему хранилищу, и теперь нам необходимо получить название его корневой директории. Если нам нужен доступ к определенному файлу, путь к нему необходимо определять относительно корневого каталога. Для этого мы применим другой статический метод класса `Environment`:

```
File externalDir = Environment.getExternalStorageDirectory();
```

С этого момента мы можем использовать стандартные Java-классы для чтения и записи файлов. Создадим небольшой пример, реализующий запись файла на карту памяти, его чтение, демонстрацию его содержимого в `TextView` и удаление его с карты. В листинге 4.8 показан исходный код для этого.

Листинг 4.8. Активность ExternalStorageTest

```
package com.badlogic.androidgames;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

import android.app.Activity;
import android.os.Bundle;
import android.os.Environment;
import android.widget.TextView;

public class ExternalStorageTest extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView textView = new TextView(this);
        setContentView(textView);

        String state = Environment.getExternalStorageState();
        if (!state.equals(Environment.MEDIA_MOUNTED)) {
            textView.setText("No external storage mounted");
        } else {
            File externalDir = Environment.getExternalStorageDirectory()
            File textFile = new File(externalDir.getAbsolutePath()
                    + File.separator + "text.txt");
```

```
            try {
                writeTextFile(textFile, "This is a test. Roger");
                String text = readTextFile(textFile);
                textView.setText(text);
                if (!textFile.delete()) {
                    textView.setText("Couldn't remove temporary file");
                }
            } catch (IOException e) {
                textView.setText("something went wrong! " + e.getMessage());
            }
        }
    }

    private void writeTextFile(File file, String text) throws IOException {
        BufferedWriter writer = new BufferedWriter(new FileWriter(file));
        writer.write(text);
        writer.close();
    }

    private String readTextFile(File file) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(file));
        StringBuilder text = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null) {
            text.append(line);
            text.append("\n");
        }
        reader.close();
        return text.toString();
    }
}
```

Сначала мы проверяем физическую доступность SD-карты (если проверка не удалась, на этом все и заканчивается). Далее получаем корневой каталог хранилища и создаем новый экземпляр объекта `File`, указывающий на файл, который мы создадим в следующем выражении. Метод `writeTextFile()` использует стандартные Java-классы ввода-вывода для реализации наших целей. Если файл еще не существует, метод создаст его; в ином случае он перепишет существующий. После успешной записи текста в файл на карте мы вновь его считываем оттуда и устанавливаем в качестве содержимого `TextView`. Финальный шаг — удаление файла из внешнего хранилища. Все действия совершаются с соблюдением необходимых мер предосторожности, благодаря которым сообщения о проблемах также выводятся в `TextView`. Рисунок 4.11 демонстрирует вывод активности.

Из этого урока необходимо извлечь следующие моменты.

- Не работайте с файлами, которые вам не принадлежат. Пользователи очень рассердятся, если вы удалите фотографии с их последней вечеринки.
- Всегда проверяйте доступность внешнего хранилища.
- Не связывайтесь со служебными файлами во внешнем хранилище. Я серьезно!

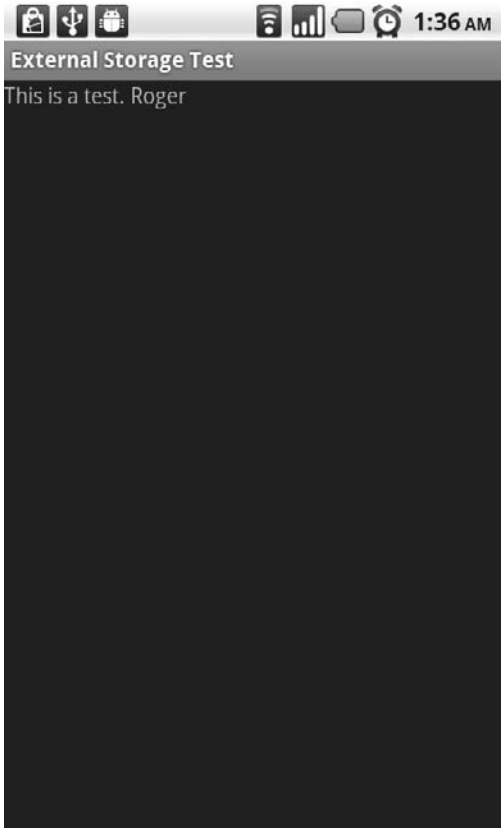

Рис. 4.11. Послание от веселого Роджера

Увидев, как легко удалить все файлы из внешнего хранилища, вы должны подумать дважды, создавая и размещая на Android Market приложение, запрашивающее доступ к карте SD, — ведь после инсталляции оно будет иметь полный доступ ко всем файлам.

Обработка звука

Android предлагает простые в применении API для воспроизведения звуковых эффектов и музыкальных файлов — как раз то, что нам нужно для написания игры. Рассмотрим их.

Работа с регуляторами громкости

Если у вас есть телефон на Android, то знаете, что при нажатии кнопок увеличения и уменьшения громкости вы управляете разными параметрами в зависимости от того, какое приложение в данный момент используете. При звонке вы регулируете громкость голоса вашего абонента, в приложении YouTube — громкость звукового сопровождения ролика, на главном экране — громкость звонка.

Android использует различные аудиопотоки для разных целей. При воспроизведении звука в игре мы оперируем с классами, осуществляющими вывод эффектов и музыки в специальный поток, называемый музыкальным. Однако прежде, чем заняться воспроизведением, нам необходимо для начала убедиться в том, что наши кнопки громкости управляют нужным нам потоком. Для этого в интерфейсе `Context` имеется специальный метод:

```
context.setVolumeControlStream(AudioManager.STREAM_MUSIC);
```

Как обычно, реализация `Context` будет осуществлена нашей активностью. После вызова этого метода кнопки громкости будут управлять музыкальным потоком, в который мы будем выводить звуковые эффекты и музыку. Метод вызывается лишь один раз за жизненный цикл активности. Лучшее место для этого — метод `Activity.onCreate()`.

Писать пример для демонстрации работы одной строки кода нецелесообразно. Поэтому сейчас обойдемся без нее. Просто помните о необходимости использовать этот метод для всех активностей, работающих со звуком.

Использование звуковых эффектов

В главе 3 мы обсудили разницу между потоковой музыкой и воспроизведением звуковых эффектов. Последние хранятся в памяти и длятся обычно не более нескольких секунд. Android предлагает нам класс `SoundPool`, делающий решение этой задачи весьма простым.

Мы можем просто инициализировать новые экземпляры объекта `SoundPool` следующим образом:

```
SoundPool soundPool = new SoundPool(20, AudioManager.STREAM_MUSIC, 0);
```

Первый параметр — количество звуковых эффектов, которые могут воспроизводиться одновременно. Это не значит, что мы не сможем загрузить большее количество звуковых эффектов — это ограничение одновременного воспроизведения. Второй параметр определяет звуковой поток, в который будет выводить звук класс `SoundPool`. Мы выбрали музыкальный поток, к которому привязали кнопки громкости. Последний параметр в настоящее время не используется и всегда равен 0.

Для загрузки звукового эффекта из аудиофайла в память мы можем применить метод `SoundPool.load()`. Все такие файлы мы храним в папке `assets/`, поэтому нам необходимо перегрузить метод `SoundPool.load()`, чтобы он принимал дескриптор `AssetFileDescriptor`. Как его получить? Легко — с помощью `AssetManager`, который мы обсуждали ранее. Именно так мы загрузим OGG-файл `explosion.ogg` из каталога `assets/`:

```
AssetFileDescriptor descriptor = assetManager.openFd("explosion.ogg");
int explosionId = soundPool.load(descriptor, 1);
```

Дескриптор `AssetFileDescriptor` получается напрямую через метод `AssetManager.openFd()`. Загрузка звукового эффекта с помощью `SoundPool` также не составляет проблемы. Первый аргумент метода `SoundPool.load()` — это полученный нами только что дескриптор, второй определяет приоритет звукового эффекта (в настоящее

время он не задействован и должен быть установлен равным 1 для будущей совместимости).

Метод SoundPool.load() возвращает значение типа integer, являющееся указателем на загруженный эффект. Когда нам необходимо его воспроизвести, мы используем этот указатель для информирования SoundPool о том, какой именно эффект необходимо воспроизвести. Воспроизведение звукового эффекта реализуется весьма просто:

```
soundPool.play(explosionId, 1.0f, 1.0f, 0, 0, 1);
```

Первый аргумент — тот самый указатель, полученный нами от метода SoundPool.load(). Следующие два параметра определяют уровень громкости для левого и правого каналов. Их значения должны находиться в диапазоне от 0 (тишина) до 1 (кровь из ушей). Следующие два аргумента используются редко. Первый из них — все тот же приоритет, пока не используемый и всегда равный 0. Второй определяет количество повторений звукового эффекта. Я бы не рекомендовал использовать циклические звуковые эффекты, поэтому лучше установить для него значение 0. Последний параметр — скорость воспроизведения. Задание ему значения больше единицы означает воспроизведение эффекта со скоростью, большей, чем он был записан, меньше единицы — замедленное воспроизведение.

Если эффект нам больше не нужен, мы можем очистить его из памяти с помощью метода SoundPool.unload():

```
soundPool.unload(explosionId);
```

В качестве параметра ему передается дескриптор звукового эффекта, полученный от метода SoundPool.load().

Обычно для игры нужен лишь один экземпляр SoundPool, используемый для загрузки, воспроизведения и выгрузки звуковых эффектов. После окончания работы со звуком SoundPool нам больше не нужен, поэтому необходимо вызвать метод SoundPool.release(), освобождающий все ресурсы, связанные с SoundPool. После этого мы, конечно, не сможем применять SoundPool (загруженные им звуковые эффекты также будут удалены из памяти).

Создадим небольшую тестовую активность, воспроизводящую звук взрыва при каждом касании экрана. Сейчас мы знаем все, чтобы это сделать, поэтому листинг 4.9 не будет содержать для вас никаких сюрпризов.

Листинг 4.9. SoundPoolTest.java: воспроизведение звуковых эффектов

```
package com.badlogic.androidgames;

import java.io.IOException;

import android.app.Activity;
import android.content.res.AssetFileDescriptor;
import android.content.res.AssetManager;
import android.media.AudioManager;
import android.media.SoundPool;
import android.os.Bundle;
```

```
import android.view.MotionEvent;
import android.view.View;
import android.view.View.OnTouchListener;
import android.widget.TextView;

public class SoundPoolTest extends Activity implements OnTouchListener {
    SoundPool soundPool;
    int explosionId = -1;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView textView = new TextView(this);
        textView.setOnTouchListener(this);
        setContentView(textView);

        setVolumeControlStream(AudioManager.STREAM_MUSIC);
        soundPool = new SoundPool(20, AudioManager.STREAM_MUSIC, 0);

        try {
            AssetManager assetManager = getAssets();
            AssetFileDescriptor descriptor = assetManager
                    .openFd("explosion.ogg");
            explosionId = soundPool.load(descriptor, 1);
        } catch (IOException e) {
            textView.setText("Couldn't load sound effect from asset, "
                    + e.getMessage());
        }
    }

    @Override
    public boolean onTouch(View v, MotionEvent event) {
        if (event.getAction() == MotionEvent.ACTION_UP) {
            if (explosionId != -1) {
                soundPool.play(explosionId, 1, 1, 0, 0, 1);
            }
        }
        return true;
    }
}
```

Начало традиционное — наследуем наш класс от `Activity` и реализуем в нем интерфейс `OnTouchListener`, чтобы можно было обрабатывать касания экрана. В нашем классе есть два члена: `SoundPool` и указатель на звуковой эффект, который будет загружаться и воспроизводиться (мы установили его начальное значение равным –1, пока звуковой эффект еще не загружен).

В методе `OnCreate()` мы делаем то, что выполняли уже неоднократно: создаем `TextView`, регистрируем активность как `OnTouchListener` и устанавливаем `TextView` в качестве средства отображения.

Следующая строка — привязка кнопок управления громкостью к музыкальному потоку. Далее создаем SoundPool и настраиваем его на одновременное воспроизведение 20 эффектов — этого должно быть достаточно для большинства игр. Наконец, получаем дескриптор AssetFileDescriptor для файла explosion.ogg (помещенного в каталог assets/) от AssetManager. Для проигрывания звука мы просто передаем этот дескриптор методу SoundPool.load() и храним полученный указатель. Метод SoundPool.load() обрабатывает исключение (если при загрузке возникли какие-то проблемы, показываем на экране сообщение об ошибке).

В методе OnTouch() просто проверяем отрыв пальца от дисплея, что означает факт касания. В этом случае, если звуковой эффект был загружен успешно (то есть его указатель не равен –1), мы его воспроизводим.

Если мы запустим нашу маленькую активность и дотронемся до экрана, то услышим звук взрыва. Если вы начнете быстро барабанить по дисплею, то услышите множество взрывов, перекрывающих друг друга. Будет довольно трудно превысить лимит в 20 одновременно воспроизводимых эффектов. Однако если это все же произойдет, один из звуков прекратится, чтобы дать место вновь запрошенному.

Заметьте — в этом примере мы не выгружаем звук и не освобождаем ресурсы SoundPool. Это сделано для краткости — обычно SoundPool выгружается в методе onPause() перед уничтожением активности. Помните — всегда необходимо выгружать из памяти все, что вам больше не нужно.

Хотя класс SoundPool очень прост в использовании, есть несколько моментов, на которые стоит обратить внимание.

- Метод SoundPool.load() выполняет загрузку асинхронно. Это означат, что вам придется немного подождать перед вызовом метода SoundPool.play(), поскольку загрузка может еще не закончиться. К сожалению, не существует универсального способа проверить состояние процесса загрузки эффекта. Это можно сделать только при использовании SDK SoundPool версии 8, а нам необходимо обеспечить совместимость со всеми версиями Android. Обычно это не является большой проблемой — вы можете загрузить другие активы до того, как звуковой эффект будет воспроизведен впервые.
- SoundPool имеет некоторые проблемы при работе с MP3-файлами и большими фрагментами (чья длительность определяется как «больше 5–6 секунд»). Обе неприятности не документированы, поэтому нет четкого правила для определения, вызовет ваш звуковой фрагмент проблемы или нет. В качестве общего правила я бы предложил отказаться от MP3 в пользу формата OGG и использовать как можно меньшие частоту дискретизации и длительность, при которых качество звука остается приемлемым.

ПРИМЕЧАНИЕ

Как и для всех обсужденных нами ранее API, функциональность SoundPool гораздо шире. Я упоминал, что вы можете воспроизводить звуковые эффекты циклически. Для этого вы получаете ID от метода SoundPool.play(), который вы можете использовать для паузы или приостановки воспроизведения циклического эффекта. Изучите документацию по классу SoundPool на сайте Android Developers, если вам необходима такая функциональность.

Потоковая музыка

Короткие звуковые эффекты легко помещаются в ограниченной области памяти (куче), которую приложению Android выделяет операционная система. Это не касается более объемных музыкальных файлов. Поэтому нам необходимо осуществлять потоковую передачу музыки на устройство воспроизведение (то есть читать в единицу времени небольшой фрагмент файла, декодировать его в PCM-данные и передавать его аудиочипу).

Звучит пугающе, правда? К счастью, у нас есть класс `MediaPlayer`, делающий за нас всю работу. Все, что нам нужно, — указать ему на аудиофайл и дать команду его воспроизведения. Инициализация класса `MediaPlayer` выполняется проще некуда:

```
MediaPlayer mediaPlayer = new MediaPlayer();
```

Далее нам необходимо сообщить `MediaPlayer`, какой файл воспроизводить. Это делается (как и в случае со звуковым эффектом) с помощью `AssetFileDescriptor`:

```
AssetFileDescriptor descriptor = assetManager.openFd("music.ogg");
mediaPlayer.setDataSource(descriptor.getFileDescriptor(),
descriptor.getStartOffset(), descriptor.getLength());
```

Однако здесь немного больше кода, чем в случае с `SoundPool`. Метод `MediaPlayer.setDataSource()` не принимает `AssetDescriptor` напрямую — ему нужен `FileDescriptor`, получаемый через `AssetFileDescriptor.getFileDescriptor()`. Кроме того, нам необходимо определить сдвиг и длительность воспроизводимого файла. Почему сдвиг? Дело в том, что все ресурсы на самом деле хранятся в одном файле. Чтобы `MediaPlayer` получил позицию начала нашего файла, необходимо предоставить ему величину сдвига внутри файла ресурсы.

Перед началом воспроизведения музыкального файла придется вызвать еще один метод, подготавливающий `MediaPlayer` к воспроизведению:

```
mediaPlayer.prepare();
```

Это действие на самом деле открывает файл и проверяет, можно ли его читать и воспроизводить экземпляром `MediaPlayer`. Теперь мы можем воспроизводить, приостанавливать, останавливать, зацикливать звуковой файл и менять громкость.

Для начала воспроизведения вызывается следующий метод:

```
mediaPlayer.start();
```

Обратите внимание: его можно вызывать только после успешного вызова `MediaPlayer.prepare()` — иначе возникнет ошибка выполнения.

Поставить воспроизведение на паузу можно с помощью метода `pause()`:

```
mediaPlayer.pause();
```

Вызов этого метода также легитимен лишь после успешной подготовки класса `MediaPlayer` и начала воспроизведения. Для возобновления воспроизведения мы

просто опять вызываем метод MediaPlayer.start() без какой-либо предварительной подготовки.

Для остановки воспроизведения используется метод stop():

```
mediaPlayer.stop();
```

Обратите внимание — чтобы запустить остановленный MediaPlayer, необходимо сначала вновь вызвать MediaPlayer.prepare().

Мы можем заставить MediaPlayer циклически воспроизводить файл, воспользовавшись следующим методом:

```
mediaPlayer.setLooping(true);
```

Для регулировки громкости воспроизведения применяется такой метод:

```
mediaPlayer.setVolume(1, 1);
```

Громкость устанавливается отдельно для левого и правого каналов. В документации не указываются диапазоны для этих значений, но экспериментально было установлено, что они должны помещаться между 0 и 1.

Наконец, нам необходимо знать, закончилось ли воспроизведение. Это можно сделать двумя способами. Можно зарегистрировать OnCompletionListener для MediaPlayer, чтобы он вызывался при окончании воспроизведения файла:

```
mediaPlayer.setOnCompletionListener(listener);
```

Другой метод — получение состояния MediaPlayer с помощью еще одного метода:

```
boolean isPlaying = mediaPlayer.isPlaying();
```

Понятно, что при установке MediaPlayer в режим циклического воспроизведения ни один из этих способов не сообщит, что MediaPlayer остановил воспроизведение.

Не забудьте — после окончания работы с экземпляром MediaPlayer нужно освободить занятые им ресурсы. Делается это так:

```
mediaPlayer.release();
```

Как уже говорилось, освобождать ресурсы при завершении работы с объектом является хорошей практикой.

Если мы не установили MediaPlayer в режим циклического воспроизведения, можем перезапустить его вызовом методов MediaPlayer.prepare() и MediaPlayer.start().

Большинство этих методов вызываются асинхронно. Это значит, что даже если вы вызвали MediaPlayer.stop(), значение MediaPlayer.isPlaying() может еще некоторое время оставаться равным true. Обычно это не повод для беспокойства — в большинстве игр мы устанавливаем MediaPlayer на воспроизведение по циклу и останавливаем его по мере необходимости (например, при переходе на другой экран, которому соответствует другая музыка).

Напишем небольшую активность, в которой будем циклически воспроизводить звуковой файл из каталога assets/. Этот звуковой эффект будет приостанавливаться и возобновляться в соответствии с жизненным циклом активности — при ее постановке на паузу приостановится и музыка, при продолжении соответствующим образом музыка продолжится с места остановки. Реализация этой функциональности показана в листинге 4.10.

Листинг 4.10. MediaPlayerTest.java: воспроизведение звуковых потоков

```
package com.badlogic.androidgames;

import java.io.IOException;

import android.app.Activity;
import android.content.res.AssetFileDescriptor;
import android.content.res.AssetManager;
import android.media.AudioManager;
import android.media.MediaPlayer;
import android.os.Bundle;
import android.widget.TextView;

public class MediaPlayerTest extends Activity {
    MediaPlayer mediaPlayer;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView textView = new TextView(this);
        setContentView(textView);

        setVolumeControlStream(AudioManager.STREAM_MUSIC);
        mediaPlayer = new MediaPlayer();
        try {
            AssetManager assetManager = getAssets();
            AssetFileDescriptor descriptor =
                    assetManager.openFd("music.ogg");
            mediaPlayer.setDataSource(descriptor.getFileDescriptor(),
                    descriptor.getStartOffset(), descriptor.getLength());
            mediaPlayer.prepare();
            mediaPlayer.setLooping(true);
        } catch (IOException e) {
            textView.setText("Couldn't load music file, " + e.getMessage());
            mediaPlayer = null;
        }
    }

    @Override
    protected void onResume() {
        super.onResume();
        if (mediaPlayer != null) {
```

```
            mediaPlayer.start();
        }
    }

    protected void onPause() {
        super.onPause();
        if (mediaPlayer != null) {
            mediaPlayer.pause();
            if (isFinishing()) {
                mediaPlayer.stop();
                mediaPlayer.release();
            }
        }
    }
}
```

Мы храним ссылку на `MediaPlayer` в виде члена нашей активности. В методе `OnCreate()` мы просто создаем `TextView` для вывода сообщений об ошибках, как всегда.

Перед тем как пользоваться `MediaPlayer`, необходимо удостовериться, что кнопки управления громкостью привязаны к музыкальному потоку. После этого мы инициализируем объект `MediaPlayer` и получаем дескриптор `AssetFileDescriptor` для файла `music.ogg` (находящегося в каталоге `assets/`) из `AssetManager`, а также устанавливаем его в качестве источника данных для `MediaPlayer`. Все, что остается сделать после этого, — подготовить `MediaPlayer` к воспроизведению и установить для него режим циклического воспроизведения. Если что-то пойдет не так, член `MediaPlayer` устанавливается в `null`, чтобы позже мы могли определить, успешной ли была загрузка. Кроме того, мы выводим сообщения об ошибках в `TextView`.

В методе `onResume()` мы стартуем `MediaPlayer` (если его создание совершилось успешно). `OnResume()` — прекрасное для этого место, поскольку он вызывается как после `onCreate()`, так и после `onPause()`. В первом случае мы начинаем воспроизведение впервые, во втором — возобновляем поставленный на паузе `MediaPlayer`.

Метод `onPause()` приостанавливает воспроизведение `MediaPlayer`. Если активность предназначена к уничтожению, мы останавливаем `MediaPlayer` и освобождаем связанные с ним ресурсы.

При тестировании различных операций проверьте также его реакцию на приостановку и возобновление активности, заблокировав экран или временно переключившись на Home (Стартовый экран). При возобновлении `MediaPlayer` продолжит воспроизведение с того места, на котором он остановился в прошлый раз.

При работе с потоковым звуком есть несколько тонкостей.

- Методы `MediaPlayer.start()`, `MediaPlayer.pause()` и `MediaPlayer.resume()` могут вызываться только в определенных ситуациях. Никогда не пытайтесь вызывать их, не подготовив предварительно `MediaPlayer`. Вызывайте `MediaPlayer.start()` только после подготовки `MediaPlayer` или при возобновлении воспроизведения после вызова метода `MediaPlayer.pause()`.

- Экземпляры MediaPlayer довольно тяжеловесны. Если их много, они могут порядочно загрузить систему. Всегда необходимо стараться использовать только один экземпляр этого объекта для воспроизведения. Звуковые эффекты лучше реализовывать с помощью класса SoundPool.
- Не забывайте связывать кнопки управления громкостью с музыкальным потоком, иначе ваши игроки не смогут управлять уровнем звука.

Наша глава почти закончена, но осталась еще одна большая тема — 2D-графика.

Основы программирования графики

Android предлагает нам два больших API для прорисовки экрана. Один из них используется в основном для программирования 2D-графики, второй — для аппаратно ускоренного 3D. В этой и следующей главах мы сфокусируемся на разработке 2D-графики с помощью Canvas API, являющегося, по сути, качественной оберткой для библиотеки Skia и подходящего для большинства сложных 2D-игр. Однако прежде чем этим заняться, нам необходимо прояснить для себя два аспекта: переход в полноэкранный режим и защиту от блокировки.

Использование защиты от блокировки

Если вы перестанете работать с нашим тестовым приложением на несколько секунд, экран вашего телефона потускнеет. Прежняя яркость вернется, только если вы коснетесь экрана или нажмете кнопку. Чтобы дисплей всегда был доступен, мы можем использовать так называемую защиту от блокировки. Первое, что нам необходимо сделать, — добавить соответствующий тег <user-permission> в файл манифеста с именем android.permission.WAKE_LOCK. Это позволит нам использовать класс WakeLock.

Экземпляр WakeLock получается от PowerManager следующим образом:

```
PowerManager powerManager =
(PowerManager)context.getSystemService(Context.POWER_SERVICE);
WakeLock wakeLock = powerManager.newWakeLock(PowerManager.FULL_WAKE_LOCK, "My Lock");
```

Как и другие системные службы, мы получаем PowerManager от экземпляра Context. Метод PowerManager.newWakeLock() принимает два аргумента: тип блокировки и произвольно определяемый текст тега. Существует два типа защиты от блокировки; для наших целей подходит PowerManager.FULL_WAKE_LOCK. Его использование гарантирует, что экран все время будет включен, процессор будет работать на полную мощность, а клавиатура будет доступна.

Для включения защиты от блокировки необходимо вызвать следующий метод:

```
wakeLock.acquire();
```

С момента вызова телефон перестанет блокироваться независимо от времени, в течение которого пользователь не производит действий. При приостановке или уничтожении нашего приложения необходимо отключить защиту от блокировки:

```
wakeLock.release();
```

Обычно мы инициализируем экземпляр WakeLock в методе Activity.onCreate(), вызываем WakeLock.acquire() в методе Activity.onResume(), а WakeLock.release() вызываем в методе Activity.onPause(). Таким образом, мы гарантируем, что наше приложение будет корректно работать в режиме паузы или восстановления. Итак, нам необходимо добавить всего четыре строчки кода, поэтому предлагаю не писать для этого полноценный пример. Вместо этого добавим их в пример для полноэкранного режима и посмотрим на результаты.

Полноэкранный режим

Перед тем как с головой погрузиться в рисование первых фигур с помощью Android API, решим еще один вопрос. До этого момента все наши активности демонстрировали свои строки заголовка. Кроме того, была видна панель оповещения. Хотелось бы создать у наших будущих игроков больший эффект погружения, убрав с экрана эти элементы. Сделать это можно двумя простыми вызовами:

```
requestWindowFeature(Window.FEATURE_NO_TITLE);
getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
WindowManager.LayoutParams.FLAG_FULLSCREEN);
```

Первая строка убирает строку заголовка активности. Чтобы убрать еще и панель оповещения и таким образом сделать активность полноэкранной, мы вызываем второй метод. Обратите внимание: оба этих метода необходимо вызывать до установки представления отображения для нашей активности. Листинг 4.11 демонстрирует очень простую тестовую активность, переходящую в полноэкранный режим.

Листинг 4.11. FullScreenTest.java; делаем нашу активность полноэкранной

```
package com.badlogic.androidgames;

import android.os.Bundle;
import android.view.Window;
import android.view.WindowManager;

public class FullScreenTest extends SingleTouchTest {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                WindowManager.LayoutParams.FLAG_FULLSCREEN);
        super.onCreate(savedInstanceState);
    }
}
```

Что тут происходит? Мы просто наследуемся от класса TouchTest, созданного нами ранее, и переопределяем его метод onCreate(), в котором включаем полноэкранный режим, и затем вызываем базовый метод наследуемого класса (в данном случае это активность TouchTest), который делает всю остальную работу по созданию активности. Вновь обращаю ваше внимание: эти два метода нужно вызывать перед

установкой контейнера для содержимого, поэтому метод базового класса вызывается после этих методов. Кроме того, мы зафиксировали ориентацию нашей активности в портретном режиме в файле манифеста. Ведь вы не забывали добавлять туда элементы <activity> для каждой созданной нами активности? С этого момента мы всегда будем жестко задавать портретный или ландшафтный режим, поскольку нам не нужно изменение координатной системы.

Наследуясь от TouchTest, мы получаем полностью рабочий пример, с помощью которого сможем изучить координатную систему, в которой будем рисовать. Активность покажет нам координаты касания экрана (как в старом примере TouchTest). Разница в том, что теперь, в полноэкранном режиме, максимальные величины координат совпадают с разрешением дисплея (минус 1 по обоим направлениям, ведь мы начинаем с (0; 0)). В случае с Nexus One координатная система будет начинаться с (0; 0) и заканчиваться в (479; 799) в портретном режиме (разрешение 480×800).

Хотя у вас может создаться впечатление, что экран перерисовывается постоянно, на самом деле это не так. В классе TouchTest мы обновляем TextView каждый раз, когда происходит событие касания экрана (что, в свою очередь, заставляет TextView перерисовываться). Сам по себе TextView обновляться не будет. Для игры необходимо, чтобы экран перерисовывался как можно чаще — желательно внутри нашего главного потока. Но мы начнем с более простого — с непрерывной визуализации в потоке пользовательского интерфейса.

Непрерывная визуализация в потоке пользовательского интерфейса

Все, что мы сделали до этого момента, — обеспечили изменение текста в TextView при необходимости. Перерисовка осуществляется самим TextView. Создадим наш собственный View, единственной задачей которого будет предоставление нам возможности рисовать на экране. Нам необходимо, чтобы этот View обновлялся как можно чаще. Нам также нужен простой способ выполнять рисование каким-нибудь волшебным способом.

Хотя на слух это может казаться довольно сложной задачей, на самом деле Android обеспечивает легкость в реализации таких задач. Все, что нам надо, — создать класс, наследуемый от View, и переопределить его метод View.onDraw(). Этот метод вызывается системой Android каждый раз, когда View требуется перерисовка. Выглядит он примерно так:

```
class RenderView extends View {
    public RenderView(Context context) {
        super(context);
    }

    protected void onDraw(Canvas canvas) {
        // реализация впереди
    }
}
```

Пока не так уж все сложно, не правда ли? Мы передаем методу `onDraw()` экземпляр класса `Canvas`. В ближайших разделах он станет нашей рабочей лошадкой, позволяя нам рисовать фигуры и изображения в другом изображении или `View` (или на поверхности экрана, о чем мы поговорим чуть позже).

Использовать объект `RenderView` можно так же, как `TextView`, — мы просто устанавливаем его в качестве контейнера содержимого нашей активности и обрабатываем все процессы, которые нам нужны. Однако пока этот класс не слишком полезен для нас по двум причинам: во-первых, на самом деле он ничего не рисует; во-вторых, даже когда сможет, то будет делать это только тогда, когда активность получит запрос на перерисовку (то есть при ее старте или возобновлении либо после исчезновения перекрывающего ее диалогового окна). Как нам заставить его перерисовываться самостоятельно?

Вот так:

```
protected void onDraw(Canvas canvas) {
    // все рисование реализуется здесь
    invalidate();
}
```

Вызов метода `View.invalidate()` в конце `onDraw()` сообщит Android о необходимости перерисовки `RenderView` при первой возможности. Все это по-прежнему происходит в пользовательском потоке, который обычно слегка нетороплив. Тем не менее у нас уже есть непрерывная перерисовка в методе `onDraw()`, хотя и не слишком быстрая. Мы исправим это позже; пока для наших задач этого достаточно.

Итак, теперь пора вернуться к загадочному классу `Canvas`. Это довольно мощный инструмент, обертывающий низкоуровневую библиотеку Skia, предназначенную для выполнения 2D-рендеринга центральным процессором. Класс `Canvas` предлагает нам множество методов рисования для фигур, изображений и даже текста.

Где мы будем использовать эти методы? Это зависит от поставленных целей. `Canvas` может перерисовываться в экземпляр `Bitmap` — другого класса, предлагаемого 2D API Android (мы рассмотри его позже). В данном случае прорисовка будет происходить для части экрана, занимаемой `View`. Конечно, это довольно значительное упрощение: на самом деле прорисовка производится не на самом экране, а в некое изображение, которое система позже будет использовать в комбинации с изображениями от всех других `View` активности, чтобы составить из них конечную итоговую картинку. Получившееся изображение передается графическому процессору, отображающему его на экране еще более загадочными путями.

Вообще, у нас нет необходимости заботиться о таких деталях. В данном случае наш `View` займет всю площадь дисплея, поэтому может перерисовываться также во фреймбуфер системы. До конца обсуждения представим, что мы производим перерисовку непосредственно во фреймбуфер, оставляя системе делать всю скучную работу вроде вертикальной трассировки и двойной буферизации.

Метод `onDraw()` будет вызываться так часто, как это позволяет система. Для нас это очень похоже на тело нашего (теоретического) главного игрового цикла. Если

мы будем реализовывать игру с помощью этого метода, в него можно поместить всю игровую логику. Однако мы не будем делать этого по разным причинам (одной из которых является производительность).

Давайте сделаем что-нибудь интересное. Каждый раз, когда выходит новое API для рисования, я пишу маленький тест, проверяющий частоту перерисовки экрана. Единственное содержание вызываемого метода перерисовки — заполнение экрана случайным цветом. Мне нужно только найти метод для данного API, который позволит это сделать, не задумываясь о деталях реализации. Напишем подобный тест для нашей реализации `RenderView`.

Метод класса `Canvas`, заполняющий цель прорисовки определенным цветом, называется `Canvas.drawRGB()`:

```
Canvas.drawRGB(int r, int g, int b);
```

Аргументы `r`, `g` и `b` хранят по одному компоненту цвета, которым мы хотим заполнить экран. Каждый из них принимает значение от 0 до 255, поэтому цвета определяются в формате RGB888. Если вы не помните подробностей, касающихся цветов, вернитесь к пункту «Цифровое кодирование цветов» подраздела «Графика» раздела «Код: скучная рутина» главы 3 — информация оттуда еще понадобится нам в этой главе.

Листинг 4.12 показывает код нашего калейдоскопа.

ВНИМАНИЕ

Запуск этого кода приведет к быстрому заполнению экрана случайными цветами. Если у вас эпилепсия или проблемы с восприятием резких изменений цвета, не запускайте его.

Листинг 4.12. Активность RenderViewTest

```
package com.badlogic.androidgames;

import java.util.Random;

import android.app.Activity;
import android.content.Context;
import android.graphics.Canvas;
import android.os.Bundle;
import android.view.View;
import android.view.Window;
import android.view.WindowManager;

public class RenderViewTest extends Activity {
    class RenderView extends View {
        Random rand = new Random();

        public RenderView(Context context) {
            super(context);
        }

        protected void onDraw(Canvas canvas) {
```

```
            canvas.drawRGB(rand.nextInt(256), rand.nextInt(256),
                    rand.nextInt(256));
            invalidate();
        }
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                WindowManager.LayoutParams.FLAG_FULLSCREEN);
        setContentView(new RenderView(this));
    }
}
```

Код для нашего первого графического примера довольно сложен. Мы определяем класс `RenderView` как внутренний для активности `RenderViewTest`. Этот класс наследуется от `View` и имеет обязательный конструктор, также переопределенный метод `onDraw()`. Кроме того, он включает в себя внутренний член типа `Random` (мы будем использовать его для генерации случайных цветов).

Метод `onDraw()` предельно прост. В нем мы сначала командуем `Canvas` заполнить весь контейнер представления случайным цветом. Для каждого компонента цвета мы с помощью метода `Random.nextInt()` определяем случайное число от 0 до 255, после чего сообщаем системе, что ожидаем перерисовки экрана (то есть вызова метода `onDraw()`) как можно быстрее.

Метод `onCreate()` активности включает полноэкранный режим и устанавливает экземпляр `RenderView` в качестве контейнера представления. Для сохранения краткости примера в данном случае я решил игнорировать защиту от блокировки.

Делать скриншот этого примера не имеет смысла. Все, что он делает, — заполняет экран случайным цветом так часто, как это позволяет система для потока пользовательского интерфейса. Тут нечем особенно хвастаться. Вместо этого выполним нечто более интересное — порисуем фигурки.

ПРИМЕЧАНИЕ

Хотя описанный метод непрерывной перерисовки является вполне работоспособным, я настоятельно не рекомендую его использовать. В потоке пользовательского интерфейса нужно делать как можно меньше действий. Скоро мы обсудим, как поместить наши функции в отдельный поток, в котором мы также сможем реализовать нашу игровую логику.

Получение разрешения экрана и координатной системы

В главе 2 мы много говорили о фреймбуфере и его свойствах. Вы помните, что фреймбуфер хранит цвета пикселов, показываемых на экране. Количество доступных нам пикселов определяется разрешением экрана (шириной и высотой в пикселах).

Теперь с нашей собственной реализацией `View`, мы не перерисовываем фреймбуфер непосредственно. Но поскольку наш `View` занимает весь экран, мы можем

представить, что это так. Чтобы узнать, где мы можем размещать элементы нашей игры, нам нужно узнать количество пикселов по осям x и y (то есть ширину и высоту дисплея).

Класс `Canvas` содержит два метода, помогающие нам в этом:

```
int width = canvas.getWidth();
int height = canvas.getHeight();
```

Они возвращают ширину и высоту элемента, для которого `Canvas` осуществляет прорисовку. Обратите внимание — в зависимости от текущей ориентации активности ширина может быть больше или меньше высоты. Например, в моем Nexus One установлено разрешение 480 × 800 пикселов в портретном режиме, поэтому метод `Canvas.getWidth()` вернет 480, а `Canvas.getHeight()` — 800. В ландшафтном режиме эти значения меняются местами.

Вторая часть необходимой информации — координатная система, используемая для рендеринга. Прежде всего, значения имеют только целочисленные координаты пикселов (существует так называемая концепция субпикселов, но мы пока игнорируем ее). Мы уже также знаем, что начало координатной системы всегда находится в точке (0; 0) в левом верхнем углу дисплея — неважно, в портретном или ландшафтном режиме. Ось x всегда направлена слева направо, ось y — всегда вниз. На рис. 4.12 показан гипотетический экран с разрешением 48 × 32 пиксела в ландшафтном режиме.

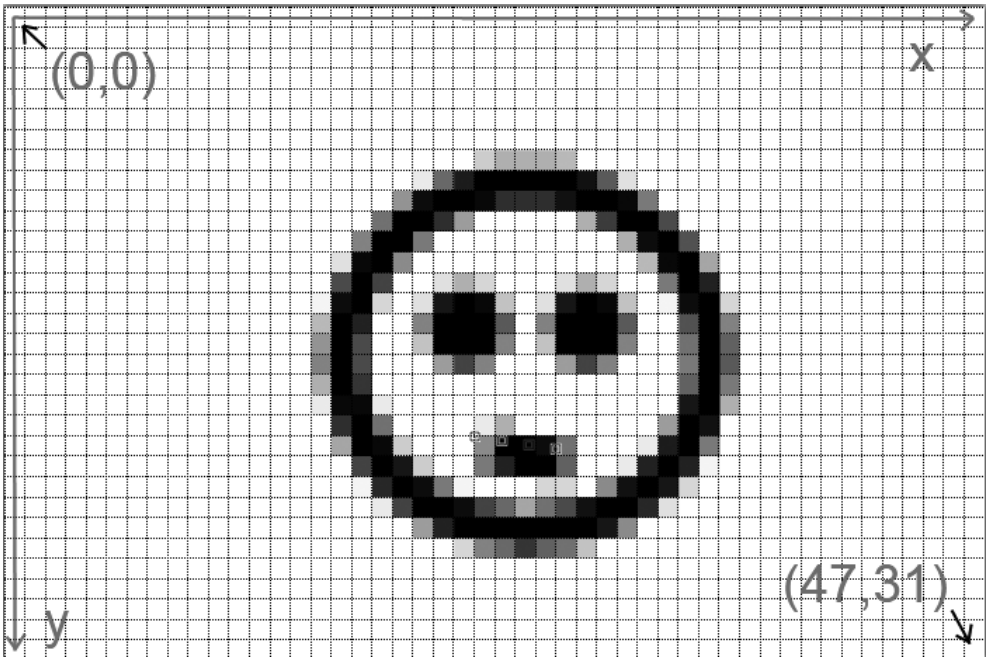

Рис. 4.12. Координатная система экрана 48 × 32 пиксела

Обратите внимание — начало координатной системы на рисунке совпадает с левым верхним пикселом экрана. Правый нижний пиксел экрана имеет координаты не (48; 32), как мы могли бы ожидать, а (47; 31). В целом (`width – 1`; `height – 1`) всегда является позицией правого нижнего пиксела экрана.

Рисунок 4.12 демонстрирует координатную систему гипотетического дисплея в ландшафтном режиме. Теперь вы можете представить, как дисплей будет выглядеть в портретном режиме.

Все методы прорисовки `Canvas` действуют внутри этой координатной системы. Обычно мы можем оперировать с гораздо большим количеством пикселов, чем 48 × 32 (например, 800 × 480). Теперь займемся рисованием пикселов, линий, кругов и прямоугольников.

ПРИМЕЧАНИЕ

Вы, вероятно, заметили, что разные устройства могут иметь различные разрешения экрана. Мы разберемся с этой проблемой в следующей главе, пока просто сконцентрируемся на получении чего-нибудь на дисплее.

Рисование простых фигур

Каких-то 150 страниц — и вот мы уже готовы нарисовать первый пиксел. Сейчас мы быстро пробежимся по некоторым методам прорисовки, предлагаемым классом `Canvas`.

Рисование пикселов. Начнем с рисования одного пиксела, которое осуществляется следующим методом:

```
Canvas.drawPoint(float x, float y, Paint paint);
```

Первое, что бросается в глаза, — координаты пикселов определены в типе `float`. Кроме того, `Canvas` не дает нам определить цвет напрямую, требуя вместо этого от нас экземпляр класса `Paint`.

Не волнуйтесь из-за того, что мы определили координаты как `float`. `Canvas` оснащен довольно мощным инструментарием, позволяющим нам перерисовывать не только целочисленные координаты, поэтому метод и принимает не только целые числа. Пока нам, правда, подобная функциональность не нужна; мы вернемся к ней в следующей главе.

Класс `Paint` хранит информацию о стиле и цвете, используемую для рисования фигур, текста и изображений. Для рисования фигур нам интересны только две вещи: цвет и стиль, хранимые в `Paint`. Поскольку пиксел не имеет стиля, сконцентрируемся на цвете. Вот так производится инициализация класса `Paint` и установка цвета в нем:

```
Paint paint = new Paint();
paint.setARGB(alpha, red, green, blue);
```

Создание экземпляра класса — довольно несложное упражнение. Метод `Paint.setARGB()` тоже не должен вызвать у вас затруднений. Каждый его аргумент представляет один из компонентов цвета и имеет диапазон от 0 до 255. Таким образом,

в данном случае мы используем цветовую схему ARGB8888. Можно применять и другой метод для установки цвета для экземпляра Paint:

```
Paint.setColor(0xff00ff00);
```

Здесь мы передаем методу один параметр — 32-битное целое число. Оно также закодировано в схеме ARGB8888 (в данном случае зеленая составляющая задана с полной непрозрачностью). Кроме того, класс `Color` определяет несколько статических констант, кодирующих цвета (например, `Color.RED`, `Color.YELLOW`). Вы можете использовать их, если не хотите оперировать шестнадцатиричными числами.

Рисование линий. Чтобы нарисовать линию, мы используем следующий метод `Canvas`:

```
Canvas.drawLine(float startX, float startY, float stopX, float stopY, Paint paint);
```

Первые два аргумента определяют координаты начальной точки линии, следующие два — конечные, последний задает экземпляр `Paint`. Полученная в результате линия будет в один пиксел толщиной. Если мы хотим, чтобы она была толще, то можем определить соответствующее значение в `Paint`:

```
Paint.setStrokeWidth(float widthInPixels);
```

Рисование прямоугольников. С помощью `Canvas` мы также можем рисовать прямоугольники:

```
Canvas.drawRect(float topleftX, float topleftY, float bottomRightX, float
bottomRightY, Paint paint);
```

Первые два аргумента определяют координаты левого верхнего угла прямоугольника, следующие два — координаты правого нижнего угла, `Paint` задает стиль и цвет прямоугольника. Что же понимается под стилем и как его определить?

Для установки стиля для экземпляра `Paint` используйте следующий метод:

```
Paint.setStyle(Style style);
```

`Style` — перечисление, имеющее значения `Style.FILL`, `Style.STROKE` и `Style.FILL_AND_STROKE`. Если мы назначим стиль `Style.FILL`, прямоугольник будет заполнен цветом, определенным в `Paint`. При использовании `Style.STROKE` будет прорисована только рамка прямоугольника (ее цвет и толщина также определены в `Paint`). Стиль `Style.FILL_AND_STROKE` объединяет два предыдущих — прямоугольник будет заполнен цветом, рамка будет нарисована.

Рисование окружностей. Окружности могут рисоваться как пустыми, так и заполненными:

```
Canvas.drawCircle(float centerX, float centerY, float radius, Paint paint);
```

Первые два аргумента определяют координаты центра окружности, следующий аргумент — ее радиус в пикселах, последний — все тот же `Paint`. Как и в случае с методом `Canvas.drawRectangle()`, цвет и стиль `Paint` будут использованы для рисования окружности.

Еще один важный момент, который стоит помнить, — все эти методы рисования выполняют альфа-смешивание. Просто определите альфа-составляющую цвета значением, отличающимся от 255 (0xff), и ваши пикселы, линии, прямоугольники и окружности станут полупрозрачными.

Все вместе

Напишем небольшую тестовую активность, демонстрирующую все вышеперечисленные методы. Я хочу, чтобы вы сначала проанализировали код в листинге 4.13. Представьте, что мы используем экран с разрешением 480 × 800 в портретном режиме, на котором будут рисоваться разные фигуры. При программировании графики очень важно представлять себе результат выполнения команд. Это требует некоторой практики, но оно того стоит.

Листинг 4.13. ShapeTest.java: Безумное рисование фигур

```
package com.badlogic.androidgames;

import android.app.Activity;
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Style;
import android.os.Bundle;
import android.view.View;
import android.view.Window;
import android.view.WindowManager;

public class ShapeTest extends Activity {
    class RenderView extends View {
        Paint paint;

        public RenderView(Context context) {
            super(context);
            paint = new Paint();
        }

        protected void onDraw(Canvas canvas) {
            canvas.drawRGB(255, 255, 255);
            paint.setColor(Color.RED);
            canvas.drawLine(0, 0, canvas.getWidth()-1, canvas.getHeight()-1,
                            paint);

            paint.setStyle(Style.STROKE);
            paint.setColor(0xff00ff00);
            canvas.drawCircle(canvas.getWidth() / 2, canvas.getHeight() / 2,
                              40, paint);

            paint.setStyle(Style.FILL);
            paint.setColor(0x770000ff);
```

```
            canvas.drawRect(100, 100, 200, 200, paint);
            invalidate();
        }
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                             WindowManager.LayoutParams.FLAG_FULLSCREEN);
        setContentView(new RenderView(this));
    }
}
```

Вы уже нарисовали в воображении картинку? Тогда давайте быстро проанализируем метод `RenderView.onDraw()`. Все остальное не изменилось с прошлого примера.

Начинаем с заполнения экрана белым цветом. Далее рисуем линию от начала до правого нижнего пиксела экрана. Мы используем `Paint`, чей цвет установлен в красный, поэтому линия будет такого же цвета.

Затем немного изменяем `Paint`, устанавливая его стиль как `Style.STROKE`, цвет — зеленый и значение альфа — 255. В центре экрана рисуется окружность с радиусом 40 пикселов (использующая `Paint`, который мы только что изменили). Из-за установленного стиля окружность будет незаполненной.

Последний блок начинается с еще одной модификации `Paint`. Мы устанавливаем его стиль `Style.FILL`, а цвет — синий. Обратите внимание — я задал альфа равной `0x77` в этот раз, что равно 119 в десятичном исчислении. Это означает, что фигура, которую мы рисуем, будет примерно на 50 % прозрачной.

Рисунок 4.13 показывает итог работы тестовой активности в портретном режиме в разрешениях 480×800 и 320×480.

Боже, что произошло? То, что случается при рисовании с абсолютными координатами и размерами на разных разрешениях. Единственный элемент, оставшийся неизменным на обеих картинках, — красная линия через весь экран (она реализована способом, не зависящим от разрешения).

Прямоугольнику установлено положение (100; 100). В зависимости от разрешения дисплея расстояния до центра экрана будет разным. Кроме того, размер прямоугольника задан равным 100×100 пикселов. Чем больше дисплей, тем меньше фигура будет занимать места относительно его площади.

Позиция центра окружности задана не зависящим от разрешения способом, а вот ее радиус — нет. Поэтому он тоже занимает на правом экране больше места, чем на левом.

Мы уже видели, что поддержка различных разрешений дисплея может быть проблемой. Дела становятся еще хуже, когда мы сталкиваемся с разными физическими размерами дисплеев. Мы попытаемся решить эти вопросы в следующей главе. Пока же помните, что означают разрешение экрана и его физический размер.

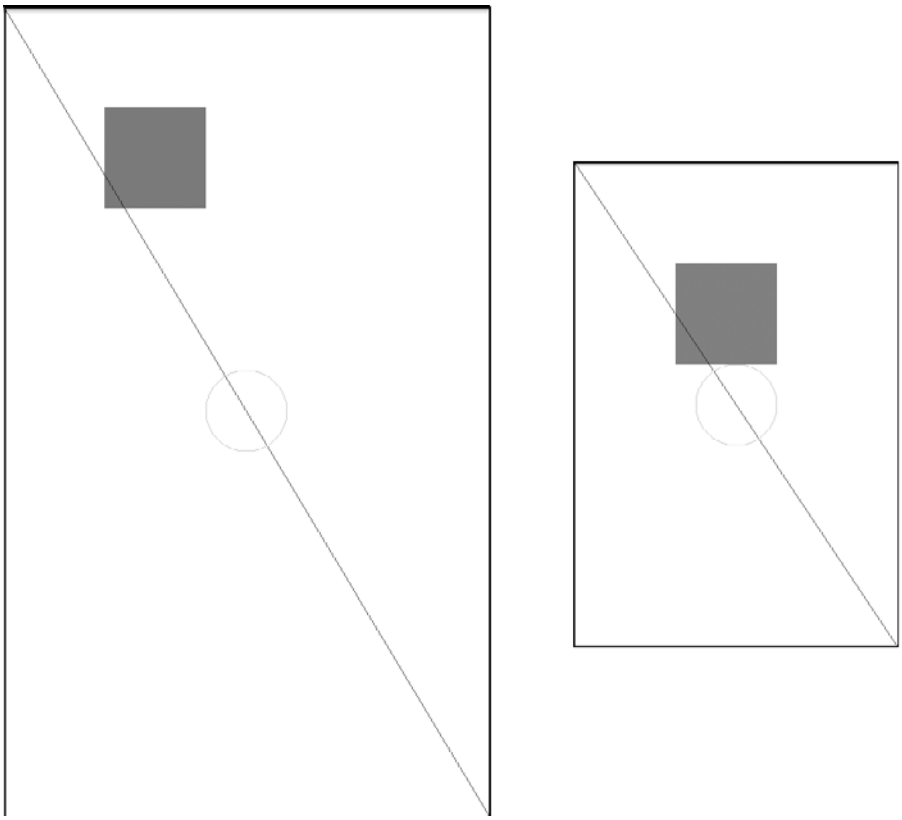

Рис. 4.13. Вывод ShapeTest на экране 480 × 800 *(слева)* и 320 × 480 *(справа)* (черная рамка добавлена позднее)

ПРИМЕЧАНИЕ

Классы Canvas и Paint предлагают гораздо больше, чем мы только что обсудили. На самом деле все стандартные View в Android используют для прорисовки этот API, так что можете представить, сколько там всего. Как всегда, читайте Android Developers для более подробной информации.

Использование битовых изображений

Хотя создание игр, состоящих только из базовых фигур, имеет право на жизнь, это не то, чего всем хочется. Нам нужен гениальный художник для создания спрайтов и фоновых изображений, которые потом сможем загружать из PNG- или JPEG-файлов. В Android все это делается легко.

Загрузка и проверка изображений. Класс `Bitmap` станет нашим лучшим другом. Мы загружаем битовое изображение из файла с помощью элемента `BitmapFactory`. Поскольку мы храним изображения в виде ресурсов, посмотрим, как мы можем загружать их из каталога `assets/`:

```
InputStream inputStream = assetManager.open("bob.png");
Bitmap bitmap = BitmapFactory.decodeStream(inputStream);
```

Класс Bitmap содержит несколько полезных для нас методов. Для начала мы хотим узнать его ширину и высоту в пикселах:

```
int width = bitmap.getWidth();
int height = bitmap.getHeight();
```

Затем нам полезно будет получить цветовую схему, в которой сохранен Bitmap:

```
Bitmap.Config config = bitmap.getConfig();
```

Bitmap.Config — перечисление со следующими значениями:

```
Config.ALPHA_8
Config.ARGB_4444
Config.ARGB_8888
Config.RGB_565
```

Из главы 3 вы должны помнить, что означают эти варианты. Если нет, то я настоятельно рекомендую перечитать пункт «Цифровое кодирование цветов» подраздела «Графика» раздела «Код: скучная рутина» главы 3.

Интересно, что в перечислении отсутствует формат RGB888. Формат PNG, в свою очередь, поддерживает только схемы ARGB8888, RGB888 и палитры. В какой цифровой формат будет загружен PNG типа RGB888?

Ответ — BitmapConfig.RGB_565. Это происходит автоматически для любого RGB888 PNG-файла, загружаемого через BitmapFactory. Причина — настоящий фреймбуфер большинства устройств Android работает с этим цветовым форматом. Загрузка изображения с большей глубиной может быть пустой тратой памяти — все равно эти пикселы будут преобразованы в RGB565 при финальном рендеринге.

Тогда зачем присутствует Config.ARGB_8888? Дело в том, что создание изображения может быть осуществлено процессором до его прорисовки во фреймбуфере. При использовании альфа-компоненты у нас также будет чуть большая глубина, чем при использовании Config.ARGB_4444, что может быть необходимо для некоторых случаев высококачественной обработки изображения.

ARGB8888 PNG-файл может быть загружен в Bitmap с конфигурацией Config.ARGB_8888. Оставшиеся два формата используются редко. Однако мы можем указать BitmapFactory попытаться загрузить изображение в определенном цветовом формате, даже если его исходный формат отличается от него.

```
InputStream inputStream = assetManager.open("bob.png");
BitmapFactory.Options options = new BitmapFactory.Options();
options.inPreferredConfig = Bitmap.Config.ARGB_4444;
Bitmap bitmap = BitmapFactory.decodeStream(inputStream, null, options);
```

Мы используем перегруженный метод BitmapFactory.decodeStream() для передачи указания в виде экземпляра BitmapFactory.Options декодировщику изображения. Мы можем определить желаемый цветовой формат экземпляра Bitmap через BitmapFactory.Options.inPreferredConfig, как показано ранее. В этом гипотетическом примере формат файла bob.png будет ARGB8888 PNG, и мы хотим заставить

BitmapFactory загрузить его и конвертировать в ARGB4444. Однако наше указание может быть проигнорировано.

Вы также можете создать пустой Bitmap следующим статическим методом:

```
Bitmap bitmap = Bitmap.createBitmap(int width, int height, Bitmap.Config config);
```

Вполне вероятно, что вы захотите вручную поработать с изображением, что называется, на лету. Класс Canvas работает с растровыми изображениями тоже:

```
Canvas canvas = new Canvas(bitmap);
```

Теперь вы можете изменять ваши изображения тем же способом, которым вы модифицировали содержимое View.

Уничтожение Bitmap. BitmapFactory может помочь нам уменьшить расход памяти при загрузке изображения. Как говорилось в главе 3, растры занимают много места в памяти. Уменьшение количества битов на пиксел за счет использования другого цветового формата помогает, но при загрузке одного изображения за другим мы в конечном итоге займем всю память. Поэтому мы должны уничтожать экземпляры Bitmap после потери необходимости в них с помощью этого метода:

```
Bitmap.recycle();
```

Рисование Bitmap. После загрузки наших изображений мы можем рисовать их с помощью Canvas. Самый простой способ это сделать выглядит так:

```
Canvas.drawBitmap(Bitmap bitmap, float topLeftX, float topLeftY, Paint paint);
```

Первый аргумент должен быть очевиден. Аргументы topLeftX и topLeftY определяют координаты размещения на экране левого верхнего угла изображения. Последний аргумент может принимать значение null. Мы могли бы определить дополнительные параметры рисования в классе Paint, но необходимости в этом нет.

Существует другая версия этого метода:

```
Canvas.drawBitmap(Bitmap bitmap, Rect src, Rect dst, Paint paint);
```

Это очень классный метод. Он позволяет нам определить через второй параметр часть изображения Bitmap, которую необходимо рисовать. Класс Rect хранит координаты верхнего левого и нижнего правого углов прямоугольника. Когда мы определяем часть изображения через переменную src, то делаем это в системе координат объекта Bitmap. Если мы укажем значение null, будет использован весь объект Bitmap. Третий параметр задает часть объекта Bitmap, в который должна происходить прорисовка, также в виде экземпляра Rect. Однако в данном случае координаты заданы относительно целевого объекта Canvas (либо View или другого Bitmap). Большой сюрприз состоит в том, что два этих прямоугольника не обязательно должны иметь одинаковые размеры. Если мы определим целевой прямоугольник с меньшим размером, чем исходный, Canvas автоматически масштабирует его (то же самое относится к случаю, когда целевой прямоугольник больше исходного). Последний параметр также обычно устанавливается в null. Хочу только

заметить, что операция масштабирования — весьма дорогое с точки зрения ресурсов удовольствие. Использовать его стоит только при крайней необходимости.

Вас может заинтересовать еще один вопрос. Если у нас есть экземпляры `Bitmap` с различными цветовыми форматами, необходимо ли нам конвертировать их в какой-нибудь стандартный формат перед прорисовкой в `Canvas`? Нет, `Canvas` сделает это автоматически. Конечно, использовать цветовые форматы, близкие к родным форматам фреймбуфера, будет немного быстрее. Однако обычно мы просто игнорируем этот шаг.

Смешивание также по умолчанию включено, так что если наши изображения содержат для своих пикселов альфа-компоненты, они будут обрабатываться.

Все вместе

Со всей полученной информацией мы теперь можем загружать и прорисовывать несколько Бобов. Листинг 4.14 содержит исходный код активности `BitmapTest`, который я написал для демонстрации.

Листинг 4.14. Активность `BitmapTest`

```
package com.badlogic.androidgames;

import java.io.IOException;
import java.io.InputStream;

import android.app.Activity;
import android.content.Context;
import android.content.res.AssetManager;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Rect;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.Window;
import android.view.WindowManager;

public class BitmapTest extends Activity {
    class RenderView extends View {
        Bitmap bob565;
        Bitmap bob4444;
        Rect dst = new Rect();

        public RenderView(Context context) {
            super(context);

            try {
                AssetManager assetManager = context.getAssets();
                InputStream inputStream =
                        assetManager.open("bobrgb888.png");
```

```
                bob565 = BitmapFactory.decodeStream(inputStream);
                inputStream.close();
                Log.d("BitmapText",
                        "bobrgb888.png format: " + bob565.getConfig());

                inputStream = assetManager.open("bobargb8888.png");
                BitmapFactory.Options options = new BitmapFactory.Options();
                options.inPreferredConfig = Bitmap.Config.ARGB_4444;
                bob4444 = BitmapFactory
                        .decodeStream(inputStream, null, options);
                inputStream.close();
                Log.d("BitmapText",
                        "bobargb8888.png format: " + bob4444.getConfig());

            } catch (IOException e) {
                // Ничего не делаем. На самом деле так нельзя.
            } finally {
                // здесь надо обязательно закрывать потоки
            }
        }

        protected void onDraw(Canvas canvas) {
            dst.set(50, 50, 350, 350);
            canvas.drawBitmap(bob565, null, dst, null);
            canvas.drawBitmap(bob4444, 100, 100, null);
            invalidate();
        }
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                WindowManager.LayoutParams.FLAG_FULLSCREEN);
        setContentView(new RenderView(this));
    }
}
```

Метод `onCreate()` изучен нами вдоль и поперек, поэтому сосредоточимся на нашей реализации `View`. Этот класс содержит два члена типа `Bitmap`: один из них хранит картинку Боба (представленного вам в главе 3) в формате RGB565, другой — Боба же в формате ARGB4444. Кроме того, у нас есть член типа `Rect`, в котором хранятся параметры целевого прямоугольника для рендеринга.

В конструкторе класса `RenderView` мы для начала загружаем Боба в объект `bob565`. Обратите внимание — изображение загружается из RGB888 PNG-файла, `BitmapFactory` автоматически конвертирует его в формат RGB565. Чтобы в этом убедиться, мы также выводим содержимое `Bitmap.Config` в **LogCat**. RGB888-версия Боба имеет прозрачный белый фон, поэтому нет необходимости производить смешивание.

Далее мы загружаем Боба из ARGB8888 PNG-файла, находящегося в каталоге `assets/`. Для экономии памяти мы также даем `BitmapFactory` команду преобразовать это изображение в формат ARGB4444. Напоминаю — `factory` может не подчиниться этой команде (по неясным причинам). Чтобы увидеть, прошла ли операция, мы также выводим содержимое `Bitmap.Config` в LogCat.

Метод `onDraw()` очень короткий. Все, что мы в нем делаем, — рисуем содержимое `bob565`, растянутое до 250 × 250 пикселов (оригинальный размер — 160 × 183 пиксела), а поверх него рисуем `bob4444`, не масштабируя, но смешивая (это делается автоматически объектом `Canvas`). Рисунок 4.14 демонстрирует нам Боба во всей красе.

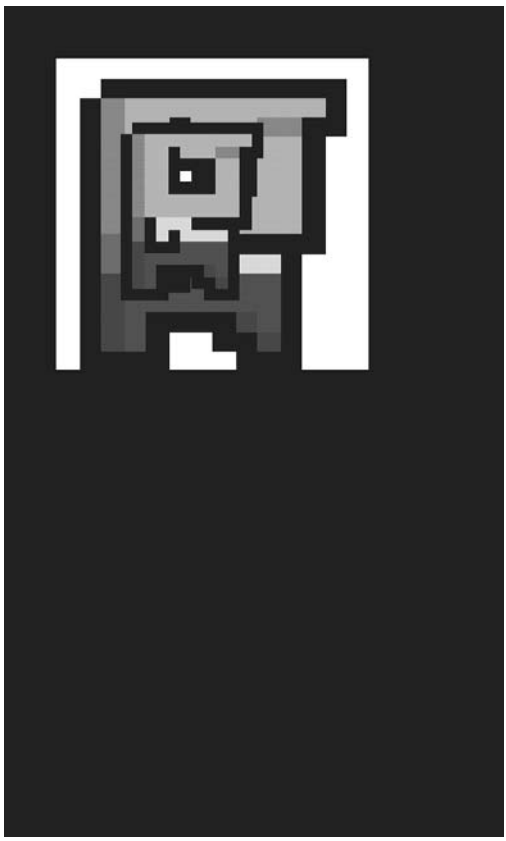

Рис. 4.14. Два Боба: один поверх другого (при разрешении 480 × 800)

LogCat сообщает, что `bob565` действительно имеет формат `Config.RGB_565`, а `bob4444` конвертирован в `Config.ARGB_444` — `BitmapFactory` не сплоховал!

Вот несколько моментов, на которые вы должны обратить внимание.

- Используйте минимально допустимый формат цвета для сохранения памяти, но помните, что ценой этой экономии может стать худшее качество визуализации и некоторое уменьшение скорости перерисовки.

- Без крайней необходимости не прибегайте к масштабированию изображений. Если вам известен необходимый размер картинки, измените исходное изображение заранее.
- Всегда вызывайте метод `Bitmap.recycle()`, если изображение вам больше не нужно. Иначе можете получить утечки памяти и ее нехватку.

Использование LogCat для вывода текста довольно утомительно. Изучим возможности прорисовки текста с помощью `Canvas`.

ПРИМЕЧАНИЕ

Как и в случае с другими классами, в Bitmap содержится намного больше, чем я смог описать на этих нескольких страницах. Я дал вам необходимый для написания «Мистера Нома» минимум. Если хотите большего, добро пожаловать на Android Developers.

Прорисовка текста

Хотя текст, выводимый в игре «Мистер Ном», будет нарисован от руки, весьма полезно будет узнать, как выводить текст с помощью шрифтов TrueType. Начнем с загрузки собственного TrueType-шрифта из папки `assets/`.

Загрузка шрифтов. Android API предлагает класс `Typeface`, предназначенный для хранения TrueType. Класс предоставляет простой статический метод для загрузки файла шрифта из каталога `assets/`:

```
Typeface font = Typeface.createFromAsset(context.getAssets(), "font.ttf");
```

Что интересно, данный метод не генерирует особого исключения, если файл шрифта не был по каким-либо причинам загружен. Вместо этого срабатывает исключение `RuntimeException`. Почему для данного метода не было описано конкретного исключения — для меня загадка.

Рисование текста с помощью шрифта. Теперь у нас есть собственный шрифт, и мы определяем его как `Typeface` нашего экземпляра `Paint`.

```
paint.setTypeFace(font);
```

С помощью `Paint` мы также определяем размер шрифта:

```
paint.setTextSize(30);
```

Документация по этому методу страдает лаконичностью — она, например, не сообщает, в каких единицах задается размер шрифта (в точках или пикселах). Мы примем за истину второй вариант.

Теперь мы можем вывести текст с использованием нужного нам шрифта с помощью следующего метода класса `Canvas`:

```
canvas.drawText("This is a test!", 100, 100, paint);
```

Первый параметр — текст, который необходимо вывести. Следующие два аргумента — координаты места, где это должно произойти. Последний аргумент хорошо нам знаком: это экземпляр `Paint`, определяющий цвет, шрифт и размер текста. Установив для `Paint` свой цвет, мы также можем раскрасить наш текст.

Выравнивание и границы текста. Вы может спросить, как координаты из предыдущего метода соотносятся с прямоугольником, в который помещается текст. Определяют ли они левый верхний угол этого прямоугольника? Ответ не так прост, как может показаться. Экземпляр `Paint` имеет атрибут, названный `align setting`. Он может быть установлен с помощью метода класса `Paint`:

```
Paint.setTextAlign(Paint.Align align);
```

Перечисление `Paint.Align` имеет три варианта значения: `Paint.Align.LEFT`, `Paint.Align.CENTER` и `Paint.Align.RIGHT`. В зависимости от установленного выравнивания интерпретируются координаты, передаваемые методу `Canvas.drawText()`, как левый верхний угол прямоугольника, верхний центральный пиксел прямоугольника или же его правый верхний угол. Значение выравнивания по умолчанию — `Paint.Align.LEFT`.

Иногда также полезно знать границы определенной строки, выраженные в пикселах. Для этого класс `Paint` также предлагает следующий метод:

```
Paint.getTextBounds(String text, int start, int end, Rect bounds);
```

Первый параметр — строка, для которой нам хочется получить границы. Следующие два аргумента задают начальную и конечную позиции внутри этой строки, для которых должны быть измерены границы. Последний параметр совершенно особый — это экземпляр класса `Rect`, который мы сами определяем и передаем методу. Метод запишет значения ширины и высоты ограничивающего прямоугольника в поля `Rect.right` и `Rect.bottom`. Для уверенности мы можем вызвать методы `Rect.width()` и `Rect.height()`, чтобы получить те же значения. Обратите внимание — все эти методы работают только для однострочного текста. Если мы хотим вывести многострочный текст, нам придется компоновать его самостоятельно.

Все вместе

Хватит разговоров — пора и запрограммировать что-нибудь. Листинг 4.15 показывает рендеринг текста в действии.

Листинг 4.15. Активность FontTest

```
package com.badlogic.androidgames;

import android.app.Activity;
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Rect;
import android.graphics.Typeface;
import android.os.Bundle;
import android.view.View;
import android.view.Window;
import android.view.WindowManager;

public class FontTest extends Activity {
```

```
    class RenderView extends View {
        Paint paint;
        Typeface font;
        Rect bounds = new Rect();

        public RenderView(Context context) {
            super(context);
            paint = new Paint();
            font = Typeface.createFromAsset(context.getAssets(),
                   "font.ttf");
        }

        protected void onDraw(Canvas canvas) {
            paint.setColor(Color.YELLOW);
            paint.setTypeface(font);
            paint.setTextSize(28);
            paint.setTextAlign(Paint.Align.CENTER);
            canvas.drawText("This is a test!", canvas.getWidth() / 2, 100,
                   paint);

            String text = "This is another test o_O";
            paint.setColor(Color.WHITE);
            paint.setTextSize(18);
            paint.setTextAlign(Paint.Align.LEFT);
            paint.getTextBounds(text, 0, text.length(), bounds);
            canvas.drawText(text, canvas.getWidth() - bounds.width(), 140,
                   paint);
            invalidate();
        }
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                WindowManager.LayoutParams.FLAG_FULLSCREEN);
        setContentView(new RenderView(this));
    }
}
```

Мы не будем обсуждать метод `onCreate()` данной активности, поскольку видели его уже не раз. Наша реализация `RenderView` включает три члена: `Paint`, `Typeface` и `Rect` (в котором мы позже сохраним границы нашего текста).

В конструкторе создаем новый экземпляр `Paint` и загружаем шрифт из файла `font.ttf` из каталога `assets/`.

В методе `onDraw()` устанавливаем для `Paint` желтый цвет, шрифт и размер текста, а также определяем выравнивание текста (которое будет использовано для интерпретации координат при вызове `Canvas.drawText()`). В итоге вызов метода `Canvas.drawText()`

приведет к появлению строки `This is a test!`, центрированной горизонтально с координатой 100 по оси *y*.

Для второго вызова процедуры рисования текста мы кое-что изменим: нам нужен выровненный по правому краю текст, расположенный у правого края дисплея. Мы могли бы сделать это, используя `Paint.Align.RIGHT` и координату *x* равной `Canvas.getWidth()` – 1. Вместо этого мы усложняем задачу (и заодно практикуемся в компоновке простого текста) — применим границы текста. Попутно изменим цвет и размер текста. На рис. 4.15 показан вывод данной активности.

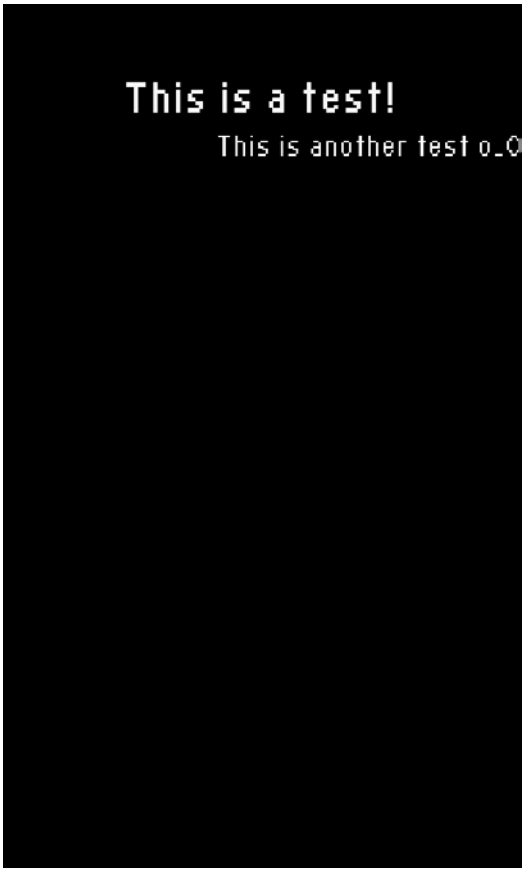

Рис. 4.15. Вариации текста (разрешение 480 × 800)

Еще одна загадка класса `Typeface` состоит в том, что он не предоставляет способ явно освободить свои ресурсы — нам приходится полагаться на сборщик мусора.

ПРИМЕЧАНИЕ

В этом разделе мы коснулись только поверхностного слоя темы прорисовки текста. Если вам нужно больше… я думаю, вы уже догадались, куда я вас отправлю.

Непрерывная прорисовка с SurfaceView

Этот пункт — для настоящих мужчин (и женщин). В нем говорится о потоках и всех неприятностях, с ними связанных. Но мы выйдем из него живыми, я обещаю!

Мотивация. Когда мы в первый раз осуществляли непрерывную прорисовку, то делали это неправильно. Интенсивно использовать пользовательский поток не стоит; нам необходимо решение, делающее всю грязную работу в отдельном потоке. Встречайте `SurfaceView`.

Как намекает его имя, класс `SurfaceView` наследуется от `View` и обрабатывает `Surface`, еще один класс Android API. Что такое `Surface`? Это абстракция необработанного буфера, используемого компоновщиком экрана для прорисовки определенного `View`. Компоновщик экрана — это властелин всего рендеринга в Android, ответственный за передачу всех пикселов графическому процессору. В некоторых случаях `Surface` может быть аппаратно ускорен, но мы не будем сильно задумываться над этим фактом. Все, что нам необходимо знать, что есть более прямой способ прорисовки элементов на экране.

Наша цель — обеспечить прорисовку в отдельном потоке, чтобы не трогать поток пользовательского интерфейса, занятый другими делами. Класс `SurfaceView` предлагает нам способ прорисовки в отдельном потоке.

SurfaceHolder и блокировка. Чтобы рисовать `SurfaceView` из отдельного от пользовательского потока, нам необходимо получить экземпляр класса `SurfaceHolder`, примерно так:

```
SurfaceHolder holder = surfaceView.getHolder();
```

`SurfaceHolder` — оболочка для `Surface`, проводящая для нас некоторые действия с помощью двух методов:

```
Canvas SurfaceHolder.lockCanvas();
SurfaceHolder.unlockAndPost(Canvas canvas);
```

Первый метод блокирует `Surface` для прорисовки и возвращает прекрасный экземпляр `Canvas`, который мы можем использовать. Второй метод вновь разблокирует `Surface` и проверяет, что прорисованный нами через `Canvas` контент выведен на экран. Мы будем применять эти два метода в нашем потоке прорисовки для получения `Canvas`, собственно рисования и вывода полученного изображения на экран. Объект `Canvas`, передаваемый в метод `SurfaceHolder.unlockAndPost()`, должен быть тем же самым, который мы получили от `SurfaceHolder.lockCanvas()`.

`Surface` не создается немедленно после инициализации `SurfaceView` — это делается асинхронно. `Surface` будет уничтожаться каждый раз при приостановке активности и создаваться вновь при ее возобновлении.

Создание и проверка Surface. Мы не можем получить `Canvas` от `SurfaceHolder` до тех пор, пока `Surface` не является корректным. У нас есть возможность проверить факт его создания с помощью следующего выражения:

```
boolean isCreated = surfaceHolder.getSurface().isValid();
```

Если метод возвращает `true`, мы можем без опаски блокировать `Surface` и рисовать в нем с помощью полученного `Canvas`. Нам необходимо быть абсолютно

уверенными, что Surface вновь разблокирован после вызова SurfaceHolder.lockCanvas(), иначе наша активность может заблокировать телефон!

Все вместе

Итак, как нам теперь объединить все это с отдельным потоком прорисовки и жизненным циклом активности? Лучший способ представить это — посмотреть на реальный пример кода. В листинге 4.16 представлен полный пример, выполняющий прорисовку в отдельном потоке при помощи SurfaceView.

Листинг 4.16. Активность SurfaceViewTest

```
package com.badlogic.androidgames;
import android.app.Activity;
import android.content.Context;
import android.graphics.Canvas;
import android.os.Bundle;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
import android.view.Window;
import android.view.WindowManager;

public class SurfaceViewTest extends Activity {
    FastRenderView renderView;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                             WindowManager.LayoutParams.FLAG_FULLSCREEN);
        renderView = new FastRenderView(this);
        setContentView(renderView);
    }

    protected void onResume() {
        super.onResume();
        renderView.resume();
    }

    protected void onPause() {
        super.onPause();
        renderView.pause();
    }

    class FastRenderView extends SurfaceView implements Runnable {
        Thread renderThread = null;
        SurfaceHolder holder;
        volatile boolean running = false;

        public FastRenderView(Context context) {
            super(context);
            holder = getHolder();
        }
```

```
        public void resume() {
            running = true;
            renderThread = new Thread(this);
            renderThread.start();
        }

        public void run() {
            while(running) {
                if(!holder.getSurface().isValid())
                    continue;

                Canvas canvas = holder.lockCanvas();
                canvas.drawRGB(255, 0, 0);
                holder.unlockCanvasAndPost(canvas);
            }
        }

        public void pause() {
            running = false;
            while(true) {
                try {
                    renderThread.join();
                } catch (InterruptedException e) {
                    // повтор
                }
            }
        }
    }
}
```

Выглядит пугающе, не правда ли? Наша активность содержит экземпляр `FastRenderView` в качестве члена. Это наш подкласс `SurfaceView`, предназначенный для обработки всех потоковых операций и блокировки `Surface`. С точки зрения активности — это простой `View`. В методе `onCreate()` мы включаем полноэкранный режим, создаем экземпляр `FastRenderView` и устанавливаем его в качестве контейнера содержимого активности.

Кроме того, в этот раз мы переопределяем метод `onResume()`. В нем стартуем наш поток прорисовки косвенным образом — вызываем метод `FastRenderView.resume()`, делающий эту работу незаметно для нас. Это значит, что поток запустится, когда активность создается впервые (поскольку вслед за `onCreate()` всегда вызывается `onResume()`). Поток также перезапускается при каждом возвращении активности из состояния паузы. Безусловно, это подразумевает, что где-то поток должен останавливаться, иначе у нас при каждом вызове `onResume()` будет создаваться еще один новый поток. Для этого мы используем метод `onPause()` — он вызывает метод `FastRenderView.pause()`, полностью останавливающий поток. Метод не будет заканчивать свою работу, пока поток действительно не остановится.

Взглянем на ключевой класс этого примера: `FastRenderView`. Он похож на классы `RenderView`, реализованные нами в нескольких прошлых примерах, тем, что наследуется от другого класса `View`. В данном случае мы наследуем его напрямую

от SurfaceView и реализуем в нем интерфейс Runnable, чтобы иметь возможность передавать его потоку прорисовки и использовать его логику.

Класс FastRenderView имеет три члена. renderThread — просто ссылка на экземпляр Thread, ответственный за выполнение логики потока прорисовки. Переменная holder — ссылка на экземпляр SurfaceHolder, полученный нами от базового класса SurfaceView. Наконец, член running — простой логический флаг, используемый нами для сообщения потоку прорисовки о том, что он должен остановиться. Модификатор volatile имеет специальное назначение, о котором мы поговорим чуть позже. В конструкторе мы лишь вызываем конструктор базового класса и сохраняем ссылку на SurfaceHolder в переменной класса.

Теперь приходит время метода FastRenderView.resume(). Он ответственен за запуск потока прорисовки. Обратите внимание — мы создаем новый экземпляр потока каждый раз при вызове этого метода. Это соответствует тому, что мы говорили о методах активности onResume() и onPause(). Помимо этого мы устанавливаем флаг running в значение true (чуть позже вы увидите, как он используется в потоке прорисовки). Последнее, что следует уточнить, — мы установили экземпляр FastRenderView реализующим интерфейс Runnable, что позволит выполнить следующий метод FastRenderView в этом новом потоке.

Метод FastRenderView.run() выполняет основную работу для нашего класса View. Его тело выполняется в потоке прорисовки. Как видите, он состоит из одного цикла, прекращающего свое выполнение при установке флага running в false. Если это происходит, поток останавливается и уничтожается. Внутри цикла while мы сначала проверяем валидность Surface и при положительном результате блокируем его, рисуем в нем и вновь разблокируем (как говорилось ранее). В данном примере мы просто заполняем весь Surface красным цветом.

Метод FastRenderView.pause() выглядит немного странно. В его начале мы устанавливаем флаг running равным false (это сигнал для метода FastRenderView.run() остановить обработку и закрыть поток). Следующие несколько строк — ожидание полного уничтожения потока, осуществляемое вызовом метода Thread.join(). Этот метод ждет уничтожения потока, но может также вызвать исключение InterruptedException до того, как поток на самом деле сгинет. Поскольку нам необходимо быть абсолютно уверенными в уничтожении потока, прежде чем возвращаться из этого метода, мы выполняем Thread.join() в безусловном цикле до тех пор, пока операция не будет завершена успешно.

Вернемся к модификатору volatile флага running. Для чего он нужен? Причина довольно тонкая: компилятор может решить переопределить порядок выражений в методе FastRenderView.pause(), если распознает отсутствие зависимости между первой строкой метода и блоком while. У него есть такое право, если он считает, что это приведет к ускорению выполнения кода. Однако в данном случае порядок команд для нас весьма важен. Представьте, что мы установили флаг running после попытки присоединиться к потоку. Мы получим бесконечный цикл, поскольку поток никогда не будет уничтожен.

Модификатор volatile предотвращает эту коллизию. Любые выражения, связанные с переменными, которые обладают этим модификатором, будут выполнены

в установленном порядке. Это уберегает нас от скрытой ошибки, которую невозможно будет воспроизвести.

Осталась еще одна вещь, о которой вы могли задуматься как о потенциальном баге. Что произойдет, если `Surface` будет уничтожен между вызовами `SurfaceHolder.getSurface().isValid()` и `SurfaceHolder.lock()`? К счастью, нам повезло — такого никогда не произойдет. Что понять почему, нам нужно вернуться к теме жизненного цикла активности и его связи с `Surface`.

Мы знаем, что `Surface` создается асинхронно. Похоже, что наш поток прорисовки может выполниться до того, как `Surface` станет валидным. Защита от этого казуса состоит в том, что мы не блокируем `Surface` до тех пор, пока он не станет валидным, что решает проблему его создания.

Причина, по которой код потока прорисовки не вызывает ошибки при уничтожении `Surface` между проверкой валидности и блокировкой, связана с моментом времени, когда `Surface` уничтожается. Это всегда происходит после возвращения из метода `onPause()` активности. И поскольку в этом методе мы ожидаем уничтожения потока с помощью `FastRenderView.pause()`, поток прорисовки уже точно не будет «живым» при уничтожении `Surface`. Здорово, не так ли? Однако тут легко можно сбиться с толку.

Теперь мы правильно выполняем непрерывную прорисовку. Мы не загружаем поток пользовательского интерфейса, используя вместо этого отдельный поток для рисования. Кроме того, мы учитываем при этом фазы жизненного цикла активности, поэтому он не работает в фоновом режиме (что бережет аккумулятор при постановке активности на паузу). Конечно, нам еще нужно синхронизировать обработку событий ввода в пользовательском потоке с нашим потоком прорисовки. Но это будет несложно сделать — вы убедитесь в этом в следующей главе, когда мы будем реализовывать нашу игровую среду, основываясь на всей полученной в этой главе информации.

Полезные советы

Android (или, скорее, Dalvik) иногда имеет некоторые странные показатели производительности. Чтобы закруглиться с этой главой, я дам вам некоторые полезные и важные подсказки, которым стоит следовать, чтобы сделать ваши игры удобными.

- Сборщик мусора — ваш злейший враг. Когда он получает процессорное время для выполнения своей грязной работы, все другие процессы останавливаются на 600 миллисекунд. В течение этой половины секунды наша игра не будет обновляться и прорисовываться. Пользователям это не понравится. Избегайте создания объектов везде, где это возможно (особенно во внутренних циклах).
- Объекты могут создаваться в не самых очевидных местах, а этого вам хотелось бы избежать. Не пользуйтесь итераторами — они создают новые объекты. Не применяйте классы из стандартных коллекций `Set` или `Map` — они создают новые объекты при каждой вставке. Вместо этого задействуйте класс `SparseArray`, предлагаемый Android API. Используйте `StringBuffer` вместо склеивания строк

оператором «+» (это действие каждый раз неявно создает объект `StringBuffer`). И ради всего святого — не применяйте обертки над примитивными объектами!

- Вызовы методов имеют более высокую ассоциированную стоимость в Dalvik, чем в других виртуальных машинах. По возможности используйте статические методы — их производительность выше. Статические методы часто считаются злом (как и статические переменные), поскольку способствуют плохому дизайну кода. Поэтому старайтесь обеспечить четкую и ясную картину вашего кода. Кроме того, следует отказаться от использования свойств с геттерами и сеттерами — прямой доступ к полям в три раза быстрее, чем использование вызовов метода без JIT, и в семь раз быстрее — чем вместе с JIT. Тем не менее перед тем, как убирать свойства, еще раз подумайте о дизайне кода.
- Операции с плавающей точкой на старых устройствах и версиях Dalvik без JIT (то есть везде, где стоит Android ранее версии 2.2) выполняются программно. Разработчики старой школы при этих словах сразу возвращаются к математике с фиксированной точкой. Не обольщайтесь — целочисленные операции деления тоже не слишком быстры. Однако в большинстве случаев вы спокойно сможете оперировать данными типа `float`, а новые устройства поддерживают модули плавающей точки (FPU), ускоряющие работу.

  Старайтесь хранить часто используемые значения в локальных переменных внутри метода. Доступ к локальным переменным осуществляется намного быстрее, чем доступ к членам класса или применение свойств.

Конечно, существует еще много подводных камней, с которыми нужно быть осторожными. Далее в книге я буду давать и другие небольшие подсказки по производительности. Если вы будете следовать этим рекомендациям, у вас все будет хорошо. Только не дайте победить сборщику мусора!

Подводя итог

Эта глава содержит все, что вам необходимо для создания небольшой приличной игры для Android. Мы рассмотрели в ней вопросы создания нового игрового проекта, обсудили загадочный жизненный цикл активности и выяснили, как ему соответствовать. Мы занимались событиями касаний экрана (в том числе множественных), обработкой событий клавиатуры и проверяли положение устройства с помощью акселерометра. Мы узнали, как читать и записывать файлы. Вывод звука в Android оказался детской забавой, как и рисование на экране (если не считать организации потоков с помощью `SurfaceView`). Мистер Ном скоро станет реальностью — ужасной, голодной реальностью!

5 Android Game Development Framework

Мы с вами прошли уже четыре главы и до сих пор не написали ни единой строчки кода самой игры. Основная причина того, что вам пришлось изучить всю эту скучную теорию и провести несколько тестов программ, такова: если вы хотите писать игры, вам надо точно знать, как конкретно это работает. Конечно, вы можете скопировать и вставить код, который вы найдете где-то на просторах Интернета, и надеяться, что каким-то волшебным способом он превратится в новый топовый шутер от первого лица. Но гораздо лучше иметь твердые знания о том, как создать простую игру с нуля, как структурировать качественный API для 2D-программ, и о том, какие возможности предоставляют API Android для того, чтобы претворить ваши идеи в жизнь.

Чтобы сделать мистера Нома реальностью, нам необходимы всего две вещи: реализовать фреймворк интерфейсов и классов, который мы создали в главе 3, и, основываясь на этом, написать код механики игры «Мистер Ном». Начнем с фреймворка, объединив то, что мы создали в главе 3, с тем, что мы обсудили в главе 4. Вам должно быть уже знакомо 90 % кода, так как мы использовали большую часть его во время тестов в прошлой главе.

План атаки

В главе 3 мы создали минимальный и очень простой дизайн для фреймворка игры, который принимает во внимание все особенности платформы, поэтому сконцентрируемся здесь на написании самой игры. Реализуем все наши интерфейсы и абстрактные классы снизу вверх — от простого к сложному. Интерфейсы из главы 3 расположены в пакете `com.badlogic.androidgames.framework`. Мы размещаем нашу разработку в пакете `com.badlogic.androidgames.framework.impl`. Это показывает, что именно в этом пакете находится актуальная реализация фреймворка для Android. Мы помечаем все наши реализации интерфейса словом `Android`, чтобы было легче отличить их от других интерфейсов. Начнем с самой простой части: файлового ввода-вывода.

Программный код из этой и следующей глав будет объединен в один проект Eclipse. Пока просто создайте проект в Eclipse так, как об этом было рассказано выше. Название проекта на данный момент не имеет значения.

Класс AndroidFileIO

Начальный интерфейс FileIO был достаточно скромным. В нем было всего лишь три метода: один, чтобы получить InputStream для assets, еще один, чтобы получить входной поток InputStream к файлу на внешнем диске, и третий, возвращающий OutputStream файлу на внешнем диске. В главе 4 было рассказано, как можно открывать ресурсы (assets) и файлы из внешнего хранилища с помощью API Android. В листинге 5.1 продемонстрирована реализация интерфейса FileIO, для работы с которым мы воспользуемся материалом, изученным в главе 4.

Листинг 5.1. AndroidFileIO.java; реализация интерфейса FileIO

```
package com.badlogic.androidgames.framework.impl;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import android.content.res.AssetManager;
import android.os.Environment;

import com.badlogic.androidgames.framework.FileIO;

public class AndroidFileIO implements FileIO {
    AssetManager assets;
    String externalStoragePath;

    public AndroidFileIO(AssetManager assets) {
        this.assets = assets;
        this.externalStoragePath = Environment.getExternalStorageDirectory()
                .getAbsolutePath() + File.separator;
    }

    @Override
    public InputStream readAsset(String fileName) throws IOException {
        return assets.open(fileName);
    }

    @Override
    public InputStream readFile(String fileName) throws IOException {
        return new FileInputStream(externalStoragePath + fileName);
    }

    @Override
    public OutputStream writeFile(String fileName) throws IOException {
        return new FileOutputStream(externalStoragePath + fileName);
    }
}
```

Все делается напрямую. Мы реализуем интерфейс FileIO, сохраняем AssetManager вместе с путем к внешнему диску и реализуем три метода на базе этих параметров. Кроме того, мы передаем все исключения IOException, которые могут быть выброшены системой, — так мы сразу узнаем обо всех подозрительных событиях, которые могут произойти на стороне, откуда приходит вызов.

Наша реализация интерфейса Game хранит экземпляр этого класса и возвращает его через Game.getFileIO(). Это также значит, что нашей реализации интерфейса Game нужно будет передать AssetManager, чтобы впоследствии AndroidFileIO мог работать.

Обратите внимание: мы не проверяем, доступно ли внешнее хранилище. Если оно недоступно или мы забыли добавить соответствующее разрешение в манифест (файл описания приложения), мы получим исключение. Соответственно, появится сообщение об ошибке. Таким образом, можно переходить к следующей части фреймворка — работе со звуком.

AndroidAudio, AndroidSound и AndroidMusic — все о звуке!

В главе 3 мы создали три интерфейса: Audio, Sound и Music — для всех наших нужд. С помощью Audio из файлов ресурсов (assets) создаются экземпляры классов Sound и Music.

Sound позволяет проигрывать звуковые эффекты, полностью хранящиеся в оперативной памяти, Music воспроизводит большие по размеру звуковые файлы, хранящиеся на диске. Из главы 4 вы узнали, что требуется API android, чтобы решить эти задачи. Начнем с реализации AndroidAudio так, как это показано в листинге 5.2.

Листинг 5.2. AndroidAudio.java; реализация аудиоинтерфейса

```
package com.badlogic.androidgames.framework.impl;

import.java.io.IOException;

import android.app.Activity;
import android.content.res.AssetFileDescriptor;
import android.content.res.AssetManager;
import android.media.AudioManager;
import android.media.SoundPool;

import com.badlogic.androidgames.framework.Audio;
import com.badlogic.androidgames.framework.Music;
import com.badlogic.androidgames.framework.Sound;

public class AndroidAudio implements Audio {
    AssetManager assets;
    SoundPool soundPool;
```

Реализация AndroidAudio содержит AssetManager и экземпляр SoundPool. Нам нужен AssetManager для того, чтобы мы могли загрузить звуковой эффект из файлов ресурсов (assets) в SoundPool при обращении к AndroidAudio.newSound(). В свою очередь, для управления SoundPool используется экземпляр класса AndroidAudio.

```
public AndroidAudio(Activity activity) {
    activity.setVolumeControlStream(AudioManager.STREAM_MUSIC);
    this.assets = activity.getAssets();
    this.soundPool = new SoundPool(20, AudioManager.STREAM_MUSIC, 0);
}
```

В конструкторе мы передаем Activity нашей игры, это делается по двум причинам: так мы можем установить регулятор звука медиапотока (а нам это в любом случае понадобится), а также получаем AssetManager, который мы сохраняем в соответствующем члене класса. SoundPool создан таким образом, что он способен проигрывать одновременно до 20 звуковых эффектов. Этого более чем достаточно для решения стоящей перед нами задачи.

```
    @Override
    public Music newMusic(String filename) {
        try {
            AssetFileDescriptor assetDescriptor = assets.openFd(filename);
            return new AndroidMusic(assetDescriptor);
        } catch (IOException e) {
            throw new RuntimeException("Невозможно загрузить музыку '" +
                                       filename + "'");
        }
    }
```

Метод newMusic() создает новый экземпляр класса AndroidMusic. Конструктор данного класса использует AssetFileDescriptor, из которого внутрисистемно создается MediaPlayer (позже мы рассмотрим этот вопрос подробнее). Метод AssetManager.openFd() вызовет исключение IOException, если что-то пойдет не так. Мы примем это исключение и повторно выдадим его как RuntimeException. Вы спросите, почему бы не оставить IOException для того, чтобы сообщить об ошибке вызывающей стороне? Прежде всего, так работающий код станет гораздо более запутанным, так что лучше используем RuntimeException, которое не требуется специально перехватывать. Мы загружаем музыку из файла assets. Единственный случай, когда может возникнуть сбой, — если мы забудем добавить музыкальный файл в папку assets/ или если внутри нашего файла будут содержаться фиктивные байты. В такой ситуации могут возникать неисправимые ошибки, поскольку нам нужен экземпляр класса Music, чтобы наша игра правильно функционировала. Чтобы не попадать в такие ситуации, мы используем стратегию замены проверяемых исключений на RuntimeException еще в нескольких точках игрового фреймворка.

```
    @Override
    public Sound newSound(String filename) {
        try {
            AssetFileDescriptor assetDescriptor = assets.openFd(filename);
```

```
            int soundId = soundPool.load(assetDescriptor, 0);
            return new AndroidSound(soundPool, soundId);
        } catch (IOException e) {
            throw new RuntimeException("Невозможно загрузить звук '" +
                                       filename + "'");
        }
    }
}
```

В итоге метод `newSound()` загружает звуковой эффект из `assets` в `SoundPool` и возвращает экземпляр класса `AndroidSound`. Конструктор данного экземпляра класса принимает `SoundPool` и ID звукового эффекта — этот идентификатор ему присвоил `SoundPool`. Мы снова заменяем перехваченные исключения на `RuntimeException`.

ПРИМЕЧАНИЕ

Мы не освобождаем SoundPool ни в одном из методов. Причина этого заключается в том, что у нас в любом случае будет экземпляр класса Game, содержащий один экземпляр класса Audio, который, в свою очередь, включает в себя один экземпляр класса SoundPool. SoundPool будет существовать до тех пор, пока существует игра. Он автоматически удалится, как только активность закончит работу.

Далее обсудим класс `AndroidSound`, который реализует интерфейс `Sound` (листинг 5.3).

Листинг 5.3. AndroidSound.java: Реализация интерфейса Sound

```
package com.badlogic.androidgames.framework.impl;

import android.media.SoundPool;

import com.badlogic.androidgames.framework.Sound;

public class AndroidSound implements Sound {
    int soundId;
    SoundPool soundPool;

    public AndroidSound(SoundPool soundPool, int soundId) {
        this.soundId = soundId;
        this.soundPool = soundPool;
    }

    @Override
    public void play(float volume) {
        soundPool.play(soundId, volume, volume, 0, 0, 1);
    }

    @Override
    public void dispose() {
        soundPool.unload(soundId); }
    }
}
```

Ничего сверхъестественного. Мы просто сохранили SoundPool и ID загруженных звуковых эффектов для их последующего воспроизведения и утилизации с помощью методов play() и dispose(). Проще не бывает. Все это — благодаря удобству самого API Android.

И наконец, нам необходимо реализовать класс AndroidMusic, возвращаемый AndroidAudio.newMusic().

В листинге 5.4 приведен код для этого класса. Он выглядит немного более сложным, чем предыдущие примеры. Это объясняется применением машины состояний, которую использует MediaPlayer. Она сгенерирует кучу исключений, если мы будем вызывать методы в тех состояниях, когда этого делать не следует.

Листинг 5.4. AndroidMusic.java: реализация интерфейса Music

```
package com.badlogic.androidgames.framework.impl;

import java.io.IOException;

import android.content.res.AssetFileDescriptor;
import android.media.MediaPlayer;

import android.media.MediaPlayer.OnCompletionListener;

import com.badlogic.androidgames.framework.Music;

public class AndroidMusic implements Music, OnCompletionListener {
    MediaPlayer mediaPlayer;
    boolean isPrepared = false;
    package com.badlogic.androidgames.framework.impl;

import java.io.IOException;

import android.content.res.AssetFileDescriptor;
import android.media.MediaPlayer;
import android.media.MediaPlayer.OnCompletionListener;

import com.badlogic.androidgames.framework.Music;

public class AndroidMusic implements Music, OnCompletionListener {
    MediaPlayer mediaPlayer;
    boolean isPrepared = false;
```

Класс AndroidMusic сохраняет экземпляр класса MediaPlayer вместе с булевым значением isPrepared. Обратите внимание, мы можем использовать MediaPlayer.start()/stop()/pause(), только когда MediaPlayer готов (prepared). Этот член позволяет отслеживать состояние MediaPlayer.

Класс AndroidMusic реализует не только интерфейс Music, но также интерфейс OnCompletionListener. В главе 3 мы кратко описали данный интерфейс как удобный инструмент, всегда позволяющий узнать, что MediaPlayer прекратил воспроизводить музыкальный файл. Если это случится, следует заново подготовить MediaPlayer,

так как до этого мы не можем вызывать никаких методов. Метод OnCompletionListener.onCompletion() может быть использован в другом потоке, и, поскольку мы устанавливаем член isPrepared в данном методе, необходимо убедиться, что к нему не будут применяться противоречащие друг другу изменения.

```
public AndroidMusic(AssetFileDescriptor assetDescriptor) {
    mediaPlayer = new MediaPlayer();
    try {
        mediaPlayer.setDataSource(assetDescriptor.getFileDescriptor(),
                assetDescriptor.getStartOffset(),
                assetDescriptor.getLength());
        mediaPlayer.prepare();
        isPrepared = true;
        mediaPlayer.setOnCompletionListener(this);
    } catch (Exception e) {
        throw new RuntimeException("Couldn't load music");
    }
}
```

В конструкторе создаем и готовим MediaPlayer из AssetFileDescriptor, устанавливаем флаг isPrepared, а также регистрируем экземпляр класса AndroidMusic в качестве слушателя (приемника) OnCompletionListener в MediaPlayer. Если что-то пойдет не так, будет сгенерировано непроверяемое исключение RuntimeException.

```
@Override
public void dispose() {

    if (mediaPlayer.isPlaying())
        mediaPlayer.stop();
    mediaPlayer.release();
}
```

Метод dispose() сначала проверяет, по-прежнему ли работает MediaPlayer, и, если воспроизведение продолжается, останавливает его. В противном случае обращение к MediaPlayer.release() вызовет исключение времени выполнения.

```
@Override
public boolean isLooping() {
    return mediaPlayer.isLooping();
}

@Override
public boolean isPlaying() {
    return mediaPlayer.isPlaying();
}

@Override
public boolean isStopped() {
    return !isPrepared;
}
```

Методы isLooping(), isPlaying() и isStopped() довольно просты. Первые два используют методы, предоставленные MediaPlayer; последний применяет флаг isPrepared, который показывает, что MediaPlayer остановлен. Например, MediaPlayer.isPlaying() не может предоставить нам точную информацию по этому вопросу, так как он не различает, остановлен MediaPlayer или же просто поставлен на паузу.

```
@Override
public void play() {
    if (mediaPlayer.isPlaying())
        return;

    try {
        synchronized (this) {
            if (!isPrepared)
                mediaPlayer.prepare();
            mediaPlayer.start();
        }
    } catch (IllegalStateException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Теперь к работе подключается метод play(). Если воспроизведение уже идет, то просто выполняем возврат из функции. Далее используем универсальный блок try...catch, который сначала проверяет, готов ли MediaPlayer (это можно узнать по значению флага), и подготавливает его, если это требуется. Если ошибок нет, обращаемся к методу MediaPlayer.start(), который воспроизводит музыку. Все это делается в синхронизированном блоке, поскольку мы используем флаг isPrepared, который благодаря интерфейсу OnCompletionListener может быть установлен в отдельном потоке. Если возникнут какие-то проблемы, опять-таки будет сгенерировано непроверяемое исключение RuntimeException.

```
@Override
    public void setLooping(boolean isLooping) {
        mediaPlayer.setLooping(isLooping);
    }

    @Override
    public void setVolume(float volume) {
        mediaPlayer.setVolume(volume, volume);
    }
```

Методы setLooping() и setVolume() могут быть использованы при любом состоянии MediaPlayer. Они просто служат делегатами соответствующих методов MediaPlayer.

```
@Override
    public void stop() {
        mediaPlayer.stop();
```

```
        synchronized (this) {
            isPrepared = false;
        }
    }
```

Метод `stop()` останавливает `MediaPlayer` и снова устанавливает флаг `isPrepared` в синхронизированном блоке.

```
@Override
    public void onCompletion(MediaPlayer player) {
        synchronized (this) {
            isPrepared = false;
        }
    }
}
```

И последний метод в реализации `AndroidMusic` — `OnCompletionListener.onCompletion()`. Он просто устанавливает флаг `isPrepared` в синхронизированном блоке, чтобы другие методы не выдавали ненужные исключения.

Теперь перейдем к классам, обслуживающим ввод информации.

AndroidInput и AccelerometerHandler

Интерфейс `Input`, который мы создали в главе 3, с помощью нескольких удобных методов предоставляет нам доступ к акселерометру, сенсорному экрану и клавиатуре в режиме опроса и режиме событий. Помещать весь код для реализации данного интерфейса в один файл немного неудобно, поэтому мы вынесем всю обработку событий ввода в отдельные обработчики. Реализация `Input` будет использовать данные обработчики, чтобы создавать видимость того, что именно она выполняет всю работу.

AccelerometerHandler: что сверху?

Начнем с самого простого из всех обработчиков — с `AccelerometerHandler` (листинг 5.5).

Листинг 5.5. `AccelerometerHandler.java`; выполнение обработки, связанной с акселерометром

```
package com.badlogic.androidgames.framework.impl;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;

public class AccelerometerHandler implements SensorEventListener {
    float accelX;
```

```
    float accelY;
    float accelZ;

    public AccelerometerHandler(Context context) {
        SensorManager manager = (SensorManager) context
                .getSystemService(Context.SENSOR_SERVICE);
        if (manager.getSensorList(Sensor.TYPE_ACCELEROMETER).size() != 0) {
            Sensor accelerometer = manager.getSensorList(
                    Sensor.TYPE_ACCELEROMETER).get(0);
            manager.registerListener(this, accelerometer,
                    SensorManager.SENSOR_DELAY_GAME);
        }
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // Здесь ничего не делаем
    }

    @Override
    public void onSensorChanged(SensorEvent event) {
        accelX = event.values[0];
        accelY = event.values[1];
        accelZ = event.values[2];
    }

    public float getAccelX() {
        return accelX;
    }

    public float getAccelY() {
        return accelY;
    }

    public float getAccelZ() {
        return accelZ;
    }
}
```

Неудивительно, что данный класс применяет интерфейс `SensorEventListener`, которым мы пользовались в главе 4. Этот класс хранит три переменные, запоминая информацию об ускорении по каждой из трех осей акселерометра.

Конструктор принимает `Context`, из которого получает экземпляр класса `SensorManager`, чтобы начать слушать события. Оставшаяся часть кода эквивалентна той, с которой мы работали в прошлой главе. Обратите внимание, что если акселерометр отсутствует, то обработчик будет все время считать, что ускорение по всем осям равно нулю, а нам ни к чему дополнительная проверка на наличие ошибок или код для генерации исключений.

Следующие два метода — `onAccuracyChanged()` и `onSensorChanged()` — также должны быть вам уже знакомы. В первом мы в данном случае ничего не делаем, так как

здесь не о чем сообщать. Используя второй метод, мы выбираем данные акселерометра из принятого SensorEvent и сохраняем их в членах обработчика.

Последние три метода возвращают текущие значения ускорения по каждой из осей.

Обратите внимание, что мы не занимаемся здесь какой-либо синхронизацией, даже несмотря на то, что метод onSensorChanged() может быть реализован в отдельном потоке. Модель памяти Java гарантирует, что операции записи и считывания из примитивных типов данных, таких как boolean, int или byte, являются атомарными. В данном случае можно положиться на этот факт, так как мы не делаем ничего сложного, а просто присваиваем новое значение. В другом случае, например, если бы мы оперировали переменными экземпляра в методе onSensorChanged(), нам понадобилась бы хорошая синхронизация.

Класс Pool: используем повторно!

Какая самая ужасная вещь может случиться при программировании под Android? Несомненно, это всеостанавливающая сборка мусора! Если обратить внимание на определение интерфейса Input в главе 3, то сразу будут заметны методы getTouchEvents() и getKeyEvents(). Они возвращают списки TouchEvents и KeyEvents. В наших обработчиках событий клавиатуры и сенсорного экрана мы будем постоянно создавать экземпляры этих двух классов и хранить их во внутренних списках обработчиков. Система ввода Android постоянно генерирует множество событий при касании экрана или нажатиях клавиш, поэтому нам пришлось бы непрерывно создавать новые экземпляры классов, которые подхватывались бы сборщиком мусора через небольшие промежутки времени. Чтобы этого избежать, реализуем концепцию, известную как *пулинг экземпляров* (instance pooling). Вместо того чтобы постоянно создавать новые экземпляры класса, мы просто будем использовать ранее созданные экземпляры. Класс Pool — это удобный способ реализовать подобное поведение программы. Рассмотрим его код в листинге 5.6.

Листинг 5.6. Pool.java; учимся обращаться со сборщиком мусора

```
package com.badlogic.androidgames.framework;

import java.util.ArrayList;
import java.util.List;

public class Pool<T> {
```

Теперь в ход пойдут дженерики: первая вещь, которую необходимо учесть, — то, что данный класс характеризуется обобщенной типизацией (generically typed) подобно классам коллекций, таким как ArrayList или HashMap. Дженерик позволяет хранить объект любого типа в нашем Pool, не тратя времени на лишние операции приведения. Так что же делает класс Pool?

```
public interface PoolObjectFactory<T> {
    public T createObject();
}
```

Для начала определим интерфейс PoolObjectFactory, который опять-таки является дженериком. Он использует только один метод, createObject(), который возвращает новый обобщенный тип экземпляра класса Pool/PoolObjectFactory.

```
private final List<T> freeObjects;
private final PoolObjectFactory<T> factory;
private final int maxSize;
```

Класс Pool включает в себя три члена: ArrayList, сохраняющий объекты пула, PoolObjectFactory, который генерирует новые экземпляры типа, содержащегося в классе, и еще один член, хранящий максимальное количество объектов, которое может содержать в себе Pool. Последний элемент необходим для того, чтобы избежать неконтролируемого роста Pool, иначе может возникнуть исключение нехватки памяти.

```
public Pool(PoolObjectFactory<T> factory, int maxSize) {
    this.factory = factory;
    this.maxSize = maxSize;
    this.freeObjects = new ArrayList<T>(maxSize);
}
```

Конструктор класса Pool принимает PoolObjectFactory и максимальное количество объектов, которое он может хранить. Мы сохраняем оба параметра в соответствующих переменных и создаем новый ArrayList, вместимость которого равна максимальному количеству объектов.

```
public T newObject() {
    T object = null;

    if (freeObjects.size() == 0)
        object = factory.createObject();
    else
        object = freeObjects.remove(freeObjects.size() - 1);

    return object;
}
```

Метод newObject() отвечает или за передачу нового экземпляра класса данного типа, который Pool получил через метод PoolObjectFactory.newObject(), или за то, чтобы возвратить обобщенный экземпляр, если он уже есть, в freeObjects ArrayList. Если мы используем данный метод, то получим заново обработанные объекты при условии, что Pool уже хранит несколько таких объектов в списке freeObjects. В противном случае метод создаст новый объект с помощью фабрики.

```
    public void free(T object) {
        if (freeObjects.size() < maxSize)
            freeObjects.add(object);
    }
}
```

Метод free() позволяет перевставлять объекты, которые мы больше не используем. Он просто вставляет объекты в список freeObjects, конечно, если тот не заполнен полностью. Если список полон, объект не будет добавлен. Скорее всего, его подберет сборщик мусора, когда будет запущен в следующий раз.

Как же мы можем использовать этот класс? Рассмотрим псевдокод, который показывает применение класса Pool при обработке касаний на экране:

```
PoolObjectFactory<TouchEvent> factory = new PoolObjectFactory<TouchEvent>() {
    @Override
    public TouchEvent createObject() {
        return new TouchEvent();
    }
};
Pool<TouchEvent> touchEventPool = new Pool<TouchEvent>(factory, 50);
TouchEvent touchEvent = touchEventPool.newObject();
//… что-то здесь делаем…
touchEventPool.free(touchEvent);
```

Сначала определяем PoolObjectFactory, который создает экземпляры класса TouchEvent. Далее создаем Pool и указываем ему использовать нашу фабрику и одновременно хранить не более 50 событий TouchEvent. Когда нам нужен новый TouchEvent из Pool, используем метод newObject(), относящийся к Pool. Изначально Pool пустой, поэтому он попросит фабрику создать новый экземпляр класса TouchEvent. Когда мы больше не нуждаемся в TouchEvent, мы можем вернуть его в Pool с помощью метода Pool.free(). В следующий раз, когда мы используем метод newObject(), мы снова получим этот же экземпляр класса TouchEvent и вновь его задействуем. Таким образом, в данной ситуации мы обойдемся без сборщика мусора. Данный класс удобно применять во многих ситуациях. Просто запомните, что нужно быть внимательными, если вы многократно используете объекты. Если вы берете их из класса Pool, то их повторная инициализация, возможно, получится неполной, а этого допускать нельзя.

KeyboardHandler: вверх, вверх, вниз, вниз, влево, вправо…

Обработчик событий клавиатуры KeyboardHandler используется для выполнения целого ряда задач. Прежде всего он должен быть связан с View, так как от него KeyboardHandler получает информацию о событиях клавиатуры. Далее ему необходимо сохранить текущее состояние каждой клавиши для предъявления этого состояния в ходе опроса. Ему также понадобится список экземпляров класса KeyEvent, который мы создали в главе 3 для обработки событий ввода. И наконец, все это нужно правильно синхронизировать, так как он будет принимать события из потока пользовательского интерфейса, которые будут обрабатываться в главном игровом цикле, запущенном в другом потоке. Достаточно много работы. В качестве

небольшого напоминания еще раз посмотрим на класс KeyEvent, который мы определили в главе 3 как часть интерфейса ввода Input:

```
public static class KeyEvent {
    public static final int KEY_DOWN = 0;
    public static final int KEY_UP = 1;

    public int type;
    public int keyCode;
    public char keyChar;
}
```

Он просто определяет две константы, кодирующие тип клавиатурного события, а также содержит три члена, в которых хранятся тип, код клавиши и Юникод-символ события. Располагая этой информацией, мы можем реализовать наш обработчик.

Листинг 5.7 показывает реализацию обработчика с помощью нашего нового класса Pool и Android API, который мы обсуждали ранее.

Листинг 5.7. KeyboardHandler.java: обрабатывает клавиатуру

```
package com.badlogic.androidgames.framework.impl;

import java.util.ArrayList;
import java.util.List;

import android.view.View;
import android.view.View.OnKeyListener;

import com.badlogic.androidgames.framework.Input.KeyEvent;
import com.badlogic.androidgames.framework.Pool;
import com.badlogic.androidgames.framework.Pool.PoolObjectFactory;

public class KeyboardHandler implements OnKeyListener {
    boolean[] pressedKeys = new boolean[128];
    Pool<KeyEvent> keyEventPool;
    List<KeyEvent> keyEventsBuffer = new ArrayList<KeyEvent>();
    List<KeyEvent> keyEvents = new ArrayList<KeyEvent>();
```

Класс KeyboardHandler реализует интерфейс OnKeyListener, так что он может получать клавиатурные события из View. Переходим к нашим элементам.

Первый элемент — это массив, содержащий 128 булевых значений. Мы сохраним текущее состояние (нажата или не нажата) каждой клавиши в данном массиве. Это состояние индексируется кодом клавиши. Удобно, что константы android.view.KeyEvent.KEYCODE_XXX, которые задают коды клавиш, варьируются в промежутке от 0 до 127, так что мы можем сохранить их в форме, удобной для сборщика мусора. Обратите внимание, что, к сожалению, имя нашего класса KeyEvent совпадает с классом KeyEvent Android, экземпляры которого будут передаваться методу OnKeyEventListener.onKeyEvent(). Такая путаница возникает только с кодом этого обработчика. Поскольку вряд ли клавиатурное событие можно поименовать лучше, чем KeyEvent, я предпочитаю оставить все как есть.

Следующий член Pool хранит экземпляры наших классов KeyEvent. Поскольку мы не хотим перегружать сборщик мусора работой, будем многократно использовать все объекты KeyEvent, которые создали.

Третий элемент хранит KeyEvent, которые пока не были обработаны классом Game. Каждый раз, когда мы получаем новое событие клавиатуры в потоке пользовательского интерфейса, оно добавляется в этот список.

Последний элемент хранит KeyEvent, которые мы вернем при вызове KeyboardHandler.getKeyEvents(). Чуть ниже будет показано, почему мы дважды буферизуем клавиатурные события.

```
public KeyboardHandler(View view) {
    PoolObjectFactory<KeyEvent> factory = new PoolObjectFactory<KeyEvent>() {
        @Override
        public KeyEvent createObject() {
            return new KeyEvent();
        }
    };
    keyEventPool = new Pool<KeyEvent>(factory, 100);
    view.setOnKeyListener(this);
    view.setFocusableInTouchMode(true);
    view.requestFocus();
}
```

Данный конструктор имеет только один параметр, обозначающий тот вид (View), от которого мы хотим получать клавиатурные события. Мы создаем экземпляр класса Pool с соответствующим PoolObjectFactory, регистрируем обработчик как OnKeyListener во View и, наконец, убеждаемся, что View получает события клавиатуры, поместив этот вид в фокус.

```
@Override
public boolean onKey(View v, int keyCode, android.view.KeyEvent event) {
    if (event.getAction() == android.view.KeyEvent.ACTION_MULTIPLE)
        return false;

    synchronized (this) {
        KeyEvent keyEvent = keyEventPool.newObject();
        keyEvent.keyCode = keyCode;
        keyEvent.keyChar = (char) event.getUnicodeChar();
        if (event.getAction() == android.view.KeyEvent.ACTION_DOWN) {
            keyEvent.type = KeyEvent.KEY_DOWN;
            if(keyCode > 0 && keyCode < 127)
                pressedKeys[keyCode] = true;
        }
        if (event.getAction() == android.view.KeyEvent.ACTION_UP) {
            keyEvent.type = KeyEvent.KEY_UP;
            if(keyCode > 0 && keyCode < 127)
                pressedKeys[keyCode] = false;
        }
        keyEventsBuffer.add(keyEvent);
    }
    return false;
}
```

Далее реализуем метод интерфейса OnKeyListener.onKey(), который вызывается каждый раз, когда View обрабатывает новое клавиатурное событие. Однако таким образом игнорируются все клавиатурные события с типом KeyEvent.ACTION_MULTIPLE. Чтобы это исправить, используем блок синхронизации. Не забывайте, что все события поступают из потока пользовательского интерфейса и затем обрабатываются потоком основного цикла, поэтому нам необходимо убедиться, что ни один из элементов не может быть параллельно доступен для нескольких потоков.

Внутри блока синхронизации мы сначала получаем экземпляр класса KeyEvent (нашей реализации KeyEvent) из Pool. Это поможет нам получить или переработанный, или новый экземпляр класса в зависимости от состояния Pool. Далее устанавливаем элементы KeyEvent, а именно keyCode и keyChar, которые берутся из KeyEvent Android, переданного в этот метод. Далее декодируем тип KeyEvent Android и устанавливаем соответствующие значения для типа нашего события KeyEvent и элемента в массиве pressedKey. В конце добавляем KeyEvent в список keyEventBuffer, который определили ранее.

```
public boolean isKeyPressed(int keyCode) {
    if (keyCode < 0 || keyCode > 127)
        return false;
    return pressedKeys[keyCode];
}
```

Далее переходим к методу isKeyPressed(), который обычно реализует семантику Input.isKeyPressed(). Мы передаем в него целое число, которое соответствует коду клавиши (одной из констант KeyEvent.KEYCODE_XXX Android) и возвращаем информацию о том, нажата данная клавиша или нет. Статус клавиши можно получить из массива pressedKey, предварительно проверив границы диапазона. Обратите внимание, что мы установили элементы данного массива в прошлом методе, вызываемом в потоке пользовательского интерфейса. Поскольку мы снова работаем с простейшими типами, необходимости в синхронизации нет.

```
    public List<KeyEvent> getKeyEvents() {
        synchronized (this) {
            int len = keyEvents.size();
            for (int i = 0; i < len; i++)
                    keyEventPool.free(keyEvents.get(i));
                keyEvents.clear();
                keyEvents.addAll(keyEventsBuffer);
                keyEventsBuffer.clear();
                return keyEvents;
        }
    }
}
```

Последний метод нашего обработчика называется getKeyEvents(). Он реализует семантику метода Input.getKeyEvents(). Мы снова запускаем блок синхронизации, так как данный метод вызывается из другого потока.

Теперь начинаются интересности. Мы проходим в цикле по массиву keyEvents и вставляем все KeyEvent, хранящиеся в нем, в наш Pool. Помните, как мы получали

экземпляры KeyEvent из класса Pool в методе onKey() в потоке пользовательского интерфейса? Здесь мы вставляем их назад в Pool. Но разве список KeyEvent не пуст? Данный список будет пуст только во время первого вызова данного метода. Чтобы понять, почему возникает такая ситуация, необходимо изучить данный метод до конца.

После нашего загадочного цикла вставки в Pool мы очищаем список KeyEvent и заполняем его событиями из списка keyEventsBuffer. И наконец, очищаем список keyEventsBuffer и возвращаем свежезаполненный список KeyEvent вызывающей стороне. Что же происходит здесь?

Проще всего это объясняется следующим простым примером. Мы проверяем, что происходит со списками KeyEvent и keyEventsBuffer, а также с Pool каждый раз, когда новое событие появляется в потоке пользовательского интерфейса или когда игра запрашивает новое событие из основного потока:

```
UI thread: onKey() ->
        keyEvents = { }, keyEventsBuffer = {KeyEvent1}, pool = { }
Main thread: getKeyEvents() ->
        keyEvents = {KeyEvent1}, keyEventsBuffer = { }, pool { }
UI thread: onKey() ->
        keyEvents = {KeyEvent1}, keyEventsBuffer = {KeyEvent2}, pool { }
Main thread: getKeyEvents() ->
        keyEvents = {KeyEvent2}, keyEventsBuffer = { }, pool = {KeyEvent1}
UI thread: onKey() ->
        keyEvents = {KeyEvent2}, keyEventsBuffer = {KeyEvent1}, pool = { }
```

1. Сначала получаем новое событие в потоке пользовательского интерфейса. В Pool пока ничего нет, поэтому создается новый экземпляр класса KeyEvent (KeyEvent1). Он помещается в список keyEventsBuffer.
2. Вызываем getKeyEvents() в основном потоке. Он использует KeyEvent1 из списка keyEventsBuffer и помещает его в список keyEvent, который возвращается вызывающей стороне.
3. Получаем еще одно событие в потоке UI. В нашем Pool по-прежнему ничего нет, поэтому создается еще один экземпляр класса KeyEvent (KeyEvent2). Он также помещается в список keyEventsBuffer.
4. Основной поток снова вызывает getKeyEvents(). Здесь начинается самое интересное. При входе в данный метод список KeyEvent по-прежнему содержит KeyEvent1. Цикл вставки поместит данное событие в наш Pool. Затем он очистит список KeyEvent и вставит другой KeyEvent в keyEventsBuffer. В нашем случае это KeyEvent2. Мы только что переработали клавиатурное событие.
5. Наконец, получаем еще одно событие в потоке UI. На этот раз у нас есть свободный KeyEvent в Pool, который мы, разумеется, используем еще один раз. И никаких сборщиков мусора.

Однако у данного механизма есть серьезный недостаток: нам придется достаточно часто вызывать KeyboardHandler.getKeyEvents(), иначе список быстро заполнится и нельзя будет возвратить объекты в Pool. Однако если мы будем помнить об этом, все должно быть в порядке.

Обработчики касаний

А вот и последствия «разделения» на старые и новые версии. В предыдущей главе мы практически не говорили о том, что мультитач поддерживается только версиями Android выше 1.6. Все константы, которые мы использовали в коде мультитач (например, `MotionEvent.ACTION_POINTER_ID_MASK`), недоступны в версиях Android 1.5 или 1.6. Мы вполне можем применять их в нашем коде, если нашей конечной целью является запуск программы на той версии Android, которая поддерживает данный API. Однако тогда наше приложение не будет работать на устройствах, на которых установлена версия Android 1.5 или 1.6. Но мы ведь с вами хотим, чтобы наша игра была доступна на всех версиях Android. Как же нам решить данную проблему?

Для этого воспользуемся небольшой хитростью. Напишем два обработчика, один из которых будет использовать API Android 1.5 для обработки одиночных касаний (single-touch), а другой — мультитач API Android 2.0 и выше. До тех пор пока мы не будем использовать мультитач-обработчик на устройствах с версией Android ниже 2.0, мы можем быть абсолютно спокойны. Виртуальная машина не будет загружать код, поэтому можно не волноваться о том, что появятся исключения. Все, что нам нужно, — выяснить, какая версия Android установлена на устройстве, и выбрать для нее соответствующий обработчик. Вы увидите, как это работает, когда мы будем обсуждать класс `AndroidInput`. А сейчас сконцентрируемся на наших двух обработчиках.

Интерфейс TouchHandler

Чтобы мы могли использовать два класса обработчиков, заменяя один другим, нам необходимо определить общий интерфейс. В листинге 5.8 показан данный интерфейс, называющийся `TouchHandler`.

Листинг 5.8. РеализацияTouchHandler.java для Android 1.5 и 1.6.

```
package com.badlogic.androidgames.framework.impl;

import java.util.List;

import android.view.View.OnTouchListener;

import com.badlogic.androidgames.framework.Input.TouchEvent;

public interface TouchHandler extends OnTouchListener {
    public boolean isTouchDown(int pointer);

    public int getTouchX(int pointer);

    public int getTouchY(int pointer);

    public List<TouchEvent> getTouchEvents();
}
```

Все `TouchHandler` должны реализовать интерфейс `OnTouchListener`, который используется для того, чтобы зарегистрировать обработчик во `View`. Методы интер-

фейса соответствуют методам интерфейса Input, который мы определили в главе 3. Первые три метода применяются для получения состояния определенного указателя, а четвертый необходим для получения TouchEvent, чтобы мы смогли обработать события ввода. Обратите внимание, что данный метод для опроса принимает ID указателя.

Класс SingleTouchHandler

В случае с обработчиком одиночных касаний мы принимаем во внимание только нулевой ID. Давайте вспомним, как работает класс TouchEvent, который мы рассматривали в главе 3, когда говорили об интерфейсе Input:

```
public static class TouchEvent {
    public static final int TOUCH_DOWN = 0;
    public static final int TOUCH_UP = 1;
    public static final int TOUCH_DRAGGED = 2;

    public int type;
    public int x, y;
    public int pointer;
}
```

Как и в классе KeyEvent, здесь определяется пара констант, а также хранится тип события касания, x- и y-координаты в координатной системе вида View и ID указателя.

Листинг 5.9 показывает реализацию интерфейса TouchHandler для Android версий 1.5 и 1.6.

Листинг 5.9. SingleTouchHandler.java; подходит только для обработки одиночных касаний

```
package com.badlogic.androidgames.framework.impl;

import java.util.ArrayList;
import java.util.List;

import android.view.MotionEvent;
import android.view.View;

import com.badlogic.androidgames.framework.Pool;
import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.Pool.PoolObjectFactory;

public class SingleTouchHandler implements TouchHandler {
    boolean isTouched;
    int touchX;
    int touchY;
    Pool<TouchEvent> touchEventPool;
    List<TouchEvent> touchEvents = new ArrayList<TouchEvent>();
    List<TouchEvent> touchEventsBuffer = new ArrayList<TouchEvent>();
    float scaleX;
    float scaleY;
```

Мы начинаем с того, что реализуем интерфейс TouchHandler. А это также предполагает, что нам придется реализовать интерфейс OnTouchListener. Далее следуют несколько членов, которые вам уже знакомы. На данный момент у нас есть три члена, которые хранят текущее состояние сенсорного экрана для одного пальца, за ними следует Pool и два списка TouchEvent. То же самое происходило в KeyboardHandler. У нас также есть два члена: scaleX и scaleY. К ним мы вернемся чуть позже. Они необходимы для того, чтобы справиться с разными разрешениями экрана.

ПРИМЕЧАНИЕ

Конечно, мы могли бы сделать более стройную систему, позволив KeyboardHandler и ingleTouchHandler наследовать от основного класса, который отвечает за сбор информации и синхронизацию. Однако объяснение стало бы еще более сложным, так что мы просто напишем еще несколько строк кода.

```
public SingleTouchHandler(View view, float scaleX, float scaleY) {
        PoolObjectFactory<TouchEvent> factory = new
                                    PoolObjectFactory<TouchEvent>() {
            @Override
            public TouchEvent createObject() {
                return new TouchEvent();
            }
        };
        touchEventPool = new Pool<TouchEvent>(factory, 100);
        view.setOnTouchListener(this);

        this.scaleX = scaleX;
        this.scaleY = scaleY;
    };
```

В конструкторе регистрируем обработчик как OnTouchListener и инициализируем Pool, который используем для переработки TouchEvent. Мы также сохраняем параметры scaleX и scaleY, которые передаются конструктору (однако пока не сосредотачиваемся на них).

```
@Override
    public boolean onTouch(View v, MotionEvent event) {
        synchronized(this) {
            TouchEvent touchEvent = touchEventPool.newObject();
            switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                touchEvent.type = TouchEvent.TOUCH_DOWN;
                isTouched = true;
                break;
            case MotionEvent.ACTION_MOVE:
                touchEvent.type = TouchEvent.TOUCH_DRAGGED;
                isTouched = true;
                break;
            case MotionEvent.ACTION_CANCEL:
            case MotionEvent.ACTION_UP:
```

```
                touchEvent.type = TouchEvent.TOUCH_UP;
                isTouched = false;
                break;
            }

            touchEvent.x = touchX = (int)(event.getX() * scaleX);
            touchEvent.y = touchY = (int)(event.getY() * scaleY);
            touchEventsBuffer.add(touchEvent);

            return true;
        }
    }
```

Метод `onTouch()` выполняет те же функции, что и метод `onKey()` для `KeyboardHandler`. Единственная разница состоит в том, что теперь мы обрабатываем `TouchEvent`, а не `KeyEvent`. Вы уже знакомы с особенностями синхронизации, сбора данных и обработкой `MotionEvent`. Единственная вещь, которая заслуживает отдельного внимания, это то, что мы фактически умножаем полученные x- и y-координаты на `scaleX` и `scaleY` соответственно. Запомните этот момент, мы вернемся к нему чуть позже.

```
@Override
    public boolean isTouchDown(int pointer) {
        synchronized(this) {
            if(pointer == 0)
                return isTouched;
            else
                return false;
        }
    }

    @Override
    public int getTouchX(int pointer) {
        synchronized(this) {
            return touchX;
        }
    }

    @Override
    public int getTouchY(int pointer) {
        synchronized(this) {
            return touchY;
        }
    }
```

Методы `isTouchDown()`, `getTouchX()` и `getTouchY()` позволяют получить информацию о состоянии сенсорного экрана, основываясь на значениях, которые были заданы в методе `onTouch()`. Единственный примечательный момент, связанный с ними,

заключается в том, что они возвращают полезные данные для указателя с нулевым значением ID, так как в данном классе мы работаем только с экранами, поддерживающими обработку одиночных касаний.

```
@Override
    public List<TouchEvent> getTouchEvents() {
        synchronized(this) {
            int len = touchEvents.size();
            for( int i = 0; i < len; i++ )
                touchEventPool.free(touchEvents.get(i));
            touchEvents.clear();
            touchEvents.addAll(touchEventsBuffer);
            touchEventsBuffer.clear();
            return touchEvents;
        }
    }
}
```

Последний метод, `SingleTouchHandler.getTouchEvents()`, должен уже быть вам знаком. Он работает так же, как методы `KeyboardHandler.getKeyEvents()`. Необходимо вызывать этот метод достаточно часто, чтобы список `touchEvents` не переполнялся.

Обработчик MultiTouchHandler

Для обработки мультитач используем класс `MultiTouchHandler`, описанный в листинге 5.10.

Листинг 5.10. Multitouchhandler.java (все то же самое)

```
package com.badlogic.androidgames.framework.impl;

import java.util.ArrayList;
import java.util.List;

import android.view.MotionEvent;
import android.view.View;

import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.Pool;
import com.badlogic.androidgames.framework.Pool.PoolObjectFactory;

public class Multitouchhandler implements TouchHandler {
    boolean[] isTouched = new boolean[20];
    int[] touchX = new int[20];
    int[] touchY = new int[20];
    Pool<TouchEvent> touchEventPool;

    List<TouchEvent> touchEvents = new ArrayList<TouchEvent>();
    List<TouchEvent> touchEventsBuffer = new ArrayList<TouchEvent>();
    float scaleX;
    float scaleY;
```

В свою очередь этот класс снова реализует интерфейс TouchHandler и содержит несколько членов для того, чтобы хранить текущие состояния и события. Вместо того чтобы хранить данные о состоянии каждого указателя отдельно, мы просто сохраняем состояние 20 указателей. Мы также снова используем поля scaleX и scaleY.

```
public Multitouchhandler(View view, float scaleX, float scaleY) {
        PoolObjectFactory<TouchEvent> factory = new
                                        PoolObjectFactory<TouchEvent>() {
            @Override
            public TouchEvent createObject() {
                return new TouchEvent();
            }
        };
        touchEventPool = new Pool<TouchEvent>(factory, 100);
        view.setOnTouchListener(this);

        this.scaleX = scaleX;
        this.scaleY = scaleY;
    }
```

Данный конструктор является точной копией конструктора SingleTouchHandler: мы создаем Pool для экземпляров Touch event, регистрируем класс в качестве обработчика и сохраняем значения масштабирования.

```
@Override
    public boolean onTouch(View v, MotionEvent event) {
        synchronized (this) {
            int action = event.getAction() & MotionEvent.ACTION_MASK;
            int pointerIndex = (event.getAction() &
                                MotionEvent.ACTION_POINTER_ID_MASK)
                                >> MotionEvent.ACTION_POINTER_ID_SHIFT;
            int pointerId = event.getPointerId(pointerIndex);
            TouchEvent touchEvent;

            switch (action) {
            case MotionEvent.ACTION_DOWN:
            case MotionEvent.ACTION_POINTER_DOWN:
                touchEvent = touchEventPool.newObject();
                touchEvent.type = TouchEvent.TOUCH_DOWN;
                touchEvent.pointer = pointerId;
                touchEvent.x = touchX[pointerId] = (int) (event
                        .getX(pointerIndex) * scaleX);
                touchEvent.y = touchY[pointerId] = (int) (event
                        .getY(pointerIndex) * scaleY);
                isTouched[pointerId] = true;
                touchEventsBuffer.add(touchEvent);
                break;

            case MotionEvent.ACTION_UP:
            case MotionEvent.ACTION_POINTER_UP:
            case MotionEvent.ACTION_CANCEL:
```

```
                touchEvent = touchEventPool.newObject();
                touchEvent.type = TouchEvent.TOUCH_UP;
                touchEvent.pointer = pointerId;
                touchEvent.x = touchX[pointerId] = (int) (event
                        .getX(pointerIndex) * scaleX);
                touchEvent.y = touchY[pointerId] = (int) (event
                        .getY(pointerIndex) * scaleY);
                isTouched[pointerId] = false;
                touchEventsBuffer.add(touchEvent);
                break;

            case MotionEvent.ACTION_MOVE:
                int pointerCount = event.getPointerCount();
                for (int i = 0; i < pointerCount; i++) {
                    pointerIndex = i;
                    pointerId = event.getPointerId(pointerIndex);

                    touchEvent = touchEventPool.newObject();
                    touchEvent.type = TouchEvent.TOUCH_DRAGGED;
                    touchEvent.pointer = pointerId;
                    touchEvent.x = touchX[pointerId] = (int) (event
                            .getX(pointerIndex) * scaleX);
                    touchEvent.y = touchY[pointerId] = (int) (event
                            .getY(pointerIndex) * scaleY);
                    touchEventsBuffer.add(touchEvent);
                }
                break;
            }

            return true;
        }
    }
```

Метод `onTouch()` кажется таким же страшным и непонятным, как и в тестовом примере из предыдущей главы. Все, что мы делаем, — объединяем код теста с пулингом и синхронизацией событий, об этом мы уже говорили подробно. Единственное серьезное отличие от метода `SingleTouchHandler.onTouch()` заключается в том, что мы обрабатываем несколько указателей и устанавливаем для элемента `TouchEvent.pointer` соответствующее значение (а не просто задаем его равным нулю).

```
@Override
    public boolean isTouchDown(int pointer) {
        synchronized (this) {
            if (pointer < 0 || pointer >= 20)
                return false;
            else
                return isTouched[pointer];
        }
    }

    @Override
```

```
    public int getTouchX(int pointer) {
        synchronized (this) {
            if (pointer < 0 || pointer >= 20)
                return 0;
            else
                return touchX[pointer];
         }
    }

    @Override
    public int getTouchY(int pointer) {
        synchronized (this) {
            if (pointer < 0 || pointer >= 20)
                return 0;
            else
                return touchY[pointer];
        }
    }
```

Методы для опроса `isTouchDown()`, `getTouchX()` и `getTouchY()` также должны быть вам уже знакомы. Мы выполняем проверку на ошибки, а затем переносим соответствующее состояние указателя из того элемента массива, который заполняем с помощью метода `onTouch()`.

```
@Override
    public List<TouchEvent> getTouchEvents() {
        synchronized (this) {
            int len = touchEvents.size();
            for (int i = 0; i < len; i++)
                    touchEventPool.free(touchEvents.get(i));
                touchEvents.clear();
                touchEvents.addAll(touchEventsBuffer);
                touchEventsBuffer.clear();
                return touchEvents;
        }
    }
}
```

Последний метод, который мы здесь используем, — `getTouchEvents()`. Он опять-таки полностью совпадает с соответствующим методом в `SingleTouchHandler.getTouchEvents()`.

Используя все эти обработчики, мы наконец-то можем реализовать интерфейс `Input`.

AndroidInput: отличный координатор

Реализация интерфейса `Input` для нашей игры связывает вместе все обработчики, которые мы только что создали. Любые вызовы методов будут передаваться соответствующим обработчикам. Единственная часть реализации, которая требует особого внимания, — выбор того, как запрограммировать `TouchHandler` в зависимости от версии Android, установленной на устройстве.

В листинге 5.11 показано, как реализовать AndroidInput.

Листинг 5.11. AndroidInput.java

```
package com.badlogic.androidgames.framework.impl;

import java.util.List;

import android.content.Context;
import android.os.Build.VERSION;
import android.view.View;

import com.badlogic.androidgames.framework.Input;

public class AndroidInput implements Input {
    AccelerometerHandler accelHandler;
    KeyboardHandler keyHandler;
    TouchHandler touchHandler;
```

Начинаем с того, что класс реализует интерфейс Input, который мы описывали в главе 3. Далее у нас есть три члена класса: AccelerometerHandler, KeyboardHandler и TouchHandler.

```
public AndroidInput(Context context, View view, float scaleX, float scaleY) {
        accelHandler = new AccelerometerHandler(context);
        keyHandler = new KeyboardHandler(view);
        if(Integer.parseInt(VERSION.SDK) < 5)
            touchHandler = new SingleTouchHandler(view, scaleX, scaleY);
        else
            touchHandler = new Multitouchhandler(view, scaleX, scaleY);
    }
```

Три этих члена инициализируются в конструкторе, который принимает Context, View и значения scaleX и scaleY, которые мы пока снова игнорируем. Обработчик AccelerometerHandler устанавливается при помощи Context, а KeyboardHandler в свою очередь использует View для инициализации.

Чтобы понять, какой TouchHandler необходимо применить, мы просто проверяем, на какой версии Android работает приложение. Это можно сделать с помощью строки VERSION.SDK, константы Android API. Не совсем понятно, почему это именно строка, так как она просто содержит номер версии SDK, который мы используем в нашем файле описания. Поэтому нам необходимо преобразовать ее в целое число, чтобы потом было легче сравнивать. Самая актуальная версия, поддерживающая мультитач API, — 2.0, которая соответствует пятой версии SDK. Если устройство применяет Android версии ниже, запускаем SingleTouchHandler, в противном случае используем MultiTouchHandler. Вот и все проблемы с разделением на старые и новые версии, с которыми необходимо справиться на уровне API. Когда мы займемся отображением с использованием OpenGL, придется встретиться еще с несколькими случаями разделения. Но не стоит волноваться, справиться с ними будет так же легко, как и с проблемами API.

```
@Override
    public boolean isKeyPressed(int keyCode) {
        return keyHandler.isKeyPressed(keyCode);
    }

    @Override
    public boolean isTouchDown(int pointer) {
        return touchHandler.isTouchDown(pointer);
    }

    @Override
    public int getTouchX(int pointer) {
        return touchHandler.getTouchX(pointer);
    }

    @Override
    public int getTouchY(int pointer) {
    return touchHandler.getTouchY(pointer);
    }

    @Override
    public float getAccelX() {
        return accelHandler.getAccelX();
    }

    @Override
    public float getAccelY() {
        return accelHandler.getAccelY();
    }

    @Override
    public float getAccelZ() {
        return accelHandler.getAccelZ();
    }

    @Override
    public List<TouchEvent> getTouchEvents() {
        return touchHandler.getTouchEvents();
    }

    @Override
    public List<KeyEvent> getKeyEvents() {
        return keyHandler.getKeyEvents();
    }
}
```

Оставшаяся часть данного класса понятна без объяснений. Каждый вызов метода передается соответствующему обработчику, который и выполняет саму работу. Таким образом, мы закончили работу с API обработки ввода во фреймворке нашей небольшой игры. Теперь самое время перейти к графике.

AndroidGraphics и AndroidPixmap: двойная радуга

Итак, возвращаемся к нашей самой любимой теме: программированию графики. В главе 3 мы описали два интерфейса: `Graphics` и `Pixmap`, которые мы теперь реализуем, основываясь на знаниях, полученных в главе 4. Однако есть еще один аспект, изучение которого мы отложили до текущего момента, а именно: что делать с экранами различного размера и разрешения.

Обработка различных размеров экрана и разрешений

Android поддерживает различные разрешения экрана начиная с версии 1.6. Она может обрабатывать разрешения от 240×320 до 480×854 пиксела на некоторых новых устройствах (в книжной и альбомной ориентации показатели меняются местами). В предыдущей главе мы уже видели эффект от применения различных разрешений экрана и его физических размеров: рисование с учетом абсолютных координат и размеров в пикселах может привести к самым разным результатам.

На рис. 5.1 изображено, что происходит, когда мы визуализируем прямоугольник 100×100 пикселов, верхний угол которого находится в точке (219; 379) на экранах 480×800 и 320×480.

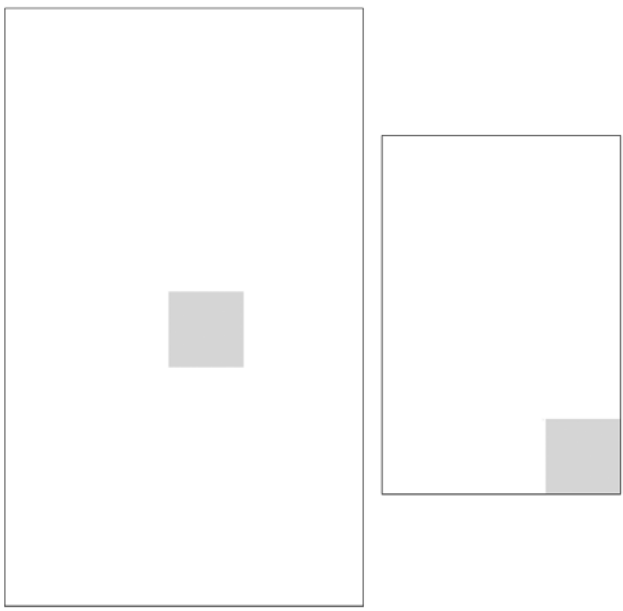

Рис. 5.1. Прямоугольник 100×100 пикселов, верхний угол которого находится в точке (219;379) на экранах 480×800 *(слева)* и 320×480 *(справа)*

Такая разница нас не устраивает по двум причинам. Мы не можем написать нашу игру для какого-то конкретного разрешения. Вторая причина не так очевидна. Дело в том, что на рис. 5.1 я принял за аксиому, что у обоих экранов одинаковая плотность (то есть каждый пиксел имеет одинаковый физический размер на обоих устройствах), но в реальности такого не будет.

Плотность

Как правило, плотность определяется в пикселах на дюйм или пикселах на сантиметр (иногда вы можете встретить выражение «точка на дюйм», но это не совсем верно). У Nexus One экран 480 × 800 пикселов при физическом размере 8 × 4,8 см. HTC Hero имеет экран 320 × 480 пикселов при физическом размере 6,5 × 4,5 см. Таким образом, 100 пикселов на сантиметр по обеим осям в Nexus One — это примерно 71 пиксел на сантиметр по обеим осям в HTC Hero. Мы можем легко подсчитать пикселы на сантиметр с помощью следующего уравнения:

```
Пикселы на сантиметр (по оси x) = ширина в пикселах / ширина в сантиметрах
```

или этого:

```
Пикселы на сантиметр (по оси y) = высота в пикселах / высота в сантиметрах
```

Как правило, достаточно подсчитать пикселы на сантиметр только под одной оси, так как пикселы имеют квадратную форму (если честно, здесь всего три пиксела, но мы сейчас не об этом).

Какого размера будет наш прямоугольник 100 × 100 пикселов в сантиметрах? На Nexus One у нас будет прямоугольник 1 × 1 см, а на HTC Hero — 1,4 × 1,4 см.

Помните об этом, когда будете отрисовывать такие объекты, как кнопки, которые должны быть достаточно большими, чтобы на кнопку можно было нажать пальцем и это легко происходило на экранах любого размера. Хотя на первый взгляд кажется, что данная особенность может представлять большую проблему для разработчика, на самом деле это не так. Все, что нужно, — удостовериться, что кнопки имеют достаточно большой размер на экранах с большой плотностью, таких как Nexus One. Они будут автоматически увеличиваться на экранах с меньшей плотностью.

Соотношение сторон

Еще одна проблема, с которой нам предстоит справиться, — соотношение сторон. Соотношение сторон экрана — это отношение между шириной и высотой экрана в пикселах или сантиметрах. Оно подсчитывается следующим образом:

```
Соотношение сторон в пикселах = ширина в пикселах / высота в пикселах
```

или так:

```
Физическое соотношение сторон = ширина в сантиметрах / высота в сантиметрах
```

Употребляя слова «ширина» и «высота», мы, как правило, говорим о ширине и высоте в ландшафтной ориентации. Соотношение сторон в Nexus One, физическое и в пикселах, примерно равно 1,66. У HTC Hero физическое и пиксельное соотношение

сторон равно 1,5. Что это значит? На Nexus One мы имеем в своем распоряжении больше пикселов по оси *x* в альбомной ориентации по сравнению с той высотой, которая доступна на HTC Hero. На рис. 5.2 показано, как выглядит игра Replica Island на обоих устройствах.

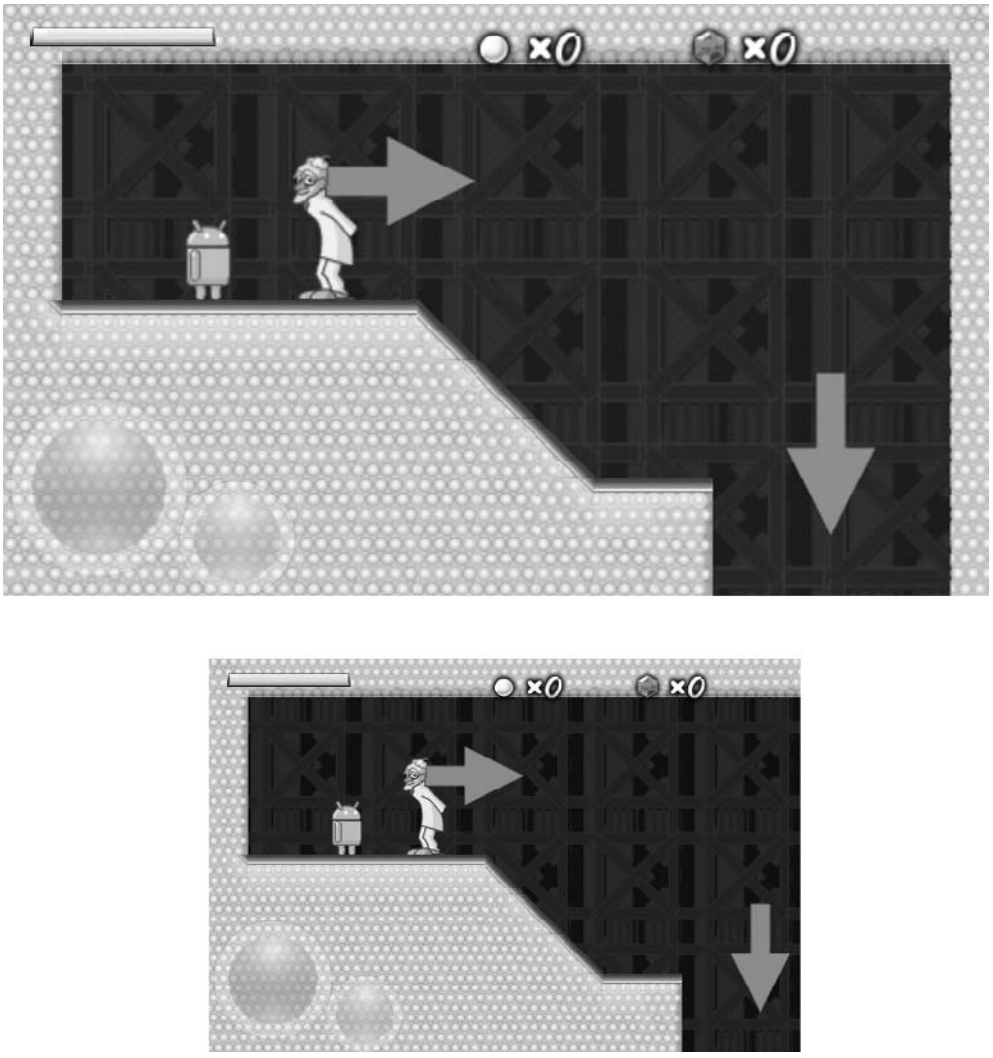

Рис. 5.2. Replica Island на Nexus One *(сверху)* и HTC Hero *(снизу)*

ПРИМЕЧАНИЕ

В этой книге используется метрическая система.

На Nexus One чуть-чуть больше обзор по оси *x*. Однако по оси *y* все выглядит точно так же. Что же создатели Replica Island здесь сделали?

Как справляться с различными соотношениями сторон

Replica Island использует достаточно простую, но весьма эффективную хитрость для того, чтобы справиться с проблемой соотношения сторон. Изначально игра создавалась для экрана 480 × 320 пикселов, включая все спрайты (например, робота и доктора), сам мир и элементы пользовательского интерфейса (например, кнопки внизу слева и информацию о статусе вверху экрана). Когда игра отображается на HTC Hero, каждый пиксел спрайта в растровом отображении соответствует одному пикселу на экране. На Nexus One во время визуализации все масштабируется, так что 1 пиксел спрайта в растровом отображении соответствует 1,5 пиксела на экране. Иными словами, персонаж 32 × 32 пиксела будет на экране иметь размер 48 × 48 пикселов. Коэффициент масштабирования можно подсчитать с помощью следующих формул:

```
Коэффициент масштабирования (по оси x) = ширина экрана в пикселах / целевая ширина
в пикселах
```

и

```
Коэффициент масштабирования (по оси y) = высота экрана в пикселах / целевая высота
в пикселах
```

Целевые ширина и высота равны разрешению экрана, для которого было разработано приложение. Так, к примеру, в Replica Island это 480 × 320 пикселов. В случае с Nexus One это значит, что коэффициент масштабирования по оси x равен 1,66, а по оси y — 1,5. Но почему коэффициент масштабирования на разных осях различный?

Дело в том, что экраны с разными разрешениями имеют различные соотношения сторон. Если мы просто растянем изображение 480 × 320 пикселов до изображения в 800 × 480 пикселов, оригинальная картинка будет растянута по оси x. Для большинства игр это будет некритично, так что мы просто нарисуем нужные графические объекты для определенного разрешения и растянем их до реального разрешения экрана во время визуализации (вспомните метод `Bitmap.drawBitmap()`).

Для некоторых игр тем не менее придется немного потрудиться. На рис. 5.3 показана просто отмасштабированная с 480 × 320 до 800 × 480 пикселов Replica Island. Кроме того, наложено полупрозрачное изображение того, как это выглядит в действительности.

Создатели Replica Island использовали здесь весьма тонкий прием: мы видим нормальное растяжение по y-оси с коэффициентом масштабирования, который мы только что вычислили (1,5). Но вместо того, чтобы использовать коэффициент масштабирования (1,66), который сделает изображение более сжатым, здесь применяется коэффициент масштабирования y-оси. Подобная уловка позволяет всем объектам на экране сохранить соотношение сторон. Спрайт размером 32 × 32 пиксела увеличивается до 48 × 48 пикселов, а не 53 × 48 пикселов. Однако это также значит, что наша система координат больше не ограничивается (0; 0) и (479; 319). Вместо этого теперь ее границы (0; 0) и (533; 319). Именно поэтому мы видим больше игрового пространства Replica Island на Nexus One, чем на HTC Hero.

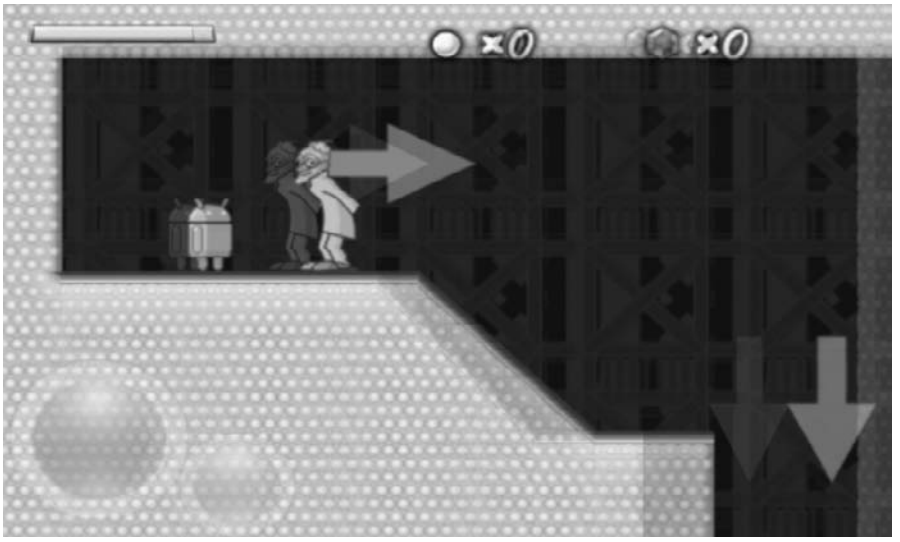

Рис. 5.3. Replica Island, растянутая с 480 × 320 до 800 × 480 пикселов с наложенным полупрозрачным изображением того, как это выглядит в действительности на экране 800 × 480

Обратите внимание, что в некоторых играх данный метод неприменим. Например, зависимость размера мира от соотношения сторон экрана может привести к несправедливому преимуществу владельцев более широких экранов. Например, данный метод неприменим для таких игр, как Starcraft 2. Однако если вы хотите поместить весь мир игры на одном экране (как в «Мистере Номе»), лучше использовать простой метод растяжения, так как с более сложной версией у нас будут пустые зоны по бокам на более широких экранах.

Решение попроще

У Replica Island есть одно преимущество: здесь растяжение и масштабирование выполнены с помощью OpenGL ES, которое поддерживает аппаратное ускорение. Пока мы обсудили только, как нарисовать `Bitmap` и `View` с помощью класса `Canvas`, который не использует графический процессор, но применяет более медленный в таких ситуациях центральный процессор.

Теперь применим небольшую уловку: создадим фреймбуфер в виде экземпляра класса `Bitmap`, имеющего нужное нам разрешение. Таким образом, нам не придется волноваться о реальном разрешении экрана, когда мы будем создавать наши графические объекты или когда будем их визуализировать. Давайте предположим, что разрешение экрана одинаково на всех устройствах. Когда мы закончим визуализацию фрейма нашей игры, мы просто нарисуем этот фреймбуфер `Bitmap` размером с `SurfaceView` с помощью метода `Canvas.drawBitmap()`, который позволяет изобразить растянутый `Bitmap`.

Если мы хотим использовать тот же прием, что и Replica Island, просто надо подстроить размер фреймбуфера по большей оси (например, по оси *x* в ландшафт-

ной ориентации и по оси *y* в портретной). Необходимо также удостовериться, что мы заполнили дополнительные пикселы, чтобы не было пустых промежутков.

Реализация

Подведем промежуточные итоги, составив план действий.

- Создаем графические объекты для фиксированного целевого разрешения (320 × 480 в случае с «Мистером Номом»).
- Создаем `Bitmap` такого же размера, как и наше целевое разрешение, и направляем все вызовы отрисовки к нему, работая в фиксированной системе координат.
- Когда закончим рисовать кадр игры, рисуем фреймбуфер `Bitmap`, растянутый до размера `SurfaceView`. На устройствах с меньшим разрешением масштаб изображения уменьшится, на устройствах с большим разрешением — увеличится.
- При масштабировании убедимся, что все элементы интерфейса, с которыми взаимодействует пользователь, достаточно велики на экранах разной плотности. Это можно сделать на стадии графического дизайна объектов, применяя размеры реальных устройств вместе с формулами, рассмотренными выше.

Теперь, когда мы уже знаем, как обращаться с различными разрешениями и плотностями экрана, можно обсудить переменные `scaleX` и `scaleY`, с которыми мы встречались, когда реализовывали обработчики `SingleTouchHandler` и `Multitouchhandler` несколько страниц назад.

Весь код нашей игры будет ориентирован на работу с фиксированным разрешением (320 × 480 пикселов). Если мы получаем события касания на устройствах, которые имеют более высокое или более низкое разрешение, *x*- и *y*-координаты данных событий будут описаны в системе координат `View`, а не в системе координат нашего целевого разрешения. Таким образом, необходимо перевести координаты из оригинальной системы в нашу систему, основанную на коэффициенте масштабирования. Вот как это можно сделать:

```
Преобразованное касание по оси x = реальное касание по оси x * (целевое количество
пикселов по оси x / реальное количество пикселов по оси x)
Преобразованное касание по оси y = реальное касание по оси y * (целевое количество
пикселов по оси y / реальное количество пикселов по оси y)
```

Рассмотрим простой пример, где целевое разрешение равно 320 × 480 пикселов, а разрешение устройства — 480 × 800 пикселов. Если мы дотронемся до середины экрана, то получим координаты (240; 400). Используя две предыдущие формулы, получаем следующие данные, которые являются серединой нашей целевой системы координат:

```
Преобразованное касание по оси x = 240 * (320 / 480) = 160
Преобразованное касание по оси y = 400 * (480 / 800) = 240
```

Решим еще один пример, в котором реальное разрешение равно 240 × 320, а экрана снова коснулись в середине (120; 160):

```
Преобразованное касание по оси x = 120 * (320 / 240) = 160
Преобразованное касание по оси y = 160 * (480 / 320) = 240
```

Как видите, данный прием работает в обоих направлениях. Если мы умножим координаты реального события касания на целевой коэффициент, разделенный на реальный коэффициент, нам не придется отвлекаться на все эти преобразования в коде игры. Все координаты касаний будут отображены в нашей фиксированной целевой системе координат.

Поскольку мы справились и с этой проблемой, пришло время реализовать последние несколько классов фреймворка нашей игры.

AndroidPixmap: пикселы для каждого

Согласно проекту интерфейса нашего `Pixmap` из главы 3, нам осталось реализовать не так уж много. Рассмотрим код в листинге 5.12.

Листинг 5.12. AndroidPixmap.java, реализация Pixmap

```
package com.badlogic.androidgames.framework.impl;

import android.graphics.Bitmap;

import com.badlogic.androidgames.framework.Graphics.PixmapFormat;
import com.badlogic.androidgames.framework.Pixmap;

public class AndroidPixmap implements Pixmap {
    Bitmap bitmap;
    PixmapFormat format;

    public AndroidPixmap(Bitmap bitmap, PixmapFormat format) {
        this.bitmap = bitmap;
        this.format = format;
    }

    @Override
    public int getWidth() {
        return bitmap.getWidth();
    }

    @Override
    public int getHeight() {
        return bitmap.getHeight();
    }

    @Override
    public PixmapFormat getFormat() {
        return format;
    }

    @Override
    public void dispose() {
        bitmap.recycle();
    }
}
```

Все, что нам нужно сделать, — сохранить экземпляр класса Bitmap, а также его формат, который хранится в виде одного из значений перечисления PixmapFormat, как это описано в главе 3. Дополнительно мы реализуем необходимые методы интерфейса Pixmap, чтобы можно было запрашивать ширину и высоту Pixmap и его формат, а также удостовериться, что пикселы извлекаются из оперативной памяти. Обратите внимание, что поле bitmap является приватным, так что у нас есть к нему доступ из AndroidGraphics, который мы сейчас и реализуем.

AndroidGraphics: то, что нужно для рисования

Интерфейс Graphics, созданный нами в главе 3, достаточно компактный и эффективный. Он может рисовать пикселы, линии, прямоугольники и Pixmap в фреймбуфере. Как мы уже говорили, мы будем использовать Bitmap в качестве нашего фреймбуфера и рисовать на нем с помощью Canvas. Он также отвечает за создание экземпляров класса Pixmap из файла ресурсов. Таким образом нам снова понадобится AssetManager. В листинге 5.13 показан код реализации интерфейса AndroidGraphics.

Листинг 5.13. AndroidGraphics.java: реализуем графический интерфейс

```
package com.badlogic.androidgames.framework.impl;

import java.io.IOException;
import java.io.InputStream;

import android.content.res.AssetManager;
import android.graphics.Bitmap;
import android.graphics.Bitmap.Config;
import android.graphics.BitmapFactory;
import android.graphics.BitmapFactory.Options;
import android.graphics.Canvas;

import android.graphics.Paint;
import android.graphics.Paint.Style;
import android.graphics.Rect;

import com.badlogic.androidgames.framework.Graphics;
import com.badlogic.androidgames.framework.Pixmap;

public class AndroidGraphics implements Graphics {
    AssetManager assets;
    Bitmap frameBuffer;
    Canvas canvas;
    Paint paint;
    Rect srcRect = new Rect();
    Rect dstRect = new Rect();
```

Данный класс реализует интерфейс Graphics. Он содержит член AssetManager, который нам нужен для того, чтобы загружать экземпляры Bitmap, член Bitmap, представляющий собой искусственный фреймбуфер, член Canvas, используемый для того, чтобы нарисовать искусственный фреймбуфер, член Paint, который необходим для рисования, и два члена Rect, которые нам понадобятся для реализации

AndroidGraphics.drawPixmap(). Последние три члена используются для того, чтобы не создавать новые экземпляры классов при каждом вызове метода рисования. Мы же не хотим, чтобы сборщик мусора сошел с ума.

```
public AndroidGraphics(AssetManager assets, Bitmap frameBuffer) {
    this.assets = assets;
    this.frameBuffer = frameBuffer;
    this.canvas = new Canvas(frameBuffer);
    this.paint = new Paint();
}
```

В конструкторе получаем AssetManager и Bitmap, которые представляют наш искусственный фреймбуфер. Сохраняем их в соответствующих полях и дополнительно создаем экземпляр класса Canvas, который будет рисовать в искусственном фреймбуфере, а также Paint, который мы используем для некоторых методов рисования.

```
@Override
public Pixmap newPixmap(String fileName, PixmapFormat format) {
    Config config = null;
    if (format == PixmapFormat.RGB565)
        config = Config.RGB_565;
    else if (format == PixmapFormat.ARGB4444)
        config = Config.ARGB_4444;
    else
        config = Config.ARGB_8888;

    Options options = new Options();
    options.inPreferredConfig = config;

    InputStream in = null;
    Bitmap bitmap = null;
    try {
        in = assets.open(fileName);
        bitmap = BitmapFactory.decodeStream(in);
        if (bitmap == null)
            throw new RuntimeException("Couldn't load bitmap from asset '"
                    + fileName + "'"); }
        catch (IOException e) {
            throw new RuntimeException("Couldn't load bitmap from asset '"
                    + fileName + "'");
        } finally {
            if (in != null) {
                try {
                    in.close();
                } catch (IOException e) {
                }
            }
        }

        if (bitmap.getConfig() == Config.RGB_565)
```

```
            format = PixmapFormat.RGB565;
        else if (bitmap.getConfig() == Config.ARGB_4444)
            format = PixmapFormat.ARGB4444;
        else
            format = PixmapFormat.ARGB8888;

        return new AndroidPixmap(bitmap, format);
    }
```

Метод `newPixmap()` загружает `Bitmap` из файла объектов, используя заданный `PixmapFormat`. Мы начинаем с того, что переводим `PixmapFormat` в одну из констант класса Android `Config`, который мы использовали в главе 4. Далее создаем экземпляр класса `Options` и устанавливаем предпочтительный формат цвета. После этого загружаем `Bitmap` из ресурса с помощью `BitmapFactory`. Если что-то идет не так, генерируется исключение `RuntimeException`. В противном случае мы проверяем, в каком формате фабрика `BitmapFactory` решила загрузить `Bitmap`, и переводим его в значение перечисления `PixmapFormat`. Помните, что `BitmapFactory` может решить игнорировать наш предпочитаемый формат цвета, так что необходимо будет проверить, как она закодировала изображение. Наконец мы создаем новый экземпляр класса `AndroidBitmap`, основанный на экземпляре `Bitmap`, который мы загрузили, и `PixmapFormat`, а затем возвращаем этот новый экземпляр вызывающей стороне.

```
@Override
public void clear(int color) {
    canvas.drawRGB((color & 0xff0000) >> 16, (color & 0xff00) >> 8,
        (color & 0xff));
}
```

Метод `clear()` просто извлекает красный, зеленый и синий компоненты из определенного 32-битного ARGB цветового параметра и вызывает метод `Canvas.drawRGB()`, который очищает наш искусственный фреймбуфер с этим цветом. Этот метод не учитывает никакие альфа-значения определенного цвета, так что нам не надо его извлекать.

```
@Override
public void drawPixel(int x, int y, int color) {
    paint.setColor(color);
    canvas.drawPoint(x, y, paint);
}
```

Метод `drawPixel()` рисует пиксел в нашем искусственном фреймбуфере с помощью метода `Canvas.drawPoint()`. Для начала устанавливаем цвет в поле класса `paint` и передаем эти данные методу рисования в дополнение к x- и y-координатам пиксела.

```
@Override
public void drawLine(int x, int y, int x2, int y2, int color) {
    paint.setColor(color);
    canvas.drawLine(x, y, x2, y2, paint);
}
```

Метод drawLine() рисует линию в искусственном фреймбуфере, также используя поле paint, чтобы задать цвет, который будет применяться при вызове Canvas.drawLine().

```
@Override
public void drawRect(int x, int y, int width, int height, int color) {
    paint.setColor(color);
    paint.setStyle(Style.FILL);
    canvas.drawRect(x, y, x + width - 1, y + width - 1, paint);
}
```

Метод drawRect() сначала устанавливает цвет в Paint и атрибут стиля, чтобы мы могли нарисовать заполненный цветом прямоугольник. Для самого вызова Canvas.drawRect() нам потребуется преобразовать параметры x, y, width и height в координаты верхнего левого и нижнего правого углов прямоугольника. Для верхнего левого угла просто используем параметры x и y. Для координат нижнего правого угла добавляем ширину и высоту к x и y и отнимаем 1. Для примера представьте, что мы визуализируем прямоугольник, где x и y имеют координаты (10; 10), а ширина и высота равны 2. Если мы не отнимем 1, в итоге прямоугольник на экране будет размером 3×3 пиксела.

```
@Override
public void drawPixmap(Pixmap pixmap, int x, int y, int srcX, int srcY,
        int srcWidth, int srcHeight) {
    srcRect.left = srcX;
    srcRect.top = srcY;
    srcRect.right = srcX + srcWidth - 1;
    srcRect.bottom = srcY + srcHeight - 1;

    dstRect.left = x;
    dstRect.top = y;
    dstRect.right = x + srcWidth - 1;
    dstRect.bottom = y + srcHeight - 1;

    canvas.drawBitmap(((AndroidPixmap) pixmap).bitmap, srcRect, dstRect,
        null);
}
```

Метод drawPixmap(), позволяющий нарисовать часть Pixmap, сначала устанавливает исходный прямоугольник и прямоугольник назначения в соответствующие поля, которые используются для вызова метода рисования. Поскольку мы рисуем прямоугольник, нам будет нужно перевести координаты x и y, а также ширину и высоту в координаты левого верхнего и правого нижнего углов. Мы снова отнимаем 1, иначе у нас получится один лишний пиксел. Далее выполняем рисование с помощью метода Canvas.drawBitmap(), который также автоматически выполнит смешивание, если Pixmap, который мы рисуем, имеет глубину цвета PixmapFormat.ARGB4444 или PixmapFormat.ARGB8888. Обратите внимание, что нам необходимо привести параметр типа Pixmap к типу AndroidPixmap, чтобы мы могли выбрать член bitmap для рисования с помощью Canvas. Так делать не рекомендуется, но в данном

случае мы можем быть уверены что переданный экземпляр Pixmap действительно будет представлять собой AndroidPixmap.

```
@Override
    public void drawPixmap(Pixmap pixmap, int x, int y) {
        canvas.drawBitmap(((AndroidPixmap)pixmap).bitmap, x, y, null);
    }
```

Второй метод drawPixmap() просто рисует весь Pixmap в искусственном фреймбуфере в заданных координатах. Мы снова используем приведение для того, чтобы добраться до члена bitmap класса AndroidPixmap.

```
@Override
    public int getWidth() {
        return frameBuffer.getWidth();
    }

    @Override
    public int getHeight() {
        return frameBuffer.getHeight();
    }
}
```

Наконец, у нас есть методы getWidth() и getHeight(), которые просто возвращают размер искусственного фреймбуфера, хранящего и отображающего экземпляр класса AndroidGraphics.

Нам нужно реализовать еще один класс, который относится к графике, — AndroidFastRenderView.

AndroidFastRenderView: собрать-растянуть, собрать-растянуть

По названию данного класса, которое переводится примерно как «вид для быстрого отображения в Android», вы можете сами догадаться, что нам предстоит. В предыдущей главе мы обсуждали использование SurfaceView для продолжительной визуализации в отдельном потоке, который также может содержать основной цикл нашей игры.

Мы разработали очень простой класс FastRenderView, унаследованный от SurfaceView. Убедились, что вписываемся в жизненный цикл активности, и настроили поток, в котором происходит визуализация в SurfaceView с помощью Canvas.

Мы вновь воспользуемся классом FastRenderView и дополним его, чтобы он мог выполнять еще несколько задач:

- хранить ссылку на экземпляр класса Game, чтобы иметь возможность получить активный Screen. Мы постоянно вызываем методы Screen.update() и Screen.present() из потока FastRenderView;
- следить за дельтой времени между кадрами, которая передается активному Screen;

- принимать искусственный фреймбуфер, в котором функционирует экземпляр AndroidGraphics, и передавать это изображение в SurfaceView, при необходимости масштабируя его.

В листинге 5.14 показано, как реализовать класс AndroidFastRenderView.

Листинг 5.14. AndroidFastRenderView.java, поток SurfaceView выполняет код нашей игры

```
package com.badlogic.androidgames.framework.impl;

import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Rect;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class AndroidFastRenderView extends SurfaceView implements Runnable {
    AndroidGame game;
    Bitmap framebuffer;
    Thread renderThread = null;
    SurfaceHolder holder;
    volatile boolean running = false;
```

Это должно быть вам уже знакомо. Нам просто нужно хранить еще два члена: экземпляр класса AndroidGame и экземпляр класса Bitmap, которые представляют наш искусственный фреймбуфер. Другие члены аналогичны тем, что использованы в FastRenderView из главы 3.

```
public AndroidFastRenderView(AndroidGame game, Bitmap framebuffer) {
    super(game);
    this.game = game;
    this.framebuffer = framebuffer;
    this.holder = getHolder();
}
```

В конструкторе мы просто вызываем конструктор базового класса с параметром AndroidGame (который унаследован от Activity; мы вернемся к этому чуть позже) и сохраняем параметры в соответствующих полях. Мы также снова получаем SurfaceHolder, как мы это уже делали раньше.

```
public void resume() {
    running = true;
    renderThread = new Thread(this);
    renderThread.start();
}
```

Метод resume() является точной копией метода FastRenderView.resume(), так что нам не придется проходить через все это снова. Просто удостоверимся, что поток нормально работает с жизненным циклом активности.

```
public void run() {
    Rect dstRect = new Rect();
```

```
        long startTime = System.nanoTime();
        while(running) {
            if(!holder.getSurface().isValid())
                continue;

            float deltaTime = (System.nanoTime()-startTime) / 1000000000.0f;

            startTime = System.nanoTime();

            game.getCurrentScreen().update(deltaTime);
            game.getCurrentScreen().present(deltaTime);

            Canvas canvas = holder.lockCanvas();
            canvas.getClipBounds(dstRect);
            canvas.drawBitmap(framebuffer, null, dstRect, null);
            holder.unlockCanvasAndPost(canvas);
        }
    }
```

Метод `run()` имеет еще несколько особенностей. Он также отслеживает дельту времени между кадрами. Мы используем для этого функцию `System.nanoTime()`, возвращающую текущее время в наносекундах как длинное число.

ПРИМЕЧАНИЕ

Наносекунда — это одна миллиардная доля секунды.

На каждой итерации цикла мы начинаем с получения разницы между временем начала предыдущей итерации и текущим временем. Чтобы упростить работу с дельтой времени, переводим все в секунды. Далее сохраняем текущую временную метку, которую будем использовать на следующей итерации, чтобы подсчитать следующую дельту времени. С помощью данных о дельте времени вызовем методы `update()` и `present()` текущего `Screen`, которые обновят логику игры и визуализируют объекты в искусственном фреймбуфере. Наконец, удерживаем `Canvas` для `SurfaceView` и рисуем искусственный фреймбуфер. Масштабирование осуществляется автоматически в случае, если конечный прямоугольник, который мы передаем методу `Canvas.drawBitmap()`, меньше или больше, чем фреймбуфер.

Обратите внимание, что здесь мы использовали сокращенный способ получения конечного прямоугольника, растянутого на весь `SufaceView` при помощи метода `Canvas.getClipBounds()`. Он присвоит членам `top` и `left`, относящимся к `dstRect`, значение 0, а членам `bottom` и `right` — реальные размеры экрана (например, 480×800 в портретной ориентации в Nexus One). Оставшаяся часть метода полностью совпадает с тем, что мы делали в тесте `FastRenderView`. Она просто проверяет, что поток останавливается, когда активность ставится на паузу или удаляется.

```
public void pause() {
        running = false;
        while(true) {
            try {
```

```
                renderThread.join();
                break;
            } catch (InterruptedException e) {
                // повтор
            }
        }
    }
}
```

Последний метод в этом классе, `pause()`, полностью совпадает с методом `FastRenderView.pause()`. Он просто заканчивает визуализацию основного цикла потока и ожидает, пока этот цикл полностью закончит работу.

Мы почти закончили с нашим фреймворком. Все, что нам осталось, — реализовать интерфейс `Game`.

AndroidGame: свяжем все вместе

Мы почти закончили небольшой каркас нашей игры. Все, что осталось сделать, — связать все вместе, реализовав интерфейс `Game`, который мы создали в главе 3, с помощью классов, написанных в предыдущем разделе. Список задач выглядит следующим образом:

- организовать управление окнами. В нашем случае это означает «правильно обрабатывать жизненный цикл активности»;
- использовать `WakeLock` и отслеживать его работу, чтобы экран не тускнел;
- инстанцировать и отдавать ссылки на `Graphics`, `Audio`, `FileIO` и `Input` тем элементам, которым они требуются;
- управлять экземплярами `Screen` и интегрировать их в жизненный цикл активности.

Наша главная цель — получить единый класс `AndroidGame`, на котором мы можем основываться. Все, что мы хотим делать впоследствии, — реализовать метод `Game.getStartScreen()`, который запустит нашу игру, например вот так:

```
public class MrNom extends AndroidGame {
    @Override
    public Screen getStartScreen() {
        return new MainMenu(this);
    }
}
```

Надеюсь, вы понимаете, почему было так важно организовать хороший компактный фреймворк перед тем, как заняться непосредственно программированием самой игры. Мы сможем заново использовать данный фреймворк во всех наших будущих играх, которые не требуют интенсивной работы с графикой. Рассмотрим листинг 5.15, который демонстрирует класс `AndroidGame`.

Листинг 5.15. AndroidGame.java: свяжем все вместе

```
package com.badlogic.androidgames.framework.impl;

import android.app.Activity;
import android.content.Context;
import android.content.res.Configuration;
import android.graphics.Bitmap;
import android.graphics.Bitmap.Config;
import android.os.Bundle;
import android.os.PowerManager;
import android.os.PowerManager.WakeLock;
import android.view.Window;
import android.view.WindowManager;

import com.badlogic.androidgames.framework.Audio;
import com.badlogic.androidgames.framework.FileIO;
import com.badlogic.androidgames.framework.Game;

import com.badlogic.androidgames.framework.Graphics;
import com.badlogic.androidgames.framework.Input;
import com.badlogic.androidgames.framework.Screen;

public abstract class AndroidGame extends Activity implements Game {
    AndroidFastRenderView renderView;
    Graphics graphics;
    Audio audio;
    Input input;
    FileIO fileIO;
    Screen screen;
    WakeLock wakeLock;
```

Определение класса начинается с того, что AndroidGame дополняет класс Activity и реализует интерфейс Game. Далее определяем несколько членов, которые вам уже знакомы. Первый — AndroidFastRenderView, в котором мы будем рисовать и который будет управлять потоком основного цикла. Члены Graphics, Audio, Input и FileIO будут хранить экземпляры классов AndroidGraphics, AndroidAudio, AndroidInput и AndroidFileIO. Как видите, ничего удивительного тут не происходит. Следующий член хранит текущий Screen. И наконец, еще один член отвечает за WakeLock, который нужен для того, чтобы экран не гаснул.

```
@Override
    public void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);

        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
            WindowManager.LayoutParams.FLAG_FULLSCREEN);

        boolean isLandscape = getResources().getConfiguration().orientation
```

```
                        == Configuration.ORIENTATION_LANDSCAPE;
        int frameBufferWidth = isLandscape ? 480 : 320;
        int frameBufferHeight = isLandscape ? 320 : 480;
        Bitmap frameBuffer = Bitmap.createBitmap(frameBufferWidth,
                frameBufferHeight, Config.RGB_565);

        float scaleX = (float) frameBufferWidth
                / getWindowManager().getDefaultDisplay().getWidth();
        float scaleY = (float) frameBufferHeight
                / getWindowManager().getDefaultDisplay().getHeight();

        renderView = new AndroidFastRenderView(this, frameBuffer);
        graphics = new AndroidGraphics(getAssets(), frameBuffer);
        fileIO = new AndroidFileIO(getAssets());
        audio = new AndroidAudio(this);
        input = new AndroidInput(this, renderView, scaleX, scaleY);
        screen = getStartScreen();
        setContentView(renderView);

        PowerManager powerManager = (PowerManager)
                    getSystemService(Context.POWER_SERVICE);
        wakeLock = powerManager.newWakeLock(PowerManager.FULL_WAKE_LOCK,
                    "GLGame");
}
```

Уже знакомый метод `onCreate()`, вызываемый при запуске `Activity`, начинает свою работу с вызова метода базового класса `onCreate()`, как это и требуется. Далее мы делаем активность `Activity` полноэкранной, как это происходило в тестах из предыдущей главы. В следующих нескольких строках устанавливаем искусственный фреймбуфер. В зависимости от ориентации активности нам необходим фреймбуфер размером либо 320×480 (в книжной ориентации), либо 480×320 (в альбомной ориентации). Чтобы определить ориентацию экрана, которую использует `Activity`, выбираем член `orientation` из класса `Configuration`, который мы получаем с помощью вызова `getResources().getConfiguration()`. Основываясь на значениях данного члена, устанавливаем размер фреймбуфера и содаем `Bitmap`, который мы передадим экземплярам классов `AndroidFastRenderView` и `AndroidGraphics` чуть позже.

ПРИМЕЧАНИЕ

Экземпляр класса Bitmap имеет цветовой формат RGB565. Таким образом, нам не придется тратить память, а все наши рисунки будут обрабатываться немного быстрее.

Мы также можем вычислить значения `scaleX` и `scaleY` для классов `SingleTouchHandler` и `Multitouchhandler`, которые необходимы для перевода координат событий касания в нашу систему фиксированных координат.

Далее инстанцируем `AndroidFastRenderView`, `AndroidGraphics`, `AndroidAudio`, `AndroidInput` и `AndroidFileIO` с необходимыми значениями аргументов конструктора. Потом вызываем метод `getStartScreen()`, который реализуется нашей игрой, и задаем `AndroidFastRenderView` как вид с контентом `Activity`. Все эти вспомогательные классы, которые мы только что инстанцировали, конечно, будут выполнять

еще кое-какую работу в фоновом режиме. Например, класс AndroidInput свяжет свой обработчик касаний с видом AndroidFastRenderView.

```
@Override
public void onResume() {
    super.onResume();
    wakeLock.acquire();
    screen.resume();
    renderView.resume();
}
```

Переходим к методу onResume() из класса Activity. Как обычно, первое, что мы делаем, — вызываем метод родительского класса, как это и положено при программировании на Android. Далее получаем WakeLock и убеждаемся, что Screen получил информацию о том, что игра (а соответственно, и активность) была возобновлена. Затем просим AndroidFastRenderView возобновить поток визуализации, также запускающий основной цикл игры, в котором мы указываем текущему Screen обновить данные и отобразиться на каждой интерации.

```
@Override
public void onPause() {
    super.onPause();
    wakeLock.release();
    renderView.pause();
    screen.pause();

    if (isFinishing())
        screen.dispose();
}
```

Метод onPause() снова вызывает метод родительского класса. Далее он высвобождает WakeLock и проверяет, завершен ли поток визуализации. Если мы не завершили поток визуализации до того, как вызывали текущий onPause()Screen, могут возникнуть проблемы конкурентного доступа, так как потоку пользовательского интерфейса и потоку основного цикла одновременно нужен Screen. Когда мы удостоверились, что поток основного цикла завершен, мы приказываем текущему Screen остановиться. Если Activity должна быть удалена, мы также сообщаем Screen об этом событии, чтобы он мог выполнить необходимую очистку.

```
@Override
public Input getInput() {
    return input;
}

@Override
public FileIO getFileIO() {
    return fileIO;
}

@Override
public Graphics getGraphics() {
```

```
    return graphics;
}

@Override
public Audio getAudio() {
    return audio;
}
```

Пожалуй, методы `getInput()`, `getFileIO()`, `getGraphics()` и `getAudio()` понятны без объяснения. Мы просто возвращаем вызывающей стороне соответствующие экземпляры классов. Вызывающая сторона всегда будет одной из реализаций `Screen` нашей игры.

```
@Override
public void setScreen(Screen screen) {
    if (screen == null)
        throw new IllegalArgumentException("Screen must not be null");

    this.screen.pause();
    this.screen.dispose();
    screen.resume();
    screen.update(0);
    this.screen = screen;
}
```

Метод `setScreen()`, который мы наследуем от интерфейса `Game`, на первый взгляд кажется достаточно простым. Поскольку `null` в качестве значения `Screen` нам не подходит, выполняем классическую `null`-проверку. Далее указываем текущему `Screen` остановиться и уничтожить себя, чтобы мы могли освободить место для нового `Screen`. Приказываем новому `Screen` возобновиться и обновиться с дельтой времени, равной нулю. Далее присваиваем полю `Screen` значение нового `Screen`.

Давайте задумаемся, кто и когда должен вызывать метод `setScreen()`. Когда мы создавали «Мистера Нома», мы определили все переходы между различными экземплярами класса `Screen`. Обычно мы вызываем метод `AndroidGame.setScreen()` в методе `update()` одного из экземпляров этого класса `Screen`.

Допустим, у нас есть `Screen` главного меню, где мы можем проверить, нажата ли кнопка **Play** (Воспроизвести) в методе `update()`. В этом случае нам понадобится переход к следующему `Screen`, и мы можем его выполнить с помощью вызова метода `AndroidGame.setScreen()` из метода `MainMenu.update()` с новым экземпляром класса следующего экрана `Screen`. После вызова `AndroidGame.setScreen()` экран `MainMenu` получит управление, после чего он должен сразу же вернуть управление, поскольку не является активным `Screen`. В данном случае вызывающая сторона — это `AndroidFastRenderView` в потоке основного цикла. Если рассмотреть ту часть основного цикла, которая отвечает за обновление и визуализацию активного `Screen`, то видно, что метод `update()` будет вызываться для класса `MainMenu`, а метод `present()` будет вызываться для нового текущего `Screen`. Этого лучше избежать, поскольку мы определили интерфейс `Screen` таким образом, что методы `resume()` и `update()` должны быть вызваны как минимум однажды перед тем, как `Screen` должен будет

отобразить себя. Поэтому мы вызываем два этих метода в методе AndroidGame.setScreen() для нового Screen. Мощный класс AndroidGame выполнит всю сопутствующую работу.

```
public Screen getCurrentScreen() {
        return screen;
    }
}
```

Последний метод называется getCurrentScreen(). Он просто возвращает текущий активный Screen.

Таким образом, мы создали каркас простой игры для Android. Все, что нам осталось сделать, — реализовать Screen нашей игры. Мы можем также заново использовать данный фреймворк для любых будущих игр, если они не требуют значительной мощности при обработке графики (иначе не обойтись без OpenGL ES). Следует также заменить графическую часть фреймворка. Все остальные классы для аудио, ввода и файлового ввода-вывода могут быть использованы заново.

Подводя итог

В этой главе мы с нуля создали полноценный фреймворк для 2D-игры на Android, который сможем использовать в будущем для других игр (если в них нет какой-то запредельной графики). Достаточно подробно был рассмотрен удобный и расширяемый дизайн. Чтобы наш мистер Ном стал 3D-игрой, необходимо просто взять этот код и заменить в нем части кода, отвечающие за визуализацию, с помощью OpenGL ES.

Теперь, когда фреймворк готов, сконцентрируемся на нашей основной задаче — написании игры!

6 «Мистер Ном» покоряет Android

В главе 3 мы продумали полный дизайн «Мистера Нома», который состоит из механики игры, простого фона, нарисованных вручную графических объектов и описаний для всех экранов, основанных на бумажных вырезках. В предыдущей главе мы разработали полноценный фреймворк игры, который позволяет легко реализовать спроектированные экраны в коде. Но хватит разговоров, приступим к написанию нашей первой игры!

Создание ресурсов

Для «Мистера Нома» нам потребуется два типа ресурсов: звуки и графика. Я записал звуки с помощью удобного бесплатного приложения Audacity и непритязательного микрофона с нетбука. Я создал аудиоэффект для случаев, когда нажата кнопка или выбран пункт меню и когда мистер Ном съедает пятно, а также когда он съедает сам себя. Я сохранил их в формате OGG в папке `assets/` под именами `click.ogg`, `eat.ogg` и `bitten.ogg` соответственно.

Ранее я упомянул, что нам еще понадобятся вырезки из бумаги, которые мы использовали на стадии проектирования графики реальной игры. Итак, нам необходимо сначала сделать так, чтобы они соответствовали нашему целевому разрешению.

Я выбрал фиксированное целевое разрешение 320 × 480 (книжная ориентация), для которого мы будем разрабатывать все наши графические ресурсы. Отсканировал все бумажные вырезки и немного изменил их размер. Сохранил большинство ресурсов в отдельных файлах, а также объединил некоторые из них в один файл. Все изображения сохранил в формате PNG. Лишь фон сохранен в формате RGB888, все остальные ресурсы сохранены в формате ARGB8888. На рис. 6.1 показано, что в итоге получилось.

Рассмотрим эти изображения подробнее.

- `background.png` — это изображение фона, которое будет нарисовано в нашем фреймбуфере в первую очередь. По вполне очевидным причинам оно имеет такой же размер, как и конечное разрешение.

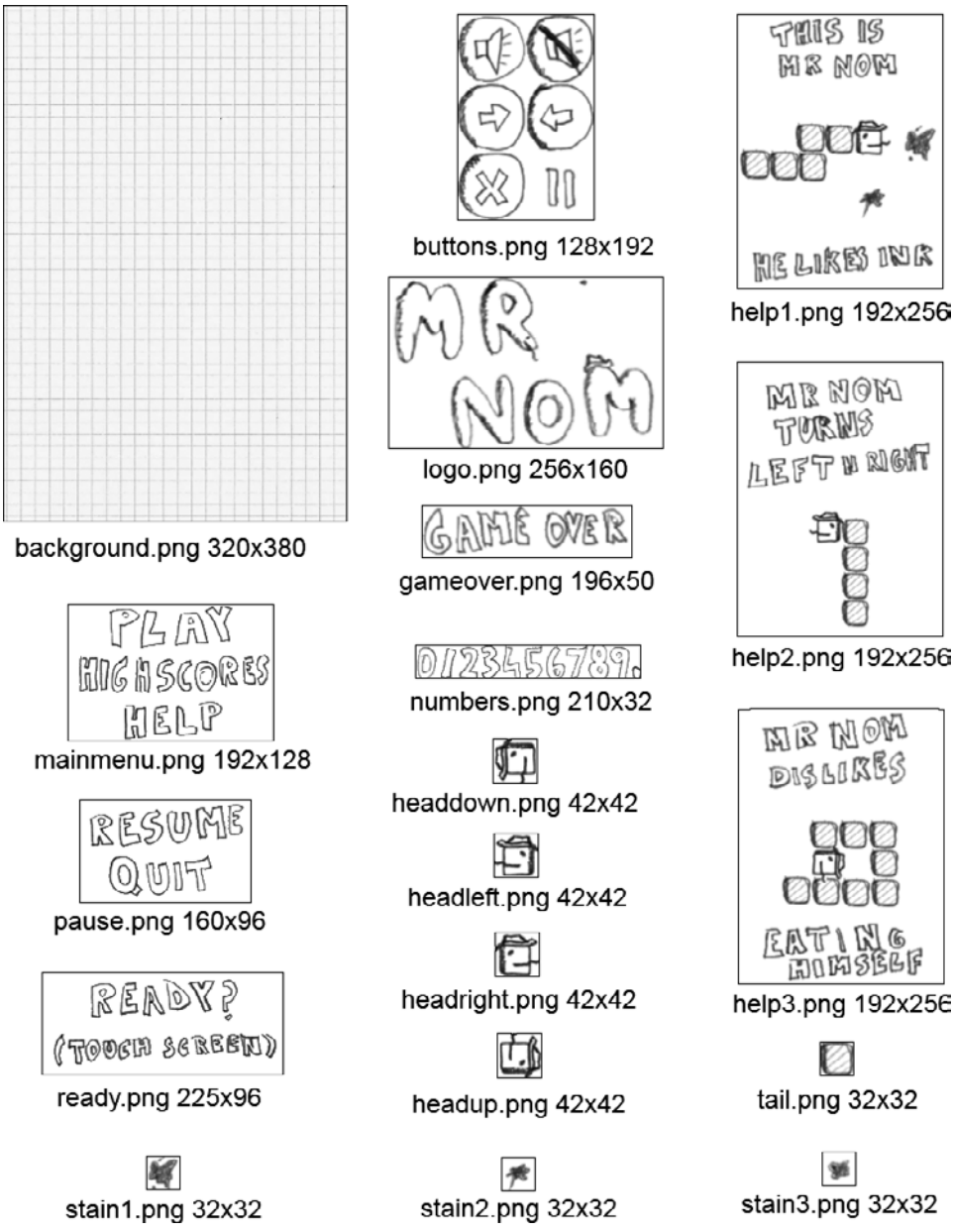

Рис. 6.1. Все графические объекты «Мистера Нома» с соответствующими названиями файлов и размерами в пикселах

- `buttons.png` — данный файл содержит все кнопки, которые понадобятся для игры. Я поместил их все в один файл, так как мы можем легко создать их с помощью метода `Graphics.drawPixmap()`, который может рисовать фрагменты изображения. Мы будем использовать эту технику гораздо чаще, когда станем

рисовать с помощью OpenGL ES, так что лучше привыкнуть к ней прямо сейчас. Объединение нескольких изображений в одно часто называют минимизацией количества текстур (atlasing), а само изображение называют атласом изображений (другие названия: атлас текстур, таблица изображений). Каждая кнопка имеет размер 64 × 64 пикселов, что весьма удобно в случае, когда приходится решать, была ли нажата клавиша при событии касания.

- `help1.png`, `help2.png`, `help3.png` — это изображения, которые показываются при вызове экранов помощи в «Мистере Номе». Все они имеют одинаковый размер, что значительно упрощает их размещение на экране.
- `logo.png` — это логотип, который будет показываться на экране главного меню.
- `mainmenu.png` — данный файл содержит три параметра, которые мы предлагаем пользователю в главном меню. Выбор одного из них приведет к переходу к соответствующему экрану. Высота изображения каждого параметра — примерно 42 пиксела. Эти данные нам понадобятся для того, чтобы определить, какой параметр был выбран.
- `ready.png`, `pause.png`, `gameover.png` — эти ресурсы будут отрисованы, когда игра, соответственно, будет готова к запуску, поставлена на паузу или закончена.
- `numbers.png` — данный файл содержит все цифры, которые необходимы для того, чтобы позже отображать таблицу рекордов. Следует запомнить, что каждая цифра имеет одинаковую ширину и высоту 20 × 32 пиксела за исключением точки в конце, которая имеет размер 10 × 32 пиксела. Мы можем позже использовать этот способ для визуализации любого необходимого числа.
- `tail.png` — это хвост мистера Нома, вернее часть его хвоста. Его размер 32 × 32 пиксела. С хвостом связаны некоторые особенности, которые мы обсудим позже.
- `headdown.png`, `headleft.png`, `headright.png`, `headup.png` — данные изображения составляют голову мистера Нома, каждый из них для соответствующего направления движения. Поскольку на голове у мистера Нома есть шляпа, эти изображения должны быть немного больше хвоста. Каждое изображение головы имеет размер 42 × 42 пиксела.
- `stain1.png`, `stain2.png`, `stain3.png` — даные файлы представляют собой три типа пятен, которые мы визуализируем. Наличие трех типов пятен сделает игру более разнообразной. Они все размером 32 × 32 пиксела, как и изображение хвоста.

Теперь давайте реализуем экраны.

Настройка проекта

Как мы уже говорили в предыдущей главе, мы объединим код мистера Нома с кодом фреймворка. Все классы, касающиеся мистера Нома, будут помещены в пакет `com.badlogic.androidgames.mrnom`. Необходимо также изменить файл манифест так, как это описано в главе 4. Наша главная активность будет называться `MrNomGame`. Чтобы правильно установить атрибуты `<activity>`, просто выполните десять шагов, описан-

ных в подразделе «Настройка проекта игры на Android за 10 простых шагов» раздела «Определение приложения Android: файл манифеста» главы 4 (например, игра должна быть зафиксирована в книжной ориентации, изменения в конфигурации обрабатываться приложением), а также предоставьте приложению необходимые права (запись на внешний носитель, препятствие потуханию экрана и т. д.).

Все ресурсы из предыдущих разделов находятся в папке проекта `assets/`. Дополнительно нам необходимо поместить файлы `icon.png` в папки `res/drawable`, `res/drawable-ldpi`, `res/drawable-mdpi` и `res/drawable-hdpi`. Я просто взял `headright.png` мистера Нома, переименовал в `icon.png` и поместил версии нужного размера в каждую папку.

Теперь мы переместим все это в пакет `com.badlogic.androidgames.mrnom` проекта Eclipse.

MrNomGame: основная активность

Нашему приложению необходима точка входа, которая также именуется в Android активностью, задаваемой по умолчанию. Мы назовем эту активность `MrNomGame` и унаследуемся от `AndroidGame`, класса, который мы реализовали в главе 5 для того, чтобы написать игру. Этот класс необходим для создания и запуска нашего первого экрана. В листинге 6.1 показан класс `MrNomGame`.

Листинг 6.1. `MrNomGame.java`, наш главный гибрид активности и игры

```
package com.badlogic.androidgames.mrnom;

import com.badlogic.androidgames.framework.Screen;
import com.badlogic.androidgames.framework.impl.AndroidGame;

public class MrNomGame extends AndroidGame {
    @Override
    public Screen getStartScreen() {
        return new LoadingScreen(this);
    }
}
```

Все, что необходимо сделать, — унаследовать от `AndroidGame` и реализовать метод `getStartScreen()`, который вернет экземпляр класса `LoadingScreen` (мы реализуем его буквально через минуту). Это позволит нам приступить к самым важным вопросам, которые нужно решить для написания нашей игры: от настройки различных модулей звука, графики, ввода и файлового ввода/вывода до запуска потока основного цикла. Как видите, все достаточно просто, не так ли?

Ресурсы: удобное хранилище ресурсов

Загрузочный экран загрузит все объекты игры. Однако где же мы будем их хранить? Чтобы нам было удобно с ними обращаться, сделаем кое-что, не совсем типичное для Java: создадим класс, имеющий множество статических общедоступных членов.

В них будут содержаться все Pixmap и Sound, которые мы загрузили из ресурсов (весь класс показан в листинге 6.2).

Листинг 6.2. Assets.java, здесь для быстрого доступа содержатся все ресурсы Pixmap и Sound

```
package com.badlogic.androidgames.mrnom;

import com.badlogic.androidgames.framework.Pixmap;
import com.badlogic.androidgames.framework.Sound;

public class Assets {
    public static Pixmap background;
    public static Pixmap logo;
    public static Pixmap mainMenu;
    public static Pixmap buttons;
    public static Pixmap help1;
    public static Pixmap help2;
    public static Pixmap help3;
    public static Pixmap numbers;
    public static Pixmap ready;
    public static Pixmap pause;
    public static Pixmap gameOver;
    public static Pixmap headUp;
    public static Pixmap headLeft;
    public static Pixmap headDown;
    public static Pixmap headRight;
    public static Pixmap tail;
    public static Pixmap stain1;
    public static Pixmap stain2;
    public static Pixmap stain3;

    public static Sound click;
    public static Sound eat;
    public static Sound bitten;
}
```

У нас есть статический член для каждого изображения и звука, которые мы загрузили из ресурсов. Если мы захотим использовать один из этих ресурсов, то можем поступить так:

```
game.getGraphics().drawPixmap(Assets.background, 0, 0)
```

или так:

```
Assets.click.play(1);
```

Теперь хранение стало значительно удобнее. Тем не менее обратите внимание, что ничто не мешает нам перезаписать эти статические члены, поскольку они не являются финальными. Но поскольку мы не перезаписываем их, мы можем быть спокойны. Эти общедоступные нефинальные члены представляют собой паттерн проектирования, вернее, антипаттерн. Однако в нашей игре можно немного поле-

ниться. Правильнее было бы скрыть эти ресурсы за методами-получателями и методами-установщиками в так называемом *классе-синглтоне*. Однако мы будем использовать наш простенький диспетчер ресурсов.

Настройки: сохранение пользовательских настроек и таблицы рекордов

Есть еще две вещи, которые нам необходимо отобразить на экране загрузки: пользовательские настройки и таблицу рекордов. Если вы снова посмотрите на экраны главного меню и таблицы рекордов в главе 3, увидите, что пользователь может регулировать звук, а также то, что мы сохраняем пять лучших рекордов. Мы сохраним данные настройки на карту памяти для того, чтобы их можно было заново загрузить, когда игра будет запущена в следующий раз. Для этого реализуем еще один простой класс — `Settings` (листинг 6.3).

Листинг 6.3. `Settings.java`: хранит, загружает и сохраняет наши настройки

```
package com.badlogic.androidgames.mrnom;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

import com.badlogic.androidgames.framework.FileIO;

public class Settings {
    public static boolean soundEnabled = true;
    public static int[] highscores = new int[] { 100, 80, 50, 30, 10 };
```

Проигрываются ли звуковые эффекты, зависит от общедоступного статического булева значения `soundEnabled`. Таблица рекордов хранится в целочисленном массиве из пяти элементов, отсортированных в порядке убывания. Мы задаем по умолчанию для обеих настроек только такие данные, которые целесообразно так задавать. Можно получить доступ к этим двум членам так же, как к членам класса `Assets`.

```
public static void load(FileIO files) {
    BufferedReader in = null;
    try {
        in = new BufferedReader(new InputStreamReader(
                    files.readFile(".mrnom")));
        soundEnabled = Boolean.parseBoolean(in.readLine());
        for (int i = 0; i < 5; i++) {
                highscores[i] = Integer.parseInt(in.readLine());
        }
    } catch (IOException e) {
        // Хорошо. Стандартные значения у нас есть.
```

```
    } catch (NumberFormatException e) {
        // Нет, ну как же прекрасны значения по умолчанию.
    } finally {
        try {
            if (in != null)
                in.close();
        } catch (IOException e) {
        }
    }
}
```

Статический метод `load()` пытается загрузить настройки из файла `.mrnom` из внешнего хранилища. Для этого ему необходим экземпляр класса `FileIO`, который мы передаем методу. Он предполагает, что настройки звука и каждая запись рекорда хранятся в отдельной строке, и просто читает их. Если что-то идет не так (например, хранилище недоступно или файл настроек еще не готов), мы просто оставляем значения по умолчанию и игнорируем ошибку.

```
public static void save(FileIO files) {
    BufferedWriter out = null;
    try {
        out = new BufferedWriter(new OutputStreamWriter(
                files.writeFile(".mrnom")));
        out.write(Boolean.toString(soundEnabled));
        for (int i = 0; i < 5; i++) {
            out.write(Integer.toString(highscores[i]));
        }

    } catch (IOException e) {
    } finally {
        try {
            if (out != null)
                out.close();
        } catch (IOException e) {
        }
    }
}
```

Следующий метод называется `save()`. Он берет текущие настройки и сериализует их в файл `.mrnom` на карте памяти (например, `/sdcard/.mrnom`). Настройки звука и каждая запись рекордов хранятся как отдельная строка в этом файле, как это и предполагается методом `load()`. Если что-то пойдет не так, мы просто игнорируем ошибку и используем данные по умолчанию, указанные ранее. Возможно, вы захотите проинформировать пользователя об ошибке загрузки.

```
    public static void addScore(int score) {
        for (int i = 0; i < 5; i++) {
            if (highscores[i] < score) {
                for (int j = 4; j > i; j--)
                    highscores[j] = highscores[j - 1];
```

```
                highscores[i] = score;
                break;
            }
        }
    }
}
```

Последний метод, `addScore()`, — вспомогательный. Мы будем использовать его для того, чтобы добавить новый рекорд в таблицу рекордов, автоматически сортируя их.

LoadingScreen: получение ресурсов с накопителя

С помощью этих классов мы теперь сможем с легкостью реализовать загрузочный экран. В листинге 6.4 показан необходимый для этого код.

Листинг 6.4. `LoadingScreen.java`, который загружает все объекты и настройки

```
package com.badlogic.androidgames.mrnom;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Graphics;
import com.badlogic.androidgames.framework.Screen;
import com.badlogic.androidgames.framework.Graphics.PixmapFormat;

public class LoadingScreen extends Screen {
    public LoadingScreen(Game game) {
        super(game);
    }
```

Мы наследуем класс `LoadingScreen` от класса `Screen`, который мы описали в главе 3. Для этого необходимо реализовать конструктор, принимающий параметр с типом `Game`, который мы передаем конструктору родительского класса. Обратите внимание, что этот конструктор будет вызываться в `MrNomGame.getStartScreen()`, который мы описали ранее.

```
@Override
public void update(float deltaTime) {
    Graphics g = game.getGraphics();
    Assets.background = g.newPixmap("background.png", PixmapFormat.RGB565);
    Assets.logo = g.newPixmap("logo.png", PixmapFormat.ARGB4444);
    Assets.mainMenu = g.newPixmap("mainmenu.png", PixmapFormat.ARGB4444);
    Assets.buttons = g.newPixmap("buttons.png", PixmapFormat.ARGB4444);
    Assets.help1 = g.newPixmap("help1.png", PixmapFormat.ARGB4444);
    Assets.help2 = g.newPixmap("help2.png", PixmapFormat.ARGB4444);
    Assets.help3 = g.newPixmap("help3.png", PixmapFormat.ARGB4444);
    Assets.numbers = g.newPixmap("numbers.png", PixmapFormat.ARGB4444);
    Assets.ready = g.newPixmap("ready.png", PixmapFormat.ARGB4444);
    Assets.pause = g.newPixmap("pausemenu.png", PixmapFormat.ARGB4444);
    Assets.gameOver = g.newPixmap("gameover.png", PixmapFormat.ARGB4444);
```

```
        Assets.headUp = g.newPixmap("headup.png", PixmapFormat.ARGB4444);
        Assets.headLeft = g.newPixmap("headleft.png", PixmapFormat.ARGB4444);
        Assets.headDown = g.newPixmap("headdown.png", PixmapFormat.ARGB4444);
        Assets.headRight = g.newPixmap("headright.png", PixmapFormat.ARGB4444);
        Assets.tail = g.newPixmap("tail.png", PixmapFormat.ARGB4444);
        Assets.stain1 = g.newPixmap("stain1.png", PixmapFormat.ARGB4444);
        Assets.stain2 = g.newPixmap("stain2.png", PixmapFormat.ARGB4444);
        Assets.stain3 = g.newPixmap("stain3.png", PixmapFormat.ARGB4444);
        Assets.click = game.getAudio().newSound("click.ogg");
        Assets.eat = game.getAudio().newSound("eat.ogg");
        Assets.bitten = game.getAudio().newSound("bitten.ogg");
        Settings.load(game.getFileIO());
        game.setScreen(new MainMenuScreen(game));
    }
```

Далее следует реализация метода `update()`, который загружает объекты и настройки. Для объектов-изображений мы просто создаем новые `Pixmap` с помощью метода `Graphics.newPixmap()`. Обратите внимание, что мы уточняем, какой формат цвета должен иметь `Pixmap`. Фон имеет формат RGB565, а все остальные изображения — ARGB4444. Мы это делаем для ускорения визуализации. Наши исходные изображения хранятся в форматах PNG RGB888 и ARGB8888. Мы также загружаем три звуковых эффекта и сохраняем их в соответствующих членах класса `Assets`. Далее загружаем настройки из внешнего хранилища с помощью метода `Settings.load()`. Наконец, запускаем переход экрана к объекту `Screen` под названием `MainMenuScreen`, который с этого момента примет на себя выполнение команд.

```
    @Override
    public void present(float deltaTime) {

    }

    @Override
    public void pause() {

    }

    @Override
    public void resume() {

    }

    @Override
    public void dispose() {

    }
}
```

Остальные методы — просто заглушки, они ничего не делают. Поскольку метод `update()` немедленно запустит переход экрана после того, как все объекты загружены, на этом экране нам больше ничего делать не надо.

Главное меню

Экран главного меню достаточно прост. Он отображает логотип, команды главного меню, а также представляет настройки звука в виде кнопки-переключателя. Он реагирует на нажатие команд меню или кнопки регулировки звука. Чтобы реализовать это поведение, нам необходимо знать две вещи: где на экране отображаются изображения и каковы области касаний, которые приведут либо к смене экрана, либо к изменению звуковых настроек. Рисунок 6.2 показывает, как и где мы отображаем различные изображения на экране. Из этого мы сможем точно определить зоны касаний.

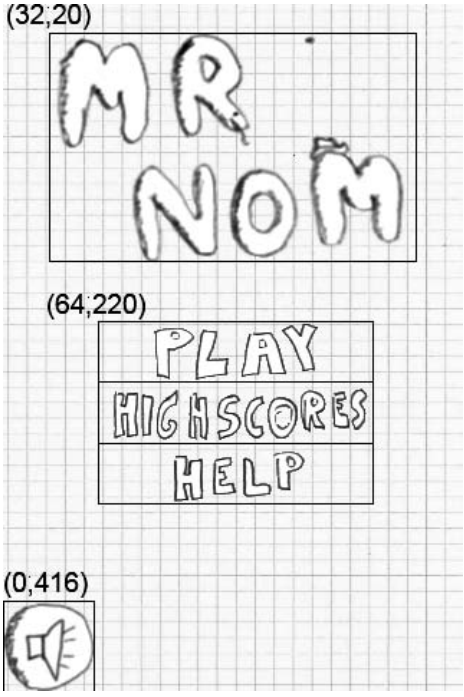

Рис. 6.2. Главное меню; координаты показывают, как и где мы визуализируем различные изображения, и выделяют зоны касания

X-координаты логотипа и команды главного меню вычислены таким образом, что их центры лежат на оси *x*.

Теперь реализуем класс `Screen` (листинг 6.5).

Листинг 6.5. `MainMenuScreen.java`, главное меню

```
package com.badlogic.androidgames.mrnom;

import java.util.List;

import com.badlogic.androidgames.framework.Game;
```

```
import com.badlogic.androidgames.framework.Graphics;
import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.Screen;

public class MainMenuScreen extends Screen {
    public MainMenuScreen(Game game) {
        super(game);
    }
```

Снова наследуем от Screen и реализуем соответствующий конструктор для него.

```
@Override
public void update(float deltaTime) {
    Graphics g = game.getGraphics();
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    game.getInput().getKeyEvents();

    int len = touchEvents.size();

    for(int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);
        if(event.type == TouchEvent.TOUCH_UP) {
            if(inBounds(event, 0, g.getHeight() - 64, 64, 64)) {
                Settings.soundEnabled = !Settings.soundEnabled;
                if(Settings.soundEnabled)
                        Assets.click.play(1);
                }
                if(inBounds(event, 64, 220, 192, 42) ) {
                    game.setScreen(new GameScreen(game));
                    if(Settings.soundEnabled)
                        Assets.click.play(1);
                    return;
                }
                if(inBounds(event, 64, 220 + 42, 192, 42) ) {
                    game.setScreen(new HighscoreScreen(game));
                    if(Settings.soundEnabled)
                        Assets.click.play(1);
                    return;
                }
                if(inBounds(event, 64, 220 + 84, 192, 42) ) {
                    game.setScreen(new HelpScreen(game));
                    if(Settings.soundEnabled)
                        Assets.click.play(1);
                    return;
                }
            }
        }
    }
```

Далее идет метод update(), в котором выполняем проверку событий касания. Сначала получаем TouchEvent и KeyEvent из экземпляра класса Input, предоставленного нам Game. Обратите внимание, что мы не используем KeyEvent, но в любом случае запрашиваем их для того, чтобы очистить внутренний буфер (да, это не очень удобно, но что делать). Далее проходим в цикле через все TouchEvent, пока не найдем то событие, которое имеет тип TouchEvent.TOUCH_UP. Мы могли бы также искать события TouchEvent.TOUCH_DOWN, но в большинстве пользовательских интерфейсов событие отпускания используется для определения того, что определенный элемент пользовательского интерфейса был нажат.

Как только мы нашли подходящее событие, проверяем, относится ли оно ко звуку или к командам меню. Чтобы подчистить код, я написал метод inBounds(), который принимает событие касания, x- и y-координаты, а также высоту и ширину. Этот метод определяет, произошло ли касание внутри прямоугольника экрана с заданными параметрами и возвращает true или false.

Если нажата кнопка-переключатель для работы со звуком, просто инвертируем булево значение Settings.soundEnabled. Если нажата какая-либо из команд главного меню, переходим к соответствующему экрану, создавая его и запуская его с помощью Game.setScreen(). Наш метод может сразу же вернуть управление, если MainMenuScreen больше нечего делать. Мы также проигрываем звуки нажатия, если нажата кнопка-переключатель звука или команда главного меню, а звук включен.

Обратите внимание, что все события касания будут сообщаться в соответствии с нашим конечным разрешением — 320×480 пикселов.Так получается благодаря магии масштабирования, которую мы обсуждали в главе 5:

```
private boolean inBounds(TouchEvent event, int x, int y, int width,
                         int height) {
        if(event.x > x && event.x < x + width - 1 &&
            event.y > y && event.y < y + height - 1)
                return true;
        else
            return false;
    }
```

Метод inBounds() работает так же, как и методы, рассмотренные ранее: принимает TouchEvent и прямоугольник и сообщает вам, находятся ли координаты события касания внутри прямоугольника.

```
@Override
    public void present(float deltaTime) {
        Graphics g = game.getGraphics();

        g.drawPixmap(Assets.background, 0, 0);
        g.drawPixmap(Assets.logo, 32, 20);
        g.drawPixmap(Assets.mainMenu, 64, 220);
        if(Settings.soundEnabled)
```

```
            g.drawPixmap(Assets.buttons, 0, 416, 0, 0, 64, 64);
        else
            g.drawPixmap(Assets.buttons, 0, 416, 64, 0, 64, 64);
    }
```

Метод `present()` покажется, наверное, самым долгожданным, но, боюсь, он не такой уж впечатляющий. Наш небольшой фреймворк игры действительно упрощает отображение главного меню. Мы просто отрисовываем фон на (0; 0), фактически стирая наш фреймбуфер, так что никакого вызова `Graphics.clear()` не требуется. Далее рисуем логотип и команды главного меню согласно координатам на рис. 6.2. Метод оканчивается рисованием кнопки звука в зависимости от текущих настроек. Как видите, мы используем тот же `Pixmap`, но просто рисуем соответствующую часть его (кнопка переключения звука, см. рис. 6.1). Это было несложно.

```
@Override
    public void pause() {
        Settings.save(game.getFileIO());
    }
```

Последний элемент, который нам нужно обсудить, — метод `pause()`. Поскольку мы изменяем одну из настроек на этом экране, необходимо удостовериться, что она сохранится во внешнем хранилище. Это совсем несложно сделать с помощью класса `Settings`.

```
    @Override
    public void resume() {

    }

    @Override
    public void dispose() {

    }
}
```

Методы `resume()` и `dispose()` на этом экране ничего не делают.

Класс(ы) HelpScreen

Теперь реализуем классы `HelpScreen`, `HighscoreScreen` и `GameScreen`, которые мы уже использовали в методе `update()`.

Мы описали три вспомогательных экрана в главе 3, каждый из них более или менее подробно объясняет один из аспектов игры. Теперь просто переведем эту информацию в реализации `Screen`, которые называются `HelpScreen`, `HelpScreen2` и `HelpScreen3`. Все они оснащены одной кнопкой, которая сменяет экран. Экран `HelpScreen3` приводит обратно к `MainMenuScreen`. На рис. 6.3 показаны три вспомогательных экрана, координаты и зоны касаний.

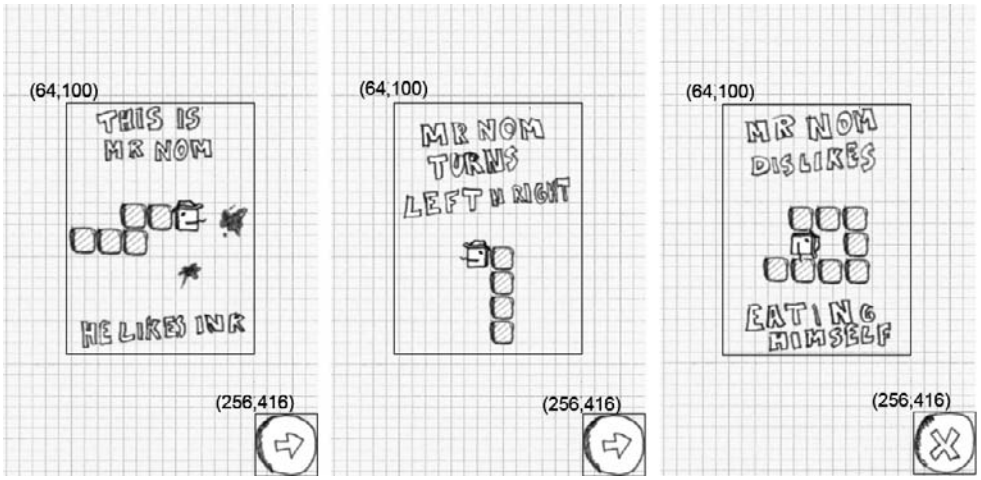

Рис. 6.3. Три вспомогательных экрана с координатами и зонами касаний

Теперь все кажется достаточно простым для реализации. Начнем с класса `HelpScreen`, показанного в листинге 6.6.

Листинг 6.6. `HelpScreen.java`, первый экран помощи

```
package com.badlogic.androidgames.mrnom;

import java.util.List;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Graphics;
import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.Screen;

public class HelpScreen extends Screen {
    public HelpScreen(Game game) {
        super(game);
    }

    @Override
    public void update(float deltaTime) {
        List<TouchEvent> touchEvents = game.getInput().getTouchEvents();

        game.getInput().getKeyEvents();

        int len = touchEvents.size();
        for(int i = 0; i < len; i++) {
            TouchEvent event = touchEvents.get(i);
            if(event.type == TouchEvent.TOUCH_UP) {
                if(event.x > 256 && event.y > 416 ) {
                    game.setScreen(new HelpScreen2(game));
                    if(Settings.soundEnabled)
```

```
                    Assets.click.play(1);
                return;
            }
        }
    }
}

@Override
public void present(float deltaTime) {
    Graphics g = game.getGraphics();
    g.drawPixmap(Assets.background, 0, 0);
    g.drawPixmap(Assets.help1, 64, 100);
    g.drawPixmap(Assets.buttons, 256, 416, 0, 64, 64, 64);
}

@Override
public void pause() {

}

@Override
public void resume() {

}

@Override
public void dispose() {
}
}
```

Опять-таки все очень просто. Мы наследуем от класса `Screen` и реализуем конструктор. Далее следует уже знакомый нам метод `update()`, который просто проверяет, нажата ли кнопка внизу. Если она нажата, проигрываем звук нажатия и переходим к `HelpScreen2`.

Метод `present()` снова отображает фон со вспомогательным изображением и кнопкой.

Классы `HelpScreen2` и `HelpScreen3` выглядят почти одинаково, различия сводятся к тому, как отображается вспомогательное изображение, и экрану, к которому они переходят. Я думаю, что вы согласитесь, что нам не нужно смотреть на их код. Переходим к экрану рекордов.

Экран рекордов

Экран рекордов просто выводит пять лучших результатов, которые мы храним в классе `Settings`, плюс заголовок, сообщающий пользователю, что он находится на экране рекордов. Еще здесь есть кнопка в нижнем левом углу, которая при нажатии возвращает нас в главное меню. Самое интересное в этом — как визуализировать

таблицу рекордов. Для начала посмотрим на то, как мы визуализируем изображения, показанные на рис. 6.4.

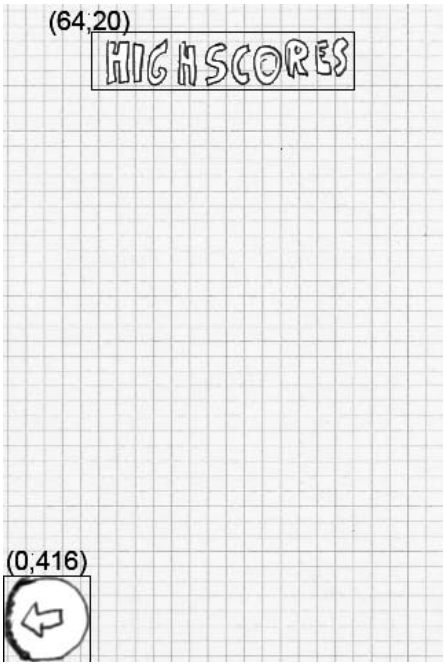

Рис. 6.4. Экран рекордов без рекордов

Он выглядит таким же простым, как и другие экраны, которые мы реализовали. Однако как нарисовать изменяющиеся рекорды?

Отрисовка чисел: краткое введение

У нас есть графический ресурс, называющийся `numbers.png` и содержащий все цифры от 0 до 9 плюс точка. Размер каждой цифры — 20×32 пиксела, а точки — 10×32 пиксела. Цифры размещены слева направо в возрастающем порядке. Экран рекордов должен показывать пять строк, каждая из которых отображает один из пяти лучших результатов. Каждая строка должна начинаться с позиции рекорда (например, «1» или «5»), затем идет пробел и сам результат. Как это сделать?

В нашем распоряжении есть две вещи: изображение `numbers.png` и `Graphics.drawPixmap()`, который позволяет нам нарисовать на экране часть изображения. Допустим, мы хотим, чтобы первая строка списка рекордов, задаваемая по умолчанию, например «1. 100» была отображена с координатами (20; 100) так, чтобы верхний левый угол цифры 1 совпадал с этими координатами.

Вызываем `Graphics.drawPixmap()` следующим образом:

```
game.getGraphics().drawPixmap(Assets.numbers, 20, 100, 20, 0, 20, 32);
```

Мы знаем, что цифра 1 имеет ширину 20 пикселов. Следующий символ нашей строки будет отображен с координатами (20 + 20; 100). В строке "1. 100," данным символом является точка, ширина которой в изображении numbers.png равна 10 пикселов:

```
game.getGraphics().drawPixmap(Assets.numbers, 40, 100, 200, 0, 10, 32);
```

Следующий символ в строке должен быть визуализирован с координатами (20 + 20 + 10; 100). Этот символ — пробел, который нам отрисовывать не нужно. Все, что необходимо сделать, — продвинуться по оси *x* на 20 пикселов, поскольку, как вы догадываетесь, это размеры пробела. Следующий символ 1 будет, соответственно, визуализирован с координатами (20 + 20 + 10 + 20; 100). Вы уже уловили закономерность?

Имея координаты верхнего левого угла нашего первого символа в строке, мы можем перебрать в цикле все символы строки, нарисовать символ и увеличить *x*-координату для следующего отрисовываемого символа на 20 или 10 пикселов в зависимости от символа, который мы только что нарисовали.

Нам также необходимо выяснить, какую часть изображения numbers.png нам необходимо отрисовать для того или иного символа. Для этого нам нужны *x*- и *y*-координаты верхнего левого угла этой части, а также ее ширина и высота. *Y*-координата всегда будет равна нулю, что очевидно, если посмотреть на рис. 6.1. Высота также является константой, в нашем случае это 32. Ширина равна или 20 пикселов (если символ — цифра), или 10 пикселов (если это точка). Единственная величина, которую нам необходимо вычислить, — это *x*-координата части изображения numbers.png. Для этого мы применим небольшую хитрость.

Символы в строке могут быть интерпретированы как Юникод или как 16-битные целые числа. Это значит, что мы фактически можем производить вычисления с символьными кодами. По счастливой случайности символы 0–9 представлены последовательными возрастающими целыми числами. Мы можем использовать этот факт для того, чтобы рассчитать значение по оси *x* на изображении number.png.

```
char character = string.charAt(index);
int x = (character – '0') * 20;
```

Таким образом, мы получим 0 для символа 0, 3 · 20 = 60 для символа 3 и т. д. Это точная координата части каждой цифры. Конечно, такой принцип неприменим к точке, так что с ней мы поступим иначе. Подведем итоги в методе, который может отображать одну из наших строк рекордов, при наличии конкретной строки, а также имея координаты *x* и *y*, в которых должно начинаться отображение.

```
public void drawText(Graphics g, String line, int x, int y) {
    int len = line.length();
    for (int i = 0; i < len; i++) {
        char character = line.charAt(i);

        if (character == ' ') {
            x += 20;
            continue;
```

```
        }

        int srcX = 0;
        int srcWidth = 0;
        if (character == '.') {
            srcX = 200;
            srcWidth = 10;
        } else {
            srcX = (character – '0') * 20;
            srcWidth = 20;
        }

        g.drawPixmap(Assets.numbers, x, y, srcX, 0, srcWidth, 32);
        x += srcWidth;
    }
}
```

Мы перебираем все символы строки. Если текущий символ — пробел, просто увеличиваем координату *x* на 20 пикселов. В противном случае мы рассчитываем *x*-координату и ширину конкретной области (участка) символа в изображении numbers.png. Символом в данном случае может быть либо точка, либо цифра. Далее рисуем текущий символ и увеличиваем положение по оси *x* на ширину символа, который мы только что нарисовали. Этот метод, естественно, не будет работать, если мы будем использовать что-либо кроме пробелов, цифр и точек. Как вы думаете, можно ли сделать так, чтобы он работал с любой строкой?

Реализация экрана

Взяв на вооружение новые знания, мы сможем без труда реализовать класс HighscoreScreen (листинг 6.7).

Листинг 6.7. HighscoreScreen.java, наши труды дают плоды

```
package com.badlogic.androidgames.mrnom;

import java.util.List;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Graphics;
import com.badlogic.androidgames.framework.Screen;
import com.badlogic.androidgames.framework.Input.TouchEvent;

public class HighscoreScreen extends Screen {
    String lines[] = new String[5];

    public HighscoreScreen(Game game) {
        super(game);

        for (int i = 0; i < 5; i++) {
            lines[i] = "" + (i + 1) + ". " + Settings.highscores[i];
        }
    }
```

Поскольку мы по-прежнему не собираемся ссориться со сборщиком мусора, мы сохраняем строки пяти лучших результатов в элементах массива. Инициализируем строки, опираясь на массив Settings.highscores в конструкторе.

```
@Override
public void update(float deltaTime) {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    game.getInput().getKeyEvents();

    int len = touchEvents.size();
    for (int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);
        if (event.type == TouchEvent.TOUCH_UP) {
            if (event.x < 64 && event.y > 416) {
                if(Settings.soundEnabled)
                    Assets.click.play(1);
                game.setScreen(new MainMenuScreen(game));
                return;
            }
        }
    }
}
```

Далее определяется метод update(), который, как обычно, довольно тривиален. Мы проверяем, произошло ли касание кнопки в нижнем левом углу. Если произошло, проигрываем звук нажатия и переходим обратно к MainMenuScreen.

```
@Override
public void present(float deltaTime) {
    Graphics g = game.getGraphics();

    g.drawPixmap(Assets.background, 0, 0);
    g.drawPixmap(Assets.mainMenu, 64, 20, 0, 42, 196, 42);

    int y = 100;
    for (int i = 0; i < 5; i++) {
        drawText(g, lines[i], 20, y);
        y += 50;
    }

    g.drawPixmap(Assets.buttons, 0, 416, 64, 64, 64, 64);
}
```

Метод present() достаточно прост при использовании в сочетании с методом drawText(), который мы недавно описали. Сначала мы, как обычно, визуализируем фоновое изображение вместе с частью HIGHSCORES (Рекорды) изображения Assets.mainmenu. Мы могли бы сохранить его в отдельном файле, но используем уже имеющийся ресурс, чтобы не занимать лишней памяти.

Далее проходим в цикле пять строк, установленных в конструкторе, — по одной на каждый рекорд. Рисуем каждую строку с помощью метода drawText(). Первая

строка начинается в координатах (20; 100), следующая — в координатах (20; 150) и т. д. Просто увеличиваем *y*-координату для визуализации текста на 50 пикселов для каждой линии, чтобы у нас были красивые вертикальные интервалы между строками.

Наконец, рисуем кнопку.

```
public void drawText(Graphics g, String line, int x, int y) {
        int len = line.length();
        for (int i = 0; i < len; i++) {
            char character = line.charAt(i);

            if (character == ' ') {
                x += 20;
                continue;
            }

            int srcX = 0;
            int srcWidth = 0;
            if (character == '.') {
                srcX = 200;
                srcWidth = 10;
            } else {
                srcX = (character - '0') * 20;
                srcWidth = 20;
            }

            g.drawPixmap(Assets.numbers, x, y, srcX, 0, srcWidth, 32);
            x += srcWidth;
        }
    }

    @Override
    public void pause() {

    }

    @Override
    public void resume() {

    }

    @Override
    public void dispose() {

    }
}
```

Оставшиеся методы понятны без объяснения. Перейдем к недостающему блоку игры «Мистер Ном» — игровому экрану.

Абстрагирование...

Мы наконец-то закончили реализацию самых скучных компонентов пользовательского интерфейса и написали часть кода для разной рутинной обработки. Теперь создадим абстракции для «Мистера Нома» и всех объектов в нем. Наш мистер Ном перестанет зависеть от разрешения экрана и сможет жить в своем небольшом мирке со своей небольшой системой координат.

Абстрагирование мира мистера Нома: модель, вид, контроллер

Если вы программируете уже давно, то, возможно, уже слышали о паттернах проектирования. Это своего рода стратегии, уместные при создании кода для данного сценария. Некоторые из них только теоретические, другие применяются на практике. Для создания игры мы можем позаимствовать некоторые идеи из паттерна проектирования «Модель — вид — контроллер» (Model-View-Controller, MVC). Он достаточно часто используется, например в базах данных, чтобы разделить модель данных от уровня представления и уровня управления данными. Мы не будем строго придерживаться этого паттерна проектирования, а приспособим его к нашим нуждам в упрощенной форме.

Что же это значит для мистера Нома? Первым делом нам необходимо отделить представление нашего мира от растровых изображений, звуков, фреймбуферов и событий ввода. Вместо этого смоделируем мир мистера Нома с несколькими простыми классами в объектно-ориентированном стиле. У нас будет класс для пятен в игровом мире и класс для самого мистера Нома. Персонаж состоит из головы и хвоста, которые мы также представим в отдельном классе. Чтобы связать все вместе, используем универсальный класс, представляющий весь мир мистера Нома, включающий пятна и самого мистера Нома. Все это будет *моделью* в составе MVC.

Вид (view) в MVC будет кодом, отвечающим за отображение мира мистера Нома. У нас будет класс или метод, который получает класс мира, читает его текущее состояние и визуализирует его на экране. Сам процесс визуализации не касается классов модели. Тем не менее это наиболее важный урок, который мы можем извлечь из MVC. Классы модели не зависят ни от чего, однако классы и методы вида зависят от классов модели.

В MVC нам также необходим *контроллер*. Он указывает классам модели изменить их состояние, основываясь на таких факторах, как пользовательский ввод чего-либо или ход времени. Классы модели предоставляют методы контроллеру (например, инструкции: «Поверните мистера Нома направо»), которые контроллер может позже использовать для того, чтобы модифицировать состояние модели. У нас в классах модели нет никакого кода, который имел бы прямой доступ к таким компонентам, как сенсорный экран или акселерометр. Таким образом, наши классы модели не имеют никаких внешних зависимостей.

Возможно, это звучит достаточно сложно и вы удивляетесь, почему мы действуем именно так. Тем не менее подобный подход имеет множество преимуществ. Мы можем реализовать всю логику игры без учета свойств графики, аудио или устройств ввода. Мы можем модифицировать визуализацию мира игры, и нам не придется при этом изменять сами классы модели. Мы можем даже перенести мир из 2D в 3D, а также легко поддерживать новые устройства ввода, используя контроллер. Контроллер всего лишь преобразует события ввода в вызовы методов, относящихся к классам модели. Хотите поворачивать мистера Нома с помощью акселерометра? Нет проблем — прочтите значения акселерометра в контроллере и преобразуйте их в вызов метода «Повернуть мистера Нома налево» или «Повернуть мистера Нома направо» в модели «Мистер Ном». Хотите добавить поддержку устройства Zeemote? Проделайте те же самые действия, что и в случае с акселерометром. При использовании контроллеров особенно удобно то, что мы не изменим ни единой строки кода «Мистера Нома» для того, чтобы все это получилось.

Начнем описание мира мистера Нома. Чтобы это сделать, немного отклонимся от строгого паттерна MVC и используем наши графические ресурсы для иллюстрации основных идей. Это также поможет нам позже реализовать вид — второй компонент рассматриваемой модели (абстрактный мир мистера Нома будет выражен в пикселах).

На рис. 6.5 показан экран игры с наложенным на него в виде клеточек миром мистера Нома.

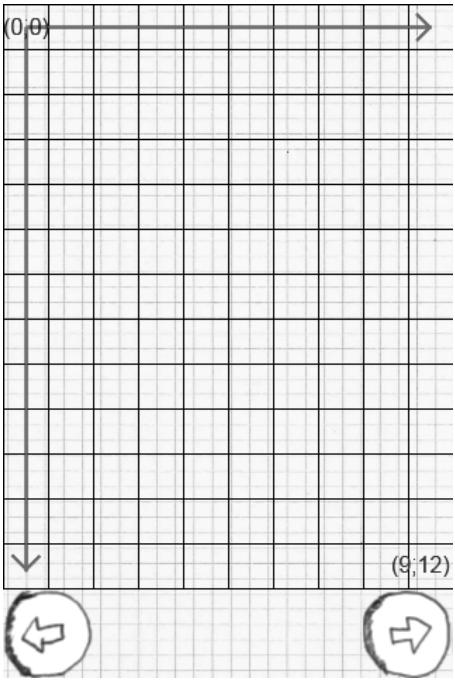

Рис. 6.5. Мир мистера Нома, наложенный на игровой экран

Обратите внимание, что мир мистера Нома состоит из 10×13 клеток. Мы адресуем клетки в координатной системе, начиная с верхнего левого угла в точке (0; 0) и передвигаясь к правому нижнему углу до (9; 12). Любая часть мистера Нома должна быть в одной из этих клеток, а также иметь целочисленные значения координат x и y внутри игрового мира. Это правило действует и для пятен в мире. Каждая часть мистера Нома занимает ровно одну клетку — 1 единицу. Запомните, что тип единицы неважен — это придуманный нами мир, который не зависит ни от системы СИ, ни от пикселов.

Мистер Ном не может путешествовать за границами этого небольшого мира. Если он дойдет до края, он просто появится с другой стороны, а за ним и все его части. (Кстати, на Земле у нас с вами все точно так же, просто идите в какую-то сторону достаточно долго, и вы вернетесь в место, откуда ушли). Мистер Ном может также передвигаться только клетка за клеткой. Все его части будут иметь целочисленные координаты. Например, он никогда не сможет занять две с половиной клетки.

ПРИМЕЧАНИЕ

Как было указано выше, здесь мы работаем с нестрогим паттерном MVC. Если вы хотите подробнее изучить классический вариант данного паттерна, можете почитать книгу Э. Гамма, Р. Хелм «Приемы объектно-ориентированного проектирования. Паттерны проектирования». — СПб.: Питер, 2012. В этой книге паттерн проектирования MVC именуется Observer (Наблюдатель).

Класс Stain

Самый простой объект в мире мистера Нома — это пятно. Оно просто находится в клетке мира, ожидая, пока его съедят. Когда мы проектировали мистера Нома, мы создали три различных на вид пятна. Тип пятна не имеет никакого значения в мире мистера Нома, однако мы все равно включим этот показатель в наш класс `Stain`. В листинге 6.8 показан класс `Stain`.

Листинг 6.8. `Stain.java`

```
package com.badlogic.androidgames.mrnom;

public class Stain {
    public static final int TYPE_1 = 0;
    public static final int TYPE_2 = 1;
    public static final int TYPE_3 = 2;
    public int x, y;
    public int type;

    public Stain(int x, int y, int type) {
        this.x = x;
        this.y = y;
        this.type = type;
    }
}
```

Класс Stain определяет три общедоступные статические константы, которые кодируют тип пятна. Каждый объект Stain имеет три члена, x- и y-координаты в мире мистера Нома и тип, определенный одной из констант. Чтобы упростить код, мы обойдемся без методов-установщиков и методов-получателей, хотя обычно работаем с ними. Мы заканчиваем класс небольшим конструктором, который позволяет нам легко инстанцировать экземпляр Stain.

Единственный момент, на который стоит обратить внимание, — отсутствие какой-либо связи с графикой, звуком и другими классами. Класс Stain обособлен и кодирует атрибуты пятна в мире мистера Нома.

Классы Snake и SnakePart

Мистер Ном — это двигающаяся цепочка, состоящая из соединенных частей, которые двигаются друг за другом, когда мы выбираем одну часть и перетаскиваем ее. Каждая часть занимает одну клетку в мире мистера Нома, как и пятно. В нашей модели нет различия между головой и хвостом, так что мы можем использовать один класс для представления обеих этих частей мистера Нома.

В листинге 6.9 показан класс SnakePart, который используется для того, чтобы описать обе части мистера Нома.

Листинг 6.9. SnakePart.java

```
package com.badlogic.androidgames.mrnom;

public class SnakePart {
    public int x, y;

    public SnakePart(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Этот класс практически ничем не отличается от класса Stain, мы просто убрали член type. Первый по-настоящему интересный класс в нашей модели мира мистера Нома — Snake. Посмотрим, что он должен уметь делать. Итак, здесь мы будем:

- хранить хвостовую и головную части;
- узнавать, в какую сторону мистер Ном двигается в данный момент;
- добавлять новый кусок хвоста, когда мистер Ном съедает пятно;
- передвигаться на одну клетку в выбранном направлении.

Выполнить первый и второй пункты достаточно просто. Мы просто используем список экземпляров класса SnakePart. Первая часть в нем — голова, а остальные — хвост. Мистер Ном может двигаться вверх, вниз, влево и вправо. Мы можем запрограммировать это с помощью нескольких констант и сохранить текущее направление мистера Нома в члене класса Snake.

Выполнить третий пункт также несложно. Мы просто добавляем еще один экземпляр класса `SnakePart` в список частей, которые у нас уже есть. Вопрос состоит в том, в каком месте следует добавлять эту часть. Возможно, вы будете удивлены, но новая часть получает такую же позицию, что и последняя часть в списке. Причина этого проясняется, когда мы смотрим на то, как реализуется последний элемент в предыдущем списке — движения мистера Нома.

На рис. 6.6 показан мистер Ном в его начальном виде. Он состоит из трех частей: головы в точке (5; 6) и двух частей хвоста в (5; 7) и (5; 8).

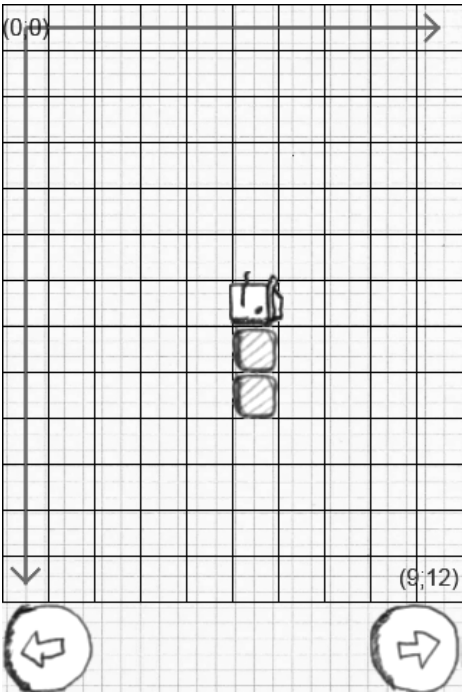

Рис. 6.6. Мистер Ном в начальном виде

Части в списке выстроены по порядку, начиная с головы и заканчивая последней частью хвоста. Когда мистер Ном передвигается на одну клетку, все его части, находящиеся за головой, следуют за ней. Однако части мистера Нома не всегда составляют прямую линию, как на рис. 6.6, поэтому простого передвижения всех частей в направлении, в котором двигается мистер Ном, недостаточно. Нам предстоит сделать кое-что посложнее.

Необходимо начать с последней части в списке, несмотря на то, что это кажется нелогичным. Мы передвигаем ее туда, где находилась часть перед ней, и повторяем это для всех других частей в списке за исключением головы, поскольку перед ней нет никакой части. В случае с головой мы смотрим, в каком направлении мистер Ном двигается в текущий момент, и изменяем позицию головы в соответствии

с этим. Рисунок 6.7 демонстрирует это с немного более сложной конфигурацией мистера Нома.

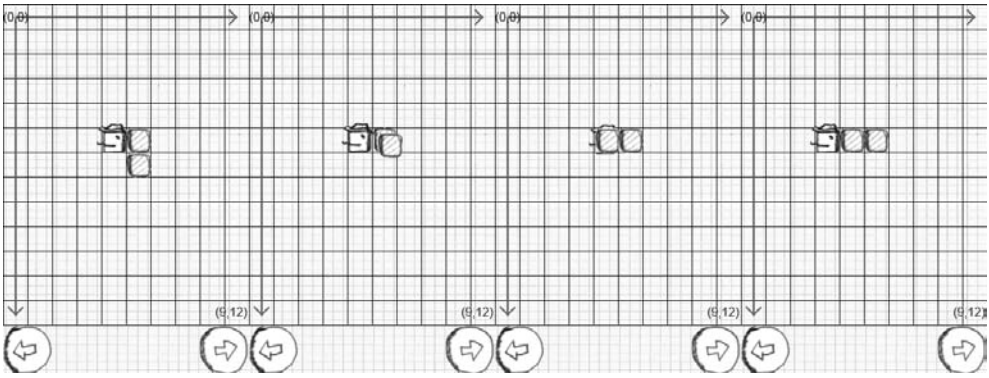

Рис. 6.7. Мистер Ном движется, и хвост перемещается за ним

Эта стратегия передвижения хорошо сочетается со стратегией поедания. Когда мы добавляем новую часть мистера Нома, то до того, как он сделает движение, новая часть будет стоять там же, где стоит часть перед ней. Обратите также внимание, что это позволит нам легко реализовать перемещение мистера Нома на другую сторону мира, когда он пересекает край экрана. Мы просто правильно устанавливаем позицию головы, остальное делается автоматически.

Теперь, когда у нас есть вся эта информация, мы можем реализовать класс `Snake`, представляющий собой мистера Нома (листинг 6.10).

Листинг 6.10. `Snake.java`: мистер Ном в коде

```
package com.badlogic.androidgames.mrnom;

import java.util.ArrayList;
import java.util.List;

public class Snake {
    public static final int UP = 0;
    public static final int LEFT = 1;
    public static final int DOWN = 2;
    public static final int RIGHT = 3;

    public List<SnakePart> parts = new ArrayList<SnakePart>();
    public int direction;
```

Мы начинаем с определения нескольких констант, задающих направление движения мистера Нома. Запомните, что он может повернуть только направо или налево, так что очень важно, как мы определим значения константы. Позже это позволит нам с легкостью поворачивать на 90°, просто увеличивая или уменьшая текущее значение направления на единицу.

Далее определяем список parts, который содержит все части мистера Нома. Первый элемент в этом списке — голова, остальные элементы — части хвоста. Второй член класса Snake содержит направление, в котором в данный момент двигается мистер Ном.

```
public Snake() {
        direction = UP;
        parts.add(new SnakePart(5, 6));
        parts.add(new SnakePart(5, 7));
        parts.add(new SnakePart(5, 8));
    }
```

В конструкторе мы задаем, что мистер Ном состоит из головы и двух частей хвоста, расположенных в центре мира, так, как это было показано на рис. 6.6. Мы также задаем направление как Snake.UP, чтобы мистер Ном перемещался на одну клетку вперед, когда ему скомандуют двигаться в следующий раз.

```
public void turnLeft() {
        direction += 1;
        if(direction > RIGHT)
            direction = UP;
    }

    public void turnRight() {
    direction -= 1;
    if(direction < UP)
        direction = RIGHT;
    }
```

Методы turnLeft() и turnRight() просто изменяют элемент направления в классе Snake. Чтобы повернуть налево, мы увеличиваем его на единицу, а чтобы повернуть направо, уменьшаем на единицу. Нам также необходимо убедиться, что мы правильно выполним поворот, если значение направления превысит диапазон констант, который мы определили ранее.

```
public void eat() {
        SnakePart end = parts.get(parts.size()-1);
        parts.add(new SnakePart(end.x, end.y));
    }
```

Далее рассмотрим метод eat(). Он добавляет новый элемент SnakePart к концу списка, эта новая часть будет иметь такую же позицию, как и текущая конечная часть. В следующий раз, когда мистер Ном переместится на одну клетку, эти перекрывающие друг друга части разъединятся, как мы и обсуждали это ранее.

```
public void advance() {
        SnakePart head = parts.get(0);

        int len = parts.size() - 1;
        for(int i = len; i > 0; i--) {
```

```
            SnakePart before = parts.get(i-1);
            SnakePart part = parts.get(i);
            part.x = before.x;
            part.y = before.y;
        }

        if(direction == UP)
            head.y -= 1;
        if(direction == LEFT)
            head.x -= 1;
        if(direction == DOWN)
            head.y += 1;
        if(direction == RIGHT)
            head.x += 1;

        if(head.x < 0)
            head.x = 9;
        if(head.x > 9)
            head.x = 0;
        if(head.y < 0)
            head.y = 12;
        if(head.y > 12)
            head.y = 0;
    }
```

Следующий метод `advance()` реализует логику, показанную на рис. 6.7. Сначала мы передвигаем каждую часть на позицию идущей перед ней части, начиная с последней. Мы исключаем голову из этой операции. Далее перемещаем голову в соответствии с текущим направлением передвижения мистера Нома. Наконец проверяем, не вышел ли мистер Ном за границы мира. Если вышел, перебрасываем его на противоположную часть экрана.

```
public boolean checkBitten() {
        int len = parts.size();
        SnakePart head = parts.get(0);
        for(int i = 1; i < len; i++) {
            SnakePart part = parts.get(i);
            if(part.x == head.x && part.y == head.y)
                return true;
        }
        return false;
    }
}
```

Последний метод `checkBitten()` — небольшой вспомогательный код, который проверяет, не укусил ли мистер Ном свой хвост. Он проверяет, чтобы части хвоста не находились в той же позиции, что и голова. Если же это происходит, мистер Ном умирает и игра заканчивается.

Класс World

Последний класс нашей модели называется `World`. Он должен выполнять следующие задачи:

- отслеживание мистера Нома (в виде экземпляра класса `Snake`), а также пятен, разбросанных по миру. В мире всегда должно быть не менее одного пятна;
- предоставление метода, который обновит мистера Нома по хронологическому принципу (например, он должен передвигаться на одну клетку каждые 0,5 секунды). Этот метод также проверяет, съел ли мистер Ном пятно или укусил себя;
- отслеживание счета, что фактически является подсчетом количества пятен, съеденных на данный момент, умноженных на 10;
- увеличение скорости мистера Нома после каждых 10 пятен, которые он съел. Это немного усложнит игру;
- отслеживание, жив ли мистер Ном до сих пор. Это понадобится нам позже для окончания игры;
- создание нового пятна после того, как мистер Ном съел текущее (небольшое, но важное и удивительно сложное задание).

В этом списке есть только два пункта, которые мы до сих пор не обсудили: обновление мира по хронологическому принципу и размещение нового пятна.

Перемещение мистера Нома по хронологическому принципу

В главе 3 мы говорили о хронологическом принципе перемещения. Это фактически значит, что мы задаем скорость всех объектов нашей игры, измеряем время, которое прошло с момента последнего обновления (дельта времени), и передвигаем объекты, умножая их скорость на дельту времени. В примере из главы 3 для того, чтобы этого достигнуть, мы использовали значения с плавающей точкой. Части мистера Нома имеют целочисленные значения позиций, поэтому нам предстоит выяснить, как передвигать объекты в этом случае.

Для начала определим скорость мистера Нома. В мире мистера Нома существует время, измеряемое в секундах. Изначально мистер Ном должен продвигаться со скоростью одна клетка в 0,5 секунды. Все, что нам нужно, — следить за тем, сколько времени прошло с того момента, как мы подвинули мистера Нома в последний раз. Если прошедшее время превышает 0,5 секунды, мы вызываем метод `Snake.advance()` и сбрасываем счетчик времени. Откуда мы получаем эти дельты времени? Помните метод `Screen update ()`? Он получает дельту времени для каждого кадра. Мы просто передаем эту информацию методу `update` нашего класса `World`, который ведет подсчеты. Чтобы сделать игру немного сложнее, мы уменьшаем порог на 0,05 секунды каждый раз, когда мистер Ном съедает 10 пя-

тен. Конечно, мы должны следить за тем, чтобы порог не достиг 0, иначе мистер Ном будет передвигаться со скоростью света. Эйнштейну это бы не понравилось.

Размещение пятен

Вторая проблема, которую нам предстоит решить, — как разместить новое пятно, когда мистер Ном съел текущее. Оно должно появиться в случайной клетке мира. Так почему бы нам не инстанцировать новое пятно в случайной позиции? К сожалению, все не так просто.

Представьте, что мистер Ном занимает значительное количество клеток. Вероятность того, что пятно появится в клетке, которую уже занимает мистер Ном, возрастает с увеличением размеров самого мистера Нома. Нам необходимо найти клетку, которая в данный момент не занята мистером Номом. Звучит несложно, не правда ли? Просто проверить все клетки и поместить пятно в первую, которая не занята мистером Номом.

Однако снова это не очень правильно. Если мы каждый раз будем начинать поиск с одной и той же позиции, пятно не будет располагаться случайно. Вместо этого мы каждый раз будем начинать поиск в случайной позиции мира, сканировать все клетки до конца мира, а затем начинать сначала, если в этой области свободных клеток не оказалось.

Как нам проверить, свободна ли клетка? Самым простым решением будет проверить все клетки, взять x- и y-координаты каждой и сравнить все части мистера Нома с этими координатами. У нас есть $10 \cdot 13 = 130$ клеток, и мистер Ном может занимать до 55 клеток. Это будет $130 \cdot 55 = 7\ 150$ проверок. Конечно, большинство устройств сможет справиться с таким количеством операций, но мы можем сделать гораздо лучше.

Создадим двухмерный массив булевых значений, каждый элемент массива будет представлять одну клетку мира. Когда нам нужно поместить новое пятно, мы сначала проходим через все части мистера Нома и отмечаем в массиве занятые элементы как `true`. Когда мы затем просто выбираем случайную позицию, с которой начинаем сканировать, находим свободную клетку, в которую можем поместить новое пятно. В случае, когда мистер Ном состоит из 55 частей, это займет $130 + 55 = 185$ проверок. Вот так гораздо лучше.

Определение окончания игры

Осталось еще одна вещь, о которой нам необходимо позаботиться: что делать, если все клетки заняты мистером Номом? В этом случае игра будет закончена, поскольку мистер Ном станет всем миром. Принимая во внимание, что мы добавляем 10 очков каждый раз, когда мистер Ном съедает пятно, максимально возможный рекорд $10 \cdot (13 - 3) \cdot 10 = 1000$ очков (помните, мистер Ном начинает с трех частей).

Реализация класса World

Нам с вами предстоит реализовать еще немало вещей, так что начнем. В листинге 6.11 показан код класса World.

Листинг 6.11. World.java

```
package com.badlogic.androidgames.mrnom;

import java.util.Random;

public class World {
    static final int WORLD_WIDTH = 10;
    static final int WORLD_HEIGHT = 13;
    static final int SCORE_INCREMENT = 10;
    static final float TICK_INITIAL = 0.5f;
    static final float TICK_DECREMENT = 0.05f;

    public Snake snake;
    public Stain stain;
    public boolean gameOver = false;;
    public int score = 0;

    boolean fields[][] = new boolean[WORLD_WIDTH][WORLD_HEIGHT];
    Random random = new Random();
    float tickTime = 0;
    static float tick = TICK_INITIAL;
```

Как всегда, начинаем с определения пары констант. В данном случае это ширина и высота мира в клетках, значение, на которое мы увеличиваем счет каждый раз, когда мистер Ном съедает пятно, начальный временной интервал, используемый для передвижения мистера Нома *(tick)*, и значение, на которое мы уменьшаем *tick* каждый раз, когда мистер Ном съедает 10 пятен, чтобы немного ускорить персонажа.

Далее используем несколько общедоступных членов, которые содержат экземпляры классов Snake и Stain, флаг, который хранит информацию о том, закончена ли игра, а также текущий счет.

Определяем еще четыре закрытых члена пакета: двухмерный массив, который будем использовать для того, чтобы разместить новое пятно, экземпляр класса Random, с помощью которого будем получать случайные числа, чтобы разместить пятно и сгенерировать его тип, переменная счетчика времени tickTime, к которой будем прибавлять дельту, и текущая продолжительность tick, определяющая, насколько часто мы продвигаем мистера Нома.

```
public World() {
        snake = new Snake();
        placeStain();
    }
```

В конструкторе создаем экземпляр класса Snake, который будет содержать начальную конфигурацию, показанную на рис. 6.6. Мы также разместим первое пятно в случайном месте с помощью метода placeStain().

```
private void placeStain() {
        for (int x = 0; x < WORLD_WIDTH; x++) {
            for (int y = 0; y < WORLD_HEIGHT; y++) {
                fields[x][y] = false;
            }
        }

        int len = snake.parts.size();
        for (int i = 0; i < len; i++) {
            SnakePart part = snake.parts.get(i);
            fields[part.x][part.y] = true;
        }

        int stainX = random.nextInt(WORLD_WIDTH);
        int stainY = random.nextInt(WORLD_HEIGHT);
        while (true) {
            if (fields[stainX][stainY] == false)
                break;
            stainX += 1;
            if (stainX >= WORLD_WIDTH) {
                stainX = 0;
                stainY += 1;
                if (stainY >= WORLD_HEIGHT) {
                    stainY = 0;
                }
            }
        }
        stain = new Stain(stainX, stainY, random.nextInt(3));
    }
```

Метод placeStain() реализует стратегию размещения, описанную выше. Мы начинаем с очистки ячеек массива. Далее устанавливаем все ячейки, занятые частями мистера Нома в true. После сканируем массив в поисках свободной ячейки, начиная из случайного места. Как только мы нашли свободную ячейку, создаем пятно случайного типа. Обратите внимание, что если все клетки заняты мистером Номом, цикл никогда не закончится. Мы исключим такую ситуацию в следующем методе.

```
public void update(float deltaTime) {
        if (gameOver)
            return;

        tickTime += deltaTime;

        while (tickTime > tick) {
```

```
            tickTime -= tick;
            snake.advance();
            if (snake.checkBitten()) {
                gameOver = true;
                return;
            }

            SnakePart head = snake.parts.get(0);
            if (head.x == stain.x && head.y == stain.y) {
                score += SCORE_INCREMENT;
                snake.eat();
                if (snake.parts.size() == WORLD_WIDTH * WORLD_HEIGHT) {
                    gameOver = true;
                    return;
                } else {
                    placeStain();
                }

                if (score % 100 == 0 && tick - TICK_DECREMENT > 0) {
                    tick -= TICK_DECREMENT;
                }
            }
        }
    }
}
```

Метод `update()` отвечает за обновление класса `World` и всех объектов в нем в зависимости от значения дельты времени, которое мы ему передаем. Этот метод будет вызываться для каждого кадра игрового экрана, чтобы мир игры постоянно обновлялся. Сначала проверяем, окончена ли игра. Если она окончена, то, естественно, нам не нужно ничего обновлять. Далее прибавляем дельту времени к нашему счетчику. Цикл `while` будет задействовать столько `tick`, сколько накопилось (например, когда `tickTime` равен 1,2, а один `tick` равен 0,5 секунды, необходимо обновить мир дважды, оставив 0,2 секунды в счетчике). Этот прием называется моделированием с фиксированным шагом времени (fixed-time-step simulation).

В каждой итерации мы сначала отнимаем интервал `tick` от счетчика. Далее приказываем мистеру Ному двигаться. Мы проверяем, не укусил ли он себя. Если укусил, устанавливаем флаг «Игра окончена» (`gameOver`). Затем проверяем, не находится ли голова мистера Нома на той же клетке, что и пятно. Если находится, увеличиваем количество набранных очков и указываем мистеру Ному вырасти на одну часть. Далее проверяем, не состоит ли мистер Ном из такого же количества частей, что и его мир. Если состоит, игра окончена. Во всех других случаях размещаем новое пятно с помощью метода `placeStain()`. Последнее, что мы делаем, — проверяем, съел ли мистер Ном еще 10 пятен. Если съел и наш порог выше нуля, уменьшаем порог на 0,05 секунды. Наш `tick` станет короче, что заставит мистера Нома двигаться быстрее.

Итак, мы закончили написание множества классов модели. Теперь реализуем игровой экран.

Класс GameScreen

Нам осталось реализовать еще всего один экран. Рассмотрим, что этот экран должен делать.

- Как описано в проекте мистера Нома в главе 3, программа может находиться всего в четырех состояниях: ожидание, пока пользователь подтвердит, что он готов; режим работы; ожидание в режиме паузы и ожидание, пока пользователь нажмет кнопку в состоянии «Игра окончена». Опишем эти состояния подробнее:
  - в состоянии готовности мы просто просим пользователя дотронуться до экрана, чтобы запустить игру;
  - в работающем состоянии обновляем мир, визуализируем его, а также приказываем мистеру Ному повернуть налево или направо, когда пользователь нажимает одну из кнопок внизу экрана;
  - в состоянии паузы мы просто показываем две команды: вернуться к игре и выйти из нее;
  - в состоянии «Игра окончена» сообщаем пользователю, что игра окончена и предоставляем ему кнопку, чтобы он мог вернуться в главное меню.
- Для каждого состояния у нас есть различные методы для реализации обновления и текущего состояния (update и present), поскольку каждое состояние отвечает за различные вещи и показывает разные пользовательские интерфейсы.
- Когда игра окончена, необходимо убедиться, что мы сохранили результат, если он является рекордным.

Для этого нам необходимо немного больше кода, чем обычно. Разделим листинг программы данного класса. Перед тем как вплотную заняться кодом, выясним, как будут расположены различные элементы пользовательского интерфейса в каждом состоянии.

Рисунок 6.8 демонстрирует четыре различных состояния.

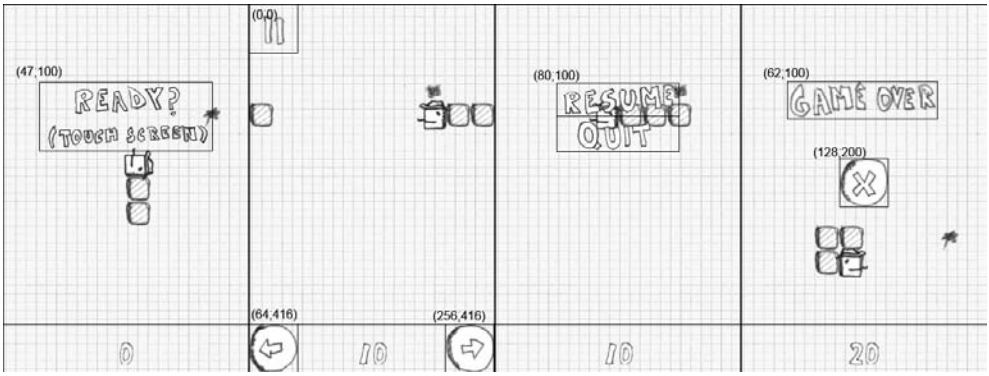

Рис. 6.8. Игровой экран в четырех состояниях: готовность, работа, пауза, игра окончена

Обратите внимание, что мы также визуализируем набранные очки в нижней части экрана вместе с линией, которая отделяем мир мистера Нома от кнопок внизу. Набранные очки визуализируются так же, как и на экране HighscoreScreen. Кроме того, мы размещаем их по центру, основываясь на ширине полоски набранных очков.

Последний недостающий фрагмент информации касается того, как визуализировать мир мистера Нома, основываясь на его модели. В общем-то, это весьма просто. Посмотрите на рис. 6.1 и 6.5 снова. Каждая клетка имеет размер ровно 32 × 32 пиксела. Изображения пятна также имеют размер 32 × 32 пиксела, как и части мистера Нома. Размер головы мистера Нома во всех направлениях составляет 42 × 42 пиксела, так что она не помещается полностью на одной клетке.

Тем не менее это не проблема. Все, что нам надо, чтобы визуализировать мир мистера Нома, — взять каждое пятно и часть мистера Нома и умножить координаты мира на 32, чтобы попасть в центр объекта в пикселах на экране. Например, центр пятна с координатами (3; 2) в мире будет находиться на 96 × 64 экрана. В такой ситуации остается только выбрать соответствующий объект и отобразить его, центрируя по координатам. Рассмотрим код. В листинге 6.12 приведен класс GameScreen.

Листинг 6.12. GameScreen.java

```
package com.badlogic.androidgames.mrnom;

import java.util.List;

import android.graphics.Color;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Graphics;
import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.Pixmap;
import com.badlogic.androidgames.framework.Screen;

public class GameScreen extends Screen {
    enum GameState {
        Ready,
        Running,
        Paused,
        GameOver
    }

    GameState state = GameState.Ready;
    World world;
    int oldScore = 0;
    String score = "0";
```

Начинаем с определения перечня под названием GameState, который задает четыре состояния (готов, работа, пауза, игра окончена). Далее определяем член, который содержит текущее состояние экрана, и член, который включает в себя экземпляр класса World, а также еще два члена, содержащие текущие набранные

очки в виде целого числа и строки. Причина, по которой нам необходимы последние два члена, состоит в том, что мы не хотим постоянно создавать новые строки из переменной World.score каждый раз, когда отрисовываем счет. Вместо этого мы кэшируем строку и создаем новую, только когда счет изменяется. Таким образом, у сборщика мусора не возникает проблем.

```
public GameScreen(Game game) {
    super(game);
    world = new World();
}
```

Конструктор данного класса просто вызывает конструктор своего родительского класса и создает новый экземпляр класса World. Игровой экран будет в состоянии готовности после того, как конструктор вернет управление вызывающей стороне.

```
@Override
    public void update(float deltaTime) {
        List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
        game.getInput().getKeyEvents();

        if(state == GameState.Ready)
            updateReady(touchEvents);
        if(state == GameState.Running)
            updateRunning(touchEvents, deltaTime);
        if(state == GameState.Paused)
            updatePaused(touchEvents);
        if(state == GameState.GameOver)
            updateGameOver(touchEvents);
    }
```

Далее следует метод update() экрана. Он выбирает TouchEvent и KeyEvent из модуля ввода и передает их для обновления одному из четырех соответствующих методов, которые мы реализуем для каждого состояния в зависимости от текущего состояния:

```
private void updateReady(List<TouchEvent> touchEvents) {
        if(touchEvents.size() > 0)
            state = GameState.Running;
    }
```

Следующий метод называется updateReady(). Он будет вызываться, когда экран находится в состоянии готовности. Он проверяет, был ли нажат экран. Если нажатие было, он изменяет состояние на рабочее.

```
private void updateRunning(List<TouchEvent> touchEvents, float deltaTime) {
        int len = touchEvents.size();
        for(int i = 0; i < len; i++) {
            TouchEvent event = touchEvents.get(i);
            if(event.type == TouchEvent.TOUCH_UP) {
                if(event.x < 64 && event.y < 64) {
                    if(Settings.soundEnabled)
                        Assets.click.play(1);
                    state = GameState.Paused;
```

```
                    return;
                }
            }
            if(event.type == TouchEvent.TOUCH_DOWN) {
                if(event.x < 64 && event.y > 416) {
                    world.snake.turnLeft();
                }
                if(event.x > 256 && event.y > 416) {
                    world.snake.turnRight();
                }
            }
        }

        world.update(deltaTime);
        if(world.gameOver) {
            if(Settings.soundEnabled)
                Assets.bitten.play(1);
            state = GameState.GameOver;
        }
        if(oldScore != world.score) {
            oldScore = world.score;
            score = "" + oldScore;
            if(Settings.soundEnabled)
                Assets.eat.play(1);
        }
    }
```

Метод `updateRunning()` проверяет, нажата ли клавиша паузы в верхнем левом углу экрана. Затем этот метод проверяет, была ли нажата какая-либо из кнопок контроллера внизу экрана. Обратите внимание, что мы не проверяем здесь события отпускания (`touch-up`). Если какая-либо кнопка была нажата, мы сообщаем экземпляру класса `Snake` в классе `World` повернуть налево или направо. Все правильно, метод `updateRunning()` содержит код контроллера нашей схемы MVC. После того как все события касания проверены, мы приказываем миру обновиться, передавая ему дельту времени. Если класс `World` сигнализирует, что игра окончена, мы переходим к соответствующему состоянию, а также воспроизводим звук `bitten.ogg`. Далее проверяем, отличается ли прежнее количество очков, которое мы поместили в кэш, от результата, который хранит `World`. Если отличается, нам становятся известны две вещи: мистер Ном съел пятно и строка результатов должна быть изменена. В этом случае мы проигрываем звук `eat.ogg`. Вот, собственно, и все, что касается обновления текущего состояния.

```
private void updatePaused(List<TouchEvent> touchEvents) {
        int len = touchEvents.size();
        for(int i = 0; i < len; i++) {
            TouchEvent event = touchEvents.get(i);
            if(event.type == TouchEvent.TOUCH_UP) {
                if(event.x > 80 && event.x <= 240) {
                    if(event.y > 100 && event.y <= 148) {
                        if(Settings.soundEnabled)
                            Assets.click.play(1);
```

```
                    state = GameState.Running;
                    return;
                }
                if(event.y > 148 && event.y < 196) {
                    if(Settings.soundEnabled)
                        Assets.click.play(1);
                    game.setScreen(new MainMenuScreen(game));
                    return;
                }
            }
        }
    }
```

Метод `updatePaused()` опять-таки просто проверяет, была ли нажата одна из команд меню, и переходит к соответствующему состоянию.

```
private void updateGameOver(List<TouchEvent> touchEvents) {
        int len = touchEvents.size();
        for(int i = 0; i < len; i++) {
            TouchEvent event = touchEvents.get(i);
            if(event.type == TouchEvent.TOUCH_UP) {
                if(event.x >= 128 && event.x <= 192 &&
                   event.y >= 200 && event.y <= 264) {
                    if(Settings.soundEnabled)
                       Assets.click.play(1);
                    game.setScreen(new MainMenuScreen(game));
                    return;
                    }
            }
        }
    }
```

Метод `updateGameOver()` также просто проверяет, нажата ли кнопка в центре экрана. Если нажата, выполняется переход назад в главное меню.

```
@Override
    public void present(float deltaTime) {
        Graphics g = game.getGraphics();

        g.drawPixmap(Assets.background, 0, 0);
        drawWorld(world);
        if(state == GameState.Ready)
            drawReadyUI();
        if(state == GameState.Running)
            drawRunningUI();
        if(state == GameState.Paused)
            drawPausedUI();
        if(state == GameState.GameOver)
            drawGameOverUI();

        drawText(g, score, g.getWidth() / 2 – score.length()*20 / 2,
                    g.getHeight() -42);
    }
```

Далее следуют методы отображения. Метод `present()` сначала рисует фоновое изображение, поскольку оно необходимо нам во всех четырех состояниях. Далее он вызывает соответствующий отрисовочный метод для состояния, в котором мы находимся. Наконец, он визуализирует мир мистера Нома и рисует количество набранных очков в нижней центральной части экрана.

```
private void drawWorld(World world) {
        Graphics g = game.getGraphics();
        Snake snake = world.snake;
        SnakePart head = snake.parts.get(0);
        Stain stain = world.stain;


        Pixmap stainPixmap = null;
        if(stain.type == Stain.TYPE_1)
            stainPixmap = Assets.stain1;
        if(stain.type == Stain.TYPE_2)
            stainPixmap = Assets.stain2;
        if(stain.type == Stain.TYPE_3)
            stainPixmap = Assets.stain3;
        int x = stain.x * 32;
        int y = stain.y * 32;
        g.drawPixmap(stainPixmap, x, y);

        int len = snake.parts.size();
        for(int i = 1; i < len; i++) {
            SnakePart part = snake.parts.get(i);
            x = part.x * 32;
            y = part.y * 32;
            g.drawPixmap(Assets.tail, x, y);
        }

        Pixmap headPixmap = null;
        if(snake.direction == Snake.UP)
            headPixmap = Assets.headUp;
        if(snake.direction == Snake.LEFT)
            headPixmap = Assets.headLeft;
        if(snake.direction == Snake.DOWN)
            headPixmap = Assets.headDown;
        if(snake.direction == Snake.RIGHT)
            headPixmap = Assets.headRight;
        x = head.x * 32 + 16;
        y = head.y * 32 + 16;
        g.drawPixmap(headPixmap, x – headPixmap.getWidth() / 2,
                        y -headPixmap.getHeight() / 2);
    }
```

Метод `drawWorld()` рисует мир примерно по тому же принципу, что был рассмотрен выше. Он начинает с выбора `Pixmap` для визуализации пятна, затем рисует пятно и центрирует его по горизонтали в нужном месте экрана. Далее визуализи-

руем все части хвоста мистера Нома, что весьма просто. Затем выбираем, какой Pixmap для головы следует использовать, основываясь на направлении, в котором двигается мистер Ном. Рисуем Pixmap в зависимости от положения головы, переведенного в экранные координаты. Размещаем его по центру, как и остальные объекты. Вот код view в MVC.

```
private void drawReadyUI() {
        Graphics g = game.getGraphics();

        g.drawPixmap(Assets.ready, 47, 100);
        g.drawLine(0, 416, 480, 416, Color.BLACK);
    }

    private void drawRunningUI() {
        Graphics g = game.getGraphics();

        g.drawPixmap(Assets.buttons, 0, 0, 64, 128, 64, 64);
        g.drawLine(0, 416, 480, 416, Color.BLACK);
        g.drawPixmap(Assets.buttons, 0, 416, 64, 64, 64, 64);
        g.drawPixmap(Assets.buttons, 256, 416, 0, 64, 64, 64);
    }

    private void drawPausedUI() {
        Graphics g = game.getGraphics();

        g.drawPixmap(Assets.pause, 80, 100);
        g.drawLine(0, 416, 480, 416, Color.BLACK);
    }

    private void drawGameOverUI() {
        Graphics g = game.getGraphics();

        g.drawPixmap(Assets.gameOver, 62, 100);
        g.drawPixmap(Assets.buttons, 128, 200, 0, 128, 64, 64);
        g.drawLine(0, 416, 480, 416, Color.BLACK);
    }

    public void drawText(Graphics g, String line, int x, int y) {

        int len = line.length();
        for (int i = 0; i < len; i++) {
            char character = line.charAt(i);

            if (character == ' ') {
                x += 20;
                continue;
            }

            int srcX = 0;
            int srcWidth = 0;
```

```
            if (character == '.') {
                srcX = 200;
                srcWidth = 10;
            } else {
                srcX = (character - '0') * 20;
                srcWidth = 20; }
            }

            g.drawPixmap(Assets.numbers, x, y, srcX, 0, srcWidth, 32);
            x += srcWidth;
        }
    }
```

В методах `drawReadUI()`, `drawRunningUI()`, `drawPausedUI()` и `drawGameOverUI()` нет ничего нового. Они выполняют все ту же визуализацию пользовательского интерфейса, основанную на координатах, которые показаны на рис. 6.8. Метод `drawText()` аналогичен методу, использованному в `HighscoreScreen`, так что мы не будем его обсуждать.

```
    @Override
    public void pause() {
        if(state == GameState.Running)
            state = GameState.Paused;

        if(world.gameOver) {
            Settings.addScore(world.score);
            Settings.save(game.getFileIO());
        }
    }

    @Override
    public void resume() {

    }

    @Override
    public void dispose() {

    }
}
```

Есть и еще один важнейший метод — `pause()`, который вызывается, когда активность ставится на паузу или игровой экран заменяется на какой-либо другой. Это самое подходящее место для сохранения настроек. Сначала мы устанавливаем состояние нашей игры как paused (Приостановлено). Если метод `paused()` вызывается в связи с приостановкой работы активности, пользователю обязательно будет предложено возобновить игру с того места, на котором игра была прервана. Это весьма неплохо, так как можно сразу приступить к игре с того момента, на котором он закончил. Далее проверяем, не находится ли игра в состоянии «Игра окончена». Если это так — добавляем результат пользователя в таблицу рекордов (или не

добавляем, это зависит от значения) и сохраняем все настройки во внешнем хранилище.

Вот и все. Мы написали полноценную игру на Android с нуля. Мы можем гордиться собой, поскольку справились со всеми аспектами, которые позволят нам создать практически любую желаемую игру. Теперь нам остались только всякие мелочи.

Подводя итог

В этой главе мы полностью реализовали игру, основанную на нашем фреймворке. Вы узнали, почему следует отделять модель от вида и контроллера, а также изучили, почему не надо описывать мир игры на уровне пикселов. Мы можем взять созданный код и частично выполнить визуализацию при помощи OpenGL ES, сделав мистера Нома трехмерным. Мы можем также сделать текущую визуализацию интереснее, обогатив мистера Нома анимацией, цветом, новой игровой механикой и т. д. Мы лишь чуть прикоснулись к доступным возможностям.

Перед тем как вы продолжите читать книгу, я предлагаю вам взять код игры и немного поэкспериментировать с ним. Добавьте какие-нибудь новые режимы игры, бонусы и врагов — все, что вы сможете придумать.

В следующей главе мы улучшим наши навыки графического программирования, чтобы сделать игры немного более привлекательными, а также выполним первые шаги в трехмерное пространство.

7 OpenGL ES: первое представление

Игра «Мистер Ном» получилась очень удачной. Благодаря тому, что мы хорошо поработали на этапе проектирования и фреймворку, который мы написали, реализация «Мистера Нома» не составила труда. Самое хорошее в игре то, что она работает гладко даже на очень слабых устройствах. «Мистер Ном» не отличается особыми графическими изысками или сложной схемой игры, так что использовать API `Canvas` для визуализации было весьма целесообразно.

Тем не менее, если вы захотите сделать что-то более сложное, например игру в стиле Replica Island, вы столкнетесь с серьезной проблемой — `Canvas` не справляется со сложной графикой подобных игр. Если же вы захотите иметь игру в 3D, `Canvas` также вам не поможет. Так что же делать?

В подобных случаях на помощь приходит OpenGL ES. В этой главе мы сначала кратко рассмотрим, что такое OpenGL ES и как он работает. Затем сосредоточимся на использовании OpenGL ES для 2D-графики, не углубляясь в сложные математические тонкости применения этого API для 3D-графики (об этом поговорим в последней главе). Начнем с малого, так как OpenGL ES может быть очень сложна. Ну что, давайте познакомимся с OpenGL ES.

Что такое OpenGL ES и почему об этом стоит задуматься

OpenGL ES — это промышленный стандарт для графического программирования (3D), предназначенный для мобильных и встраиваемых устройств. Он обслуживается Khronos Group — объединением таких компаний, как ATI, NVIDIA, и Intel, которые вместе определяют и расширяют этот стандарт.

Если говорить о стандартах, на данный момент существуют три версии OpenGL ES: 1.0, 1.1 и 2.0. В этой книге мы собираемся рассмотреть первые две. Все устройства, работающие на Android, поддерживают OpenGL ES 1.0, большинство из них также поддерживают 1.1, которая отличается от 1.0 некоторыми новыми особенностями. Как ни странно, OpenGL ES 2.0 несовместима с версиями 1x. Вы можете использовать или 1x, или 2.0, однако не две версии одновременно. Дело в том, что версии 1x используют модель программирования «Контейер с фиксированной функциональ-

ностью» (fixed-function pipeline), в то время, как 2.0 позволяет программно описать элементы потокового рендеринга с помощью так называемых шейдеров.

OpenGL ES 1.x более чем достаточно для большинства игр, так что мы будем работать именно с этим стандартом.

ПРИМЕЧАНИЕ

Эмулятор поддерживает только OpenGL ES 1.0. Тем не менее реализация немного ненадежная, так что никогда не тестируйте программу на эмуляторе, используйте настоящее устройство.

OpenGL ES — это API в виде набора заголовочных файлов на языке C, которые предоставлены Khronos Group вместе с очень подробной спецификацией того, как должны работать API, описанные в этих заголовках. Спецификация, в частности, описывает, как должны визуализироваться пикселы и строки. Производители оборудования используют эту спецификацию и реализуют ее в своих графических процессорах. Качество подобных реализаций немного различается: некоторые компании строго придерживаются стандарта (PowerVR), в то время как другие испытывают с этим сложности. Иногда это приводит к ошибкам реализации, связанным с графическим процессором, эти ошибки обусловлены не ОС Android, а самим оборудованием. При изучении OpenGL ES мы также рассмотрим возможные проблемы, специфичные для отдельных устройств.

ПРИМЕЧАНИЕ

OpenGL ES в какой-то мере подобен более мощному стандарту OpenGL, рассчитанному на работу с настольными системами. OpenGL ES основан на OpenGL, однако некоторый функционал в OpenGL ES представлен в ограниченном объеме или вообще отсутствует. Тем не менее мы можем написать приложение, которое будет соответствовать обеим спецификациям, что весьма удобно, если вы хотите, чтобы ваша программа запускалась и на ПК.

Что же делает OpenGL ES? Если ответить в нескольких словах, это достаточно простая программа для визуализации треугольников. Однако мы с вами разберем все подробно.

Модель разработки: аналогия

OpenGL ES является общим программным API для 3D-графики. Он отличается достаточно удобными и (надеюсь) легкими для понимания моделями разработки, которые мы сможем проиллюстрировать простой аналогией.

Представьте, что OpenGL ES работает как фотоаппарат. Чтобы сделать снимок, вам сначала надо прийти на место, которое вы хотите снять. Ваша сцена состоит из нескольких объектов, допустим, из стола с несколькими предметами на нем. Все эти предметы имеют местоположение и ориентацию относительно камеры. Они состоят из различных материалов и имеют разные текстуры. Стекло прозрачно и немного отражает, стол, скорее всего, сделан из дерева, на обложке журнала видна фотография политика и т. д. Некоторые предметы даже могут двигаться, например муха, от которой вы никак не можете избавиться. Ваш фотоаппарат также имеет ряд характеристик: фокусное расстояние, зона обзора,

разрешение изображения, размер фотографии, его собственное местоположение и ориентация в пространстве (относительно начала координат). Пусть даже все объекты и фотоаппарат двигаются, но когда вы нажимаете кнопку, чтобы сделать снимок, вы получаете застывшее изображение, где ничего не движется (мы в данном случае игнорируем скорость работы затвора, из-за которой изображение может получиться немного размытым).

В этот кратчайший момент ничего не движется и остается в таком положении, какое запечатлено на фотографии со всеми конфигурациями позиций, ориентациями, текстурами, материалами и освещением. На рис. 7.1 показана абстрактная сцена с фотоаппаратом, светом и тремя предметами из различных материалов.

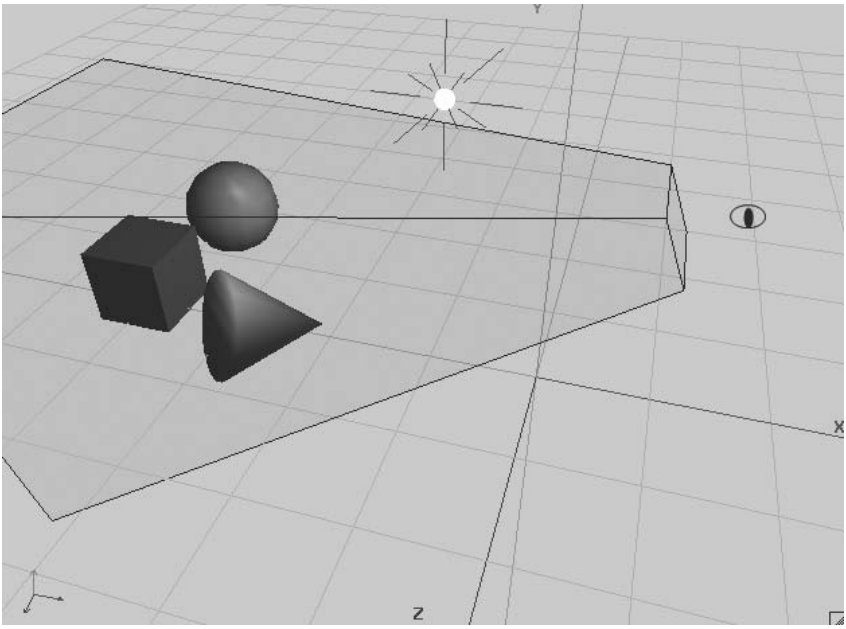

Рис. 7.1. Абстрактная сцена

Каждый объект имеет позицию и ориентацию относительно начала координат сцены. Фотоаппарат, обозначенный символом «глаз», также расположен относительно начала координат. Пирамида на рис. 7.1 — это так называемый отображаемый объем или видимое пространство, то есть конус, который показывает, какую часть сцены захватывает камера и как камера ориентирована. Маленький белый шарик с лучами обозначает источник света, который также расположен относительно начала координат.

Мы можем напрямую перенести эту сцену в OpenGL ES, но для начала нам нужно описать несколько вещей.

- *Объекты (или модели).* Как правило, они определяются их геометрией, цветом, текстурами и материалом. Геометрия определяется как набор треугольников. Каждый треугольник состоит из трех точек в трехмерном пространстве, так

что у нас есть *x*-, *y*- и *z*-координаты, расположенные относительно начала координат, как на рис. 7.1. Обратите внимание, что ось *z* направлена в нашу сторону. Как правило, цвет определяется в уже знакомом нам формате RGB. Текстуры и материалы требуют более подробного описания. Мы поговорим о них позже.

- *Освещение*. OpenGL ES предлагает несколько типов освещения с различными атрибутами. Это просто математические объекты, которые имеют местоположение и/или направление в 3D-пространстве, а также атрибуты, в частности, цвет.
- *Камера*. Это также математический объект, расположенный в 3D-пространстве. У него тоже есть параметры, которые определяют, какую часть изображения мы видим. Все эти параметры вместе определяют отображаемый объем, также называемый видимым пространством (усеченная пирамида, показанная на рис. 7.1). Камера фиксирует все, что находится внутри этой пирамиды, а все, что располагается вне ее не попадает на фотографию.
- *Область просмотра*. Определяет размер и разрешение конечного изображения. Этот параметр можно сравнить с типом пленки, вставляемой в фотоаппарат, или с разрешением изображения, когда снимок сделан на цифровую камеру.

При наличии всего этого OpenGL ES может создать 2D-изображение с точки обзора фотоаппарата. Обратите внимание, что мы описываем все в 3D-пространстве. Как же это будет выглядеть в OpenGL ES в двухмерном пространстве?

Проекции

2D-отображение создается с помощью проекций. Мы уже говорили о том, что OpenGL ES в первую очередь работает с треугольниками. Один треугольник имеет три точки в 3D-пространстве. Чтобы визуализировать такой треугольник в фреймбуфере, OpenGL ES необходимо знать координаты этих 3D-точек в пиксельной координатной системе фреймбуфера. Если ему известны три данные координаты точек, он может просто нарисовать в фреймбуфере все пикселы, которые находятся внутри треугольника. Мы можем даже нарисовать собственную небольшую реализацию OpenGL ES, проецируя 3D-точки в 2D и просто рисуя линии между ними с помощью `Canvas`.

Существует два типа проекций, которые обычно используются в 3D-графике.

- *Параллельная, или ортогональная, проекция*. Если вы когда-либо работали с CAD-приложениями, то наверняка с ней сталкивались. В параллельной проекции неважно, насколько предмет удален от фотоаппарата. Предмет всегда будет одного и того же размера на конечном снимке. Как правило, этот тип проекции используется для визуализации 2D-графики в OpenGL ES.
- *Перспективные проекции*. Так мы видим. Предметы, более удаленные от нас, кажутся на нашей сетчатке меньше. Данный тип проекции обычно используется для 3D-графики в OpenGL ES.

В обоих случаях нам необходима плоскость проекции. Она аналогична сетчатке глаза. Именно на эту плоскость падает свет, и образуется конечный снимок.

Сетчатка, в отличие от математического пространства, имеет небольшой конечный размер. Сетчатка OpenGL ES равна прямоугольнику на вершине конуса отображения, как на рис. 7.1.

Эта часть конуса отображения показывает, как будет направлена проекция в OpenGL ES. Она называется ближней плоскостью отсечения и имеет свою двухмерную систему координат. На рис. 7.2 показана эта плоскость отсечения с точки зрения камеры с наложением системы координат.

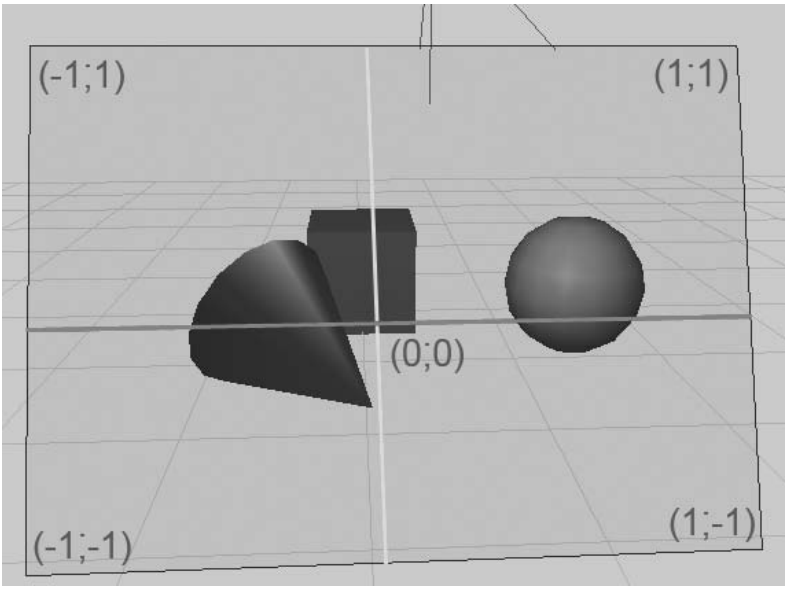

Рис. 7.2. Ближняя плоскость отсечения (она же — плоскость проекции) и ее система координат

Обратите внимание на то, что система координат никак не фиксирована. Мы можем ею управлять, то есть можем работать в такой системе координат проекции, в какой захотим (например, можем указать OpenGL ES, что начало координат должно быть в нижнем левом углу, и сделать так, чтобы видимая зона нашей «сетчатки» состояла из 480 единиц по оси x и 320 единиц по оси y). Да, OpenGL ES позволяет нам установить любую систему координат для проецируемых точек.

Когда мы укажем конус отображения, OpenGL ES возьмет все точки треугольника и проведет от каждой из них луч через плоскость проекции. Различие между параллельной проекцией и перспективной проекцией заключается в том, какое направление имеют эти лучи. На рис. 7.3 показана разница между ними (вид сверху).

Перспективная проекция позволяет провести лучи из точек треугольника через камеру (или в нашем случае — глаз). Предметы, находящиеся дальше, кажутся на проекционной плоскости более мелкими. Когда мы используем параллельную проекцию, лучи проводятся перпендикулярно плоскости проекции. В этом случае предмет сохранит свои размеры на плоскости проекции вне зависимости от того, насколько он удален.

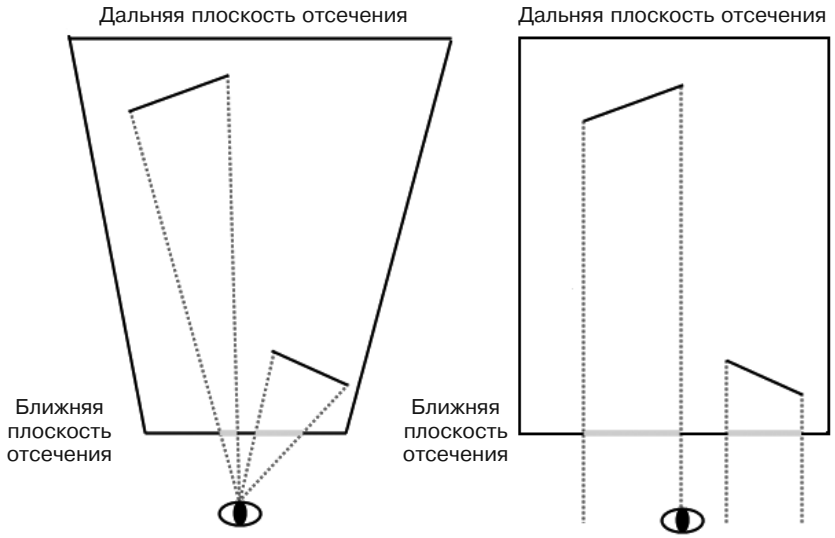

Рис. 7.3. Проекция перспективы *(слева)* и параллельная проекция *(справа)*

Как я уже отмечал, плоскость проекции на языке OpenGL ES называется ближней плоскостью отсечения. Все стороны конуса отображения имеют похожие названия. Самая отдаленная часть именуется дальней плоскостью отсечения. Остальные называются левой, правой, верхней и нижней плоскостями отсечения. Все, что находится за этими плоскостями, отображаться не будет. Объекты, которые частично находятся в конусе отображения, будут разрезаны этими плоскостями. Это значит, что часть предмета будет отсечена. Именно поэтому такие плоскости и называют плоскостями отсечения.

Может возникнуть вопрос: почему конус в параллельной проекции на рис. 7.3 прямоугольный? Дело в том, что проекция зависит от того, как мы определим плоскости отсечения. В случае перспективной проекции левая, правая, верхняя и нижняя плоскости неперпендикулярны ближней и дальней плоскостям отсечения. См. рис. 7.3, который показывает только левую и правую плоскости отсечения. В случае с параллельной проекцией эти плоскости перпендикулярны, благодаря чему OpenGL ES отображает все предметы в одинаковом масштабе независимо от того, насколько предмет удален от камеры.

Нормализованное пространство устройства и область просмотра

Когда OpenGL ES уже известны проецируемые на ближнюю плоскость отсечения точки треугольника, мы наконец-то можем перевести их в пиксельные координаты фреймбуфера. Для этого необходимо трансформировать точки в так называемое нормализированное пространство устройства. Оно соответствует системе координат, показанной на рис. 7.2. Основываясь на координатах нормализованного

пространства устройства, OpenGL ES подсчитывает конечные пиксельные координаты фреймбуфера с помощью следующей простой формулы:

```
pixelX = (norX + 1) / (viewportWidth + 1) + norX
pixelY = (norY + 1) / (viewportHeight +1) + norY
```

где `norX` и `norY` — координаты 3D-точки нормализованных координат устройства, а `viewportWidth` и `viewportHeight` — размеры точки наблюдения в пикселах на осях x и y. Не стоит волноваться о нормализованных координатах устройства, поскольку OpenGL сам сделает все преобразования автоматически. О чем стоит переживать, так это о точке наблюдения и конусе отображения.

Матрицы

Ниже будет рассказано, как настроить конус отображения и, следовательно, проекцию. OpenGL ES выражает проекции в виде так называемых матриц. Нам для наших целей необязательно знать все свойства матриц. Все, что нам нужно, это быть в курсе того, что они делают с точками, которые мы описываем в нашей сцене. Рассмотрим краткое описание матрицы.

- Матрица кодирует преобразования, которые следует применить к точке. Преобразование может представлять собой проекцию, перемещение (переход точки в другую точку), поворот вокруг другой точки и оси, масштабирование, а также многое другое.
- При умножении матрицы на точку мы применяем изменения к точке. Например, при умножении точки на матрицу, в которой закодирован переход на 10 единиц по оси x, точка передвинется на 10 единиц по оси x и, как следствие, изменит свои координаты.
- Мы можем объединять преобразования, хранящиеся в нескольких матрицах, в одну матрицу путем умножения матриц друг на друга. Когда мы умножаем одну объединенную матрицу на точку, все преобразования, хранящиеся в этой матрице, будут применены к этой точке. Порядок, в котором будут применяться преобразования, зависит от порядка, в котором мы перемножали матрицы.
- Существует особая матрица, называемая единичной (identity matrix). Если мы умножим матрицу или точку на нее, ничего не произойдет. Считайте умножение точки или матрицы на единичную матрицу как умножение на единицу. Она не имеет никакого эффекта. Назначение единичной матрицы станет понятнее, когда мы узнаем, как OpenGL ES обрабатывает матрицы. Типичная проблема: что было раньше — яйцо или курица?

ПРИМЕЧАНИЕ

Когда мы говорим о точках, в данном контексте мы имеем в виду 3D-векторы.

OpenGL ES содержит три вида матриц, применяемых к точкам моделей.

- *Модельно-видовая матрица*. Мы можем использовать эту матрицу для того, чтобы двигать, поворачивать или изменять размеры точек нашего треугольника (это *модельная* часть нашей матрицы модели представления). Данная матрица также применяется для настройки месторасположения и ориентации нашей камеры (эта часть — представление).
- *Матрица проекции*. Название говорит само за себя — данная матрица кодирует проекцию и, как следствие, конус отображения нашей камеры.
- *Матрица текстур*. Данная матрица позволяет нам управлять так называемыми координатами текстур (которые мы обсудим позже). Тем не менее мы будем избегать применения данной матрицы в этой книге, поскольку из-за несовершенства драйверов данная часть OpenGL ES отличается большим количеством ошибок на некоторых устройствах.

Конвейер визуализации

OpenGL ES отслеживает эти три матрицы. Каждый раз, когда мы устанавливаем одну из них, она сохраняется в памяти до тех пор, пока мы снова ее не изменим. Говоря языком OpenGL ES, это называется *состоянием*. OpenGL не просто отслеживает состояние матрицы, а также следит за тем, хотим ли мы произвести альфа-смешивание, учитывать освещение, за тем, какая текстура должна быть применена к нашей геометрии и т. д. Фактически OpenGL ES представляет собой одну большую машину состояний. Мы устанавливаем текущее состояние, добавляем геометрию объектов и приказываем OpenGL ES визуализировать изображение. Рассмотрим, как треугольник проходит через механизм визуализации. На рис. 7.4 показан достаточно упрощенный вид конвейера OpenGL ES:

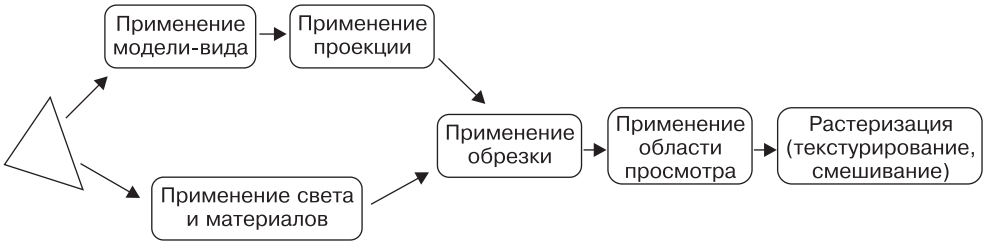

Рис. 7.4. Путь треугольника

Путь треугольника через конвейер выглядит следующим образом.

1. Сначала наш треугольник трансформируется матрицей модели представления. Это значит, все его точки умножаются на эту матрицу. Данное умножение фактически передвинет точки треугольника в необходимое место.
2. Итог данной операции умножается на матрицу проекции, которая трансформирует 3D-точки в двухмерную плоскость проекции.

3. Между предыдущими двумя шагами или одновременно с ними текущие установленные параметры освещения и материалов также применяются к треугольнику, придавая ему цвет.
4. Когда все вышеперечисленное выполнено, проектируемый треугольник отсекается нашей «сетчаткой» и преобразуется в координаты фреймбуфера.
5. И наконец. OpenGL заполняет треугольник пикселами на основе цветов из освещаемой сцены, текстур, которые требуется применить к треугольнику и состояния смешивания, в котором каждый пиксел треугольника может или не может быть соединен с пикселом из фреймбуфера.

Нам необходимо выучить, как использовать геометрию и текстуры OpenGL ES и устанавливать состояния, использованные на каждом из предыдущих этапов. Перед тем как мы научимся этому, нам необходимо узнать, как Android предоставляет доступ к OpenGL ES.

ПРИМЕЧАНИЕ

Несмотря на то что приведенное обобщенное описание конвейера OpenGL ES в целом верно, оно крайне упрощено и не учитывает некоторые детали, которые будут важны в следующих главах. Необходимо запомнить, что когда OpenGL ES создает проекцию, в двухмерную координатную систему ничего не проецируется. Вместо этого выполняется проекция в так называемую однородную систему координат, которая фактически имеет четыре измерения. Чтобы это понять, необходимо хорошо разбираться в математике. Поэтому ради упрощения мы просто допустим, что OpenGL ES проецирует объекты на двухмерные координаты.

Перед стартом

В оставшейся части этой главы мы напишем множество маленьких примеров, как мы делали это в главе 4, когда обсуждали основы Android API. Мы будем использовать тот же самый стартовый класс, как и в главе 4, который покажет нам список тестовых `Activity`, которые мы можем запустить. Единственное, что изменится, — названия `Activity`, которые мы инстанцируем в обратном виде, и пакет, где они будут располагаться. Все примеры из оставшейся части этой главы будут располагаться в `com.badlogic.androidgames.glbasics`. Остаток кода не изменится. Наша новая стартовая активность будет называться `GLBasicsStarter`. Мы также скопируем весь исходный код из главы 5. Там содержатся классы нашего фреймворка, и мы, конечно, захотим снова ими воспользоваться. Наконец, мы напишем несколько новых классов фреймворка и вспомогательных классов, которые будут размещаться в `com.badlogic.androidgames.framework`.

Нам также снова понадобится файл манифеста. Поскольку все примеры будут представлять собой `Activity`, мы должны будем убедиться в том, что они объявлены в манифесте. Во всех примерах используется фиксированная ориентация (книжная или альбомная, в зависимости от примера). Android получает информацию о том, что эти примеры могут обрабатывать события типа `keyboard`, `keyboardHidden` и `orientationChange`.

Итак, давайте начнем.

GLSurfaceView: облегчает жизнь с 2008 года

Первым делом нам понадобится определенный тип View, который позволит нам рисовать с помощью OpenGL ES. В Android API такой View имеется. Он называется GLSurfaceView и является потомком класса SurfaceView, который мы уже применяли для отрисовки мира мистера Нома.

Нам также понадобится отдельный поток главного цикла, чтобы не занимать поток пользовательского интерфейса. А теперь сюрприз: GLSurfaceView уже сделал такой поток для нас. Нам лишь нужно реализовать интерфейс слушателя GLSurfaceView.Renderer и зарегистрировать его в GLSurfaceView. Данный интерфейс использует следующие три метода:

```
interface Renderer {
    public void onSurfaceCreated(GL10 gl, EGLConfig config);

    public void onSurfaceChanged(GL10 gl, int width, int height);

    public void onDrawFrame(GL10 gl);
}
```

Метод onSurfaceCreated() вызывается каждый раз, когда создается GLSurfaceView. В первый раз это происходит, когда мы запускаем Activity, а затем каждый раз, когда мы возвращаемся к Activity из режима паузы. Данный метод использует два параметра: экземпляр GL10 и EGLConfig. Экземпляр GL10 позволяет давать команды OpenGL ES. В свою очередь, EGLConfig просто сообщает нам атрибуты поверхности, например глубину цвета. Как правило, эта информация игнорируется. Мы установим геометрию и текстуры в методе onSurfaceCreated().

Метод onSurfaceChanged() вызывается каждый раз, когда изменяется размер поверхности. Мы получаем новую ширину и высоту поверхности в пикселах, а также экземпляр класса GL10, если хотим дать OpenGL ES какие-либо команды.

В методе onDrawFrame() происходит самое интересное. Он чем-то похож на наш метод Screen.render(), который вызывается как можно чаще потоком визуализации, который установил для нас GLSurfaceView. В этом методе происходит вся визуализация.

Кроме регистрации слушателя Renderer нам также необходимо вызвать метод GLSurfaceView.onPause()/onResume() в методе onPause()/onResume() нашей активности. Причина этого проста. GLSurfaceView запустит поток визуализации в своем методе onResume() и приостановит его в методе onPause(). Это значит, что наш слушатель не будет вызываться до тех пор, пока Activity стоит на паузе, так как поток визуализации, направляющий вызовы нашему слушателю, также стоит на паузе.

Однако здесь мы сталкиваемся с главной проблемой: каждый раз, когда Activity ставится на паузу, поверхность GLSurfaceView уничтожается. Когда Activity снова возобновляет работу (а мы вызываем GLSurfaceView.onResume()), GLSurfaceView создает экземпляр новой поверхности визуализации OpenGL ES и информирует нас об этом, вызывая наш метод слушателя onSurfaceCreated(). Все было бы хорошо,

если бы не одна проблема: все состояния OpenGL ES, которые мы устанавливаем, будут утеряны. Это также касается, в частности, текстур и многого другого, что в подобном случае придется перезагружать. Данная проблема также известна под названием «потеря контекста». Термин «контекст» употребляется здесь потому, что OpenGL ES ассоциирует так называемый контекст с каждой поверхностью, которую создает и которая содержит текущее состояние. Когда мы удаляем поверхность, теряется также и контекст. Все не так плохо, как кажется, если при создании игры принять во внимание возможность потери контекста.

ПРИМЕЧАНИЕ

Вообще, это EGL отвечает за создание и удаление контекста и плоскости. EGL — это еще один стандарт Khronos Group. Он описывает, как работают пользовательский интерфейс операционной системы и OpenGL ES, а также как операционная система открывает OpenGL доступ к графическому оборудованию. Он также регламентирует создание плоскости и управление контекстом. Поскольку GLSurfaceView обрабатывает все, что касается EGL, мы можем спокойно игнорировать большинство ситуаций, связанных с EGL.

Следуя сложившейся традиции, напишем небольшой пример, который в каждом фрейме будет очищать экран и заполнять его каким-нибудь случайным цветом.

Листинг 7.1. GLSurfaceViewTest.java: Очистка экрана

```
package com.badlogic.androidgames.glbasics;

import java.util.Random;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.app.Activity;
import android.opengl.GLSurfaceView;
import android.opengl.GLSurfaceView.Renderer;
import android.os.Bundle;
import android.util.Log;
import android.view.Window;
import android.view.WindowManager;

public class GLSurfaceViewTest extends Activity {
    GLSurfaceView glView;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                WindowManager.LayoutParams.FLAG_FULLSCREEN);
        glView = new GLSurfaceView(this);
        glView.setRenderer(new SimpleRenderer());
        setContentView(glView);
    }
```

Мы сохраняем ссылку на экземпляр класса GLSurfaceView как на член этого класса. В методе onCreate() разворачиваем наше приложение на весь экран, создаем GLSurfaceView, реализуем Renderer и делаем GLSurfaceView просмотрщиком содержимого Activity.

```
@Override
public void onResume() {
    super.onPause();
    glView.onResume();
}

@Override
public void onPause() {
    super.onPause();
    glView.onPause();
}
```

В методах onResume() и onPause() вызываем суперметоды, а также соответствующие методы GLSurfaceView. Они начнут и прервут работу потока визуализации GLSurfaceView, который в свою очередь запустит методы обратного вызова реализации Renderer в соответствующее время.

```
    static class SimpleRenderer implements Renderer {
        Random rand = new Random();

        @Override
        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            Log.d("GLSurfaceViewTest", "surface created");
        }

        @Override
        public void onSurfaceChanged(GL10 gl, int width, int height) {
            Log.d("GLSurfaceViewTest", "surface changed: " + width + "x"
                    + height);
        }

        @Override
        public void onDrawFrame(GL10 gl) {
            gl.glClearColor(rand.nextFloat(), rand.nextFloat(),
                    rand.nextFloat(), 1);
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        }
    }
}
```

Последняя часть кода — это реализация нашего Renderer. Он просто регистрирует некоторую информацию из методов onSurfaceCreated() и onSurfaceChanged(). Однако особого внимания заслуживает метод onDrawFrame().

Как мы уже говорили, экземпляр класса `GL10` предоставляет доступ к API OpenGL ES. `10` в `GL10` означает, что он предоставляет все функции, описанные в стандарте OpenGL ES 1.0. Пока ограничимся этим замечанием. Все методы этого класса имеют соответствующую C-функцию, описанную в стандарте. Каждый метод начинается с префикса `gl`, что является старой доброй традицией OpenGL ES.

Первый метод OpenGL ES, который мы вызываем, — это `glClearColor()`. Вы, наверное, уже догадываетесь, что он делает. Он устанавливает цвет, который будет использоваться, когда мы дадим команду очистить экран. Цвета в OpenGL ES почти всегда представляются в формате RGBA, где каждый компонент имеет диапазон от 0 до 1. Существуют способы описать цвет в формате RGB565, но давайте пока будем пользоваться представлением в виде числа с плавающей точкой. Можно всего однажды установить любой цвет для заполнения экрана после очистки — и OpenGL ES запомнит этот цвет. Цвет, который мы задаем с помощью `glClearColor()`, является одним из состояний OpenGL ES.

Следующий вызов, собственно, очищает экран, заполняя его цветом, который мы выберем. Метод `glClear()` использует один аргумент, который определяет, какой буфер очистить. OpenGL ES включает в себя не только информацию о фреймбуфере, который содержит пикселы, но и о буферах остальных типов. Мы рассмотрим типы фреймбуферов более подробно в главе 10, а пока нас интересует только фреймбуфер, который содержит пикселы. В OpenGL ES он называется буфером цвета. Чтобы сообщить OpenGL ES, что мы хотим очистить данный конкретный буфер, мы указываем константу `GL10.GL_COLOR_BUFFER_BIT`.

OpenGL ES имеет множество констант, которые определены как статические общедоступные элементы интерфейса GL10. Как и методы, все константы имеют префикс `GL_`.

Это было наше первое приложение на OpenGL ES. С вашего позволения я не буду размещать здесь скриншот, так как вы, наверняка, знаете, как все должно выглядеть.

ПРИМЕЧАНИЕ

Не вызывай OpenGL ES из другого потока. Это первая и самая важная заповедь! Дело в том, что OpenGL ES создан для работы в однопоточной среде и нестабильно работает при применении нескольких потоков. Некоторые функции будут работать и при многопоточности, но со многими драйверами могут возникнуть проблемы, так что не стоит этого делать.

GLGame: реализация игрового интерфейса

В предыдущей главе мы реализовали класс `AndroidGame`, который связывает все подмодули для звука, файлов ввода-вывода, графики и обработки пользовательского ввода. Мы вновь используем большую часть этого кода в нашей будущей

двухмерной игре на OpenGL ES, поэтому создадим новый класс GLGame, который реализует игровой интерфейс, который мы описали ранее.

Вообще, мы пока знаем об OpenGL ES недостаточно, чтобы реализовать интерфейс Graphics. Однако здесь вас ждет сюрприз: мы не будем его реализовывать. OpenGL не очень хорошо работает с программными моделями интерфейса Graphics. Вместо этого мы создадим новый класс GLGraphics, отслеживающий экземпляр класса GL10, получаемый от GLSurfaceView (листинг 7.2).

Листинг 7.2. GLGraphics.java; отслеживание GLSurfaceView и экземпляра GL10

```
package com.badlogic.androidgames.framework.impl;

import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView;

public class GLGraphics {
    GLSurfaceView glView;
    GL10 gl;

    GLGraphics(GLSurfaceView glView) {
        this.glView = glView;
    }

    public GL10 getGL() {
        return gl;
    }

    void setGL(GL10 gl) {
        this.gl = gl;
    }

    public int getWidth() {
        return glView.getWidth();
    }

    public int getHeight() {
        return glView.getHeight();
    }
}
```

В этом классе всего несколько методов-установщиков и методов-получателей. Обратите внимание, что мы будем использовать данный класс для визуализации потока, запущенного GLSurfaceView. Таким образом, у нас может возникнуть проблема с вызовом метода View, который работает в основном в потоке пользовательского интерфейса. В данном случае это неважно, мы сможем запрашивать только ширину и высоту GLSurfaceView, чего нам вполне хватит.

Класс GLGame немного сложнее. Он использует большую часть кода из класса AndroidGame. Единственный момент, который вызывает сложности, — синхронизация

между визуализацией и потоками пользовательского интерфейса. Рассмотрим, как это происходит (листинг 7.3).

Листинг 7.3. GLGame.java, реализация OpenGL ES Game

```
package com.badlogic.androidgames.framework.impl;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.app.Activity;
import android.content.Context;
import android.opengl.GLSurfaceView;
import android.opengl.GLSurfaceView.Renderer;
import android.os.Bundle;
import android.os.PowerManager;
import android.os.PowerManager.WakeLock;
import android.view.Window;
import android.view.WindowManager;

import com.badlogic.androidgames.framework.Audio;
import com.badlogic.androidgames.framework.FileIO;
import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Graphics;

import com.badlogic.androidgames.framework.Input;
import com.badlogic.androidgames.framework.Screen;

public abstract class GLGame extends Activity implements Game, Renderer {
    enum GLGameState {
        Initialized,
        Running,
        Paused,
        Finished,
        Idle
    }

    GLSurfaceView glView;
    GLGraphics glGraphics;
    Audio audio;
    Input input;
    FileIO fileIO;
    Screen screen;
    GLGameState state = GLGameState.Initialized;
    Object stateChanged = new Object();
    long startTime = System.nanoTime();
    WakeLock wakeLock;
```

Этот класс дополняет класс `Activity` и реализует `Game` и интерфейс `GLSurfaceView.Renderer`. Он содержит перечень `GLGameState`, отслеживающий, в каком состоянии

на данный момент находится экземпляр класса GLGame. Мы вернемся к этому немного позже.

Члены данного класса состоят из экземпляров класса GLSurfaceView и GLGraphics. Класс также содержит экземпляры Audio, Input, FileIO и Screen, которые понадобятся нам для написания игры, так же как и в классе AndroidGame. Элемент state отслеживает состояние с помощью одного из перечней GLGameState. Элемент stateChanged представляет собой объект, который мы будем использовать для синхронизации потока пользовательского интерфейса и потока визуализации. Далее используем элемент, отслеживающий дельту времени, и WakeLock, который не дает экрану гаснуть.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                         WindowManager.LayoutParams.FLAG_FULLSCREEN);
    glView = new GLSurfaceView(this);
    glView.setRenderer(this);
    setContentView(glView);

    glGraphics = new GLGraphics(glView);
    fileIO = new AndroidFileIO(getAssets());
    audio = new AndroidAudio(this);
    input = new AndroidInput(this, glView, 1, 1);
    PowerManager powerManager = (PowerManager)
            getSystemService(Context.POWER_SERVICE);
    wakeLock = powerManager.newWakeLock(PowerManager.FULL_WAKE_LOCK,
            "GLGame");
}
```

В OnCreate() выполняется обычная подпрограмма установки. Мы развертываем Activity на весь экран и создаем экземпляр GLSurfaceView, устанавливая его как вид с содержимым. Мы также создаем экземпляры всех остальных классов, которые реализуют интерфейсы фреймворка, в частности классы AndroidFileIO или AndroidInput. Обратите внимание на то, что мы заново используем классы, которые применяли в классе AndroidGame, за исключением AndroidGraphics. Еще один важный момент заключается в том, что мы больше не разрешаем классу AndroidInput масштабировать координаты касания до целевого разрешения, как это было в AndroidGame. Оба значения масштаба равны 1, так что мы будем использовать реальные координаты касания. Зачем это нужно, мы разберемся чуть позже. Последнее, что нам предстоит сделать, — создать экземпляр класса WakeLock.

```
public void onResume() {
    super.onResume();
    glView.onResume();
    wakeLock.acquire();
}
```

В методе onResume() мы указываем классу GLSurfaceView запустить поток визуализации, вызвав его метод onResume(). Мы также создаем WakeLock.

```
@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    glGraphics.setGL(gl);

    synchronized(stateChanged) {
        if(state == GLGameState.Initialized)
            screen = getStartScreen();
        state = GLGameState.Running;
        screen.resume();
        startTime = System.nanoTime();
    }
}
```

Далее вызываем метод onSurfaceCreate(). Он, естественно, также вызывается в потоке визуализации. Здесь мы видим, как используются перечни состояний. Если приложение запущено впервые, то состояние будет иметь вид GLGameState.Initialized. В этом случае мы вызываем метод getStartScreen() для того, чтобы вернуться к стартовому экрану игры. Если игра находится в неинициализированном состоянии, но уже была запущена, мы знаем, что мы просто ее возобновили. В любом случае мы устанавливаем GLGameState.Running и вызываем текущий метод Screen — resume(). Мы также отслеживаем текущее время, чтобы позже можно было вычислить дельту времени.

Нам также необходима синхронизация, поскольку теми элементами, которыми мы управляем внутри блока синхронизации, также можно управлять в методе onPause() потока пользовательского интерфейса. Необходимо избежать этого, поэтому используем объект в качестве замка. Мы могли бы также применить экземпляр класса GLGame.

```
@Override
public void onSurfaceChanged(GL10 gl, int width, int height) {
}
```

Метод onSurfaceChanged() — это, в сущности, заглушка, поэтому не будем его разбирать.

```
@Override
public void onDrawFrame(GL10 gl) {
    GLGameState state = null;

    synchronized(stateChanged) {
        state = this.state;
    }

    if(state == GLGameState.Running) {
        float deltaTime = (System.nanoTime()-startTime) / 1000000000.0f;
        startTime = System.nanoTime();

        screen.update(deltaTime);
```

```
            screen.present(deltaTime);
        }

        if(state == GLGameState.Paused) {
            screen.pause();
            synchronized(stateChanged) {
                this.state = GLGameState.Idle;
                stateChanged.notifyAll();
            }
        }

        if(state == GLGameState.Finished) {
            screen.pause();
            screen.dispose();
            synchronized(stateChanged) {
            this.state = GLGameState.Idle;
            stateChanged.notifyAll();
            }
        }
    }
```

В методе `onDrawFrame()` делается большая часть работы. Он вызывается потоком визуализации максимально часто. В нем проверяем состояние нашей игры на данный момент и реагируем в соответствии с этим. Поскольку состояние может быть изменено в методе `onPause` в потоке пользовательского интерфейса, необходимо синхронизировать доступ к нему.

Если игра запущена, подсчитываем дельту времени и указываем текущему `Screen` обновиться.

Если игра поставлена на паузу, указываем текущему `Screen` также встать на паузу. Затем мы изменяем состояние на `GLGameState.Idle`, обозначая, что мы получили запрос на приостановку от потока пользовательского интерфейса. Поскольку мы ожидаем, что это произойдет в методе `onPause()` пользовательского интерфейса, мы сообщаем потоку пользовательского интерфейса, что он теперь может поставить приложение на паузу. Данное уведомление необходимо, поскольку нам нужно убедиться, что поток визуализации поставлен на паузу/закрыт в случае, если наша `Activity` закрыта или поставлена на паузу в потоке пользовательского интерфейса.

Если `Activity` завершается (и не поставлена на паузу), переходим к `GLGameState.Finished`. В этом случае указываем текущему `Screen` остановиться и удалить себя, а затем отослать еще одно уведомление потоку пользовательского интерфейса, ожидающему, пока поток визуализации завершит работу.

```
@Override
public void onPause() {
    synchronized(stateChanged) {
        if(isFinishing())
            state = GLGameState.Finished;
        else
            state = GLGameState.Paused;
```

```
            while(true) {
                try {
                    stateChanged.wait();
                    break;
                } catch(InterruptedException e) {
                }
            }
        }
        wakeLock.release();
        glView.onPause();
        super.onPause();
    }
```

Метод `onPause()` — обычный метод уведомления для работы с `Activity`, вызываемый потоком пользовательского интерфейса, когда `Activity` ставится на паузу. В зависимости от того, закрыто приложение или поставлено на паузу, устанавливаем соответствующее состояние и ждем, пока поток визуализации обработает это новое состояние. Для этого используется стандартный механизм Java ожидание/уведомление.

Наконец, высвобождаем `WakeLock` и указываем `GLSurfaceView` и `Activity` встать на паузу, остановить поток визуализации и удалить поверхность OpenGL ES, из-за чего инициируется потеря контекста OpenGL ES, о которой мы говорили ранее.

```
    public GLGraphics getGLGraphics() {
        return glGraphics;
    }
```

Метод `getGLGraphics()` — это новый метод, доступный только через класс `GLGame`. Он возвращает экземпляр класса `GLGraphics`, который мы сохраняем, чтобы позже у нас был доступ к интерфейсу `GL10` при дальнейшей реализации `Screen`.

```
    @Override
    public Input getInput() {
        return input;
    }

    @Override
    public FileIO getFileIO() {
        return fileIO;
    }

    @Override
    public Graphics getGraphics() {
        throw new IllegalStateException("We are using OpenGL!");
    }

    @Override
    public Audio getAudio() {
        return audio;
    }

    @Override
```

```
    public void setScreen(Screen screen) {
        if (screen == null)
            throw new IllegalArgumentException("Screen must not be null");

        this.screen.pause();
        this.screen.dispose();
        screen.resume();
        screen.update(0);
        this.screen = screen;
    }

    @Override
    public Screen getCurrentScreen() {
        return screen;
    }
}
```

Остаток данного класса функционально не изменился. Если мы случайно попытаемся получить доступ к стандартному экземпляру класса `Graphics`, получим исключение, поскольку `GLGame` не поддерживает такую операцию. Вместо этого будем работать с методом `GLGraphics`, который получаем через метод `GLGame.getGLGraphics()`.

Зачем нам нужна вся эта сложнейшая синхронизация с потоком визуализации? Так мы добьемся того, что наши реализации `Screen` будут полностью работать в потоке визуализации. Все методы `Screen` будут выполняться здесь, что необходимо, если мы хотим получить доступ к функционалу OpenGL ES. Запомните, мы можем получить доступ к OpenGL ES только в потоке визуализации.

Подведем итоги с помощью примера. В листинге 7.4 показано, как выглядит наш первый пример из этой главы при использовании `GLGame` и `Screen`.

Листинг 7.4. `GLGameTest.java`; больше очистки экрана, теперь на 100 % больше `GLGame`

```
package com.badlogic.androidgames.glbasics;

import java.util.Random;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Screen;
import com.badlogic.androidgames.framework.impl.GLGame;
import com.badlogic.androidgames.framework.impl.GLGraphics;

public class GLGameTest extends GLGame {
    @Override
    public Screen getStartScreen() {
        return new TestScreen(this);
    }

    class TestScreen extends Screen {
        GLGraphics glGraphics;
```

```
        Random rand = new Random();

        public TestScreen(Game game) {
            super(game);
            glGraphics = ((GLGame) game).getGLGraphics();
        }

        @Override
        public void present(float deltaTime) {
            GL10 gl = glGraphics.getGL();
            gl.glClearColor(rand.nextFloat(), rand.nextFloat(),
                    rand.nextFloat(), 1);
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        }

        @Override
        public void update(float deltaTime) {
        }

        @Override
        public void pause() {
        }

        @Override
        public void resume() {
        }

        @Override
        public void dispose() {
        }
    }
}
```

Это та же самая программа, что и в нашем предыдущем примере, за исключением того, что теперь мы наследуем от `GLGame`, а не от `Activity` и предоставляем реализацию `Screen` вместо реализации `GLSurfaceView.Renderer`.

В следующих примерах мы просто рассмотрим соответствующие части реализации `Screen` каждого примера. Общая структура наших примеров останется той же. Конечно, нам необходимо добавить пример реализации `GLGame` к нашей стартовой `Activity` и файлу манифеста.

С учетом всего этого визуализируем наш первый треугольник.

Мама, смотри: я нарисовал красный треугольник!

Итак, в OpenGL ES нужно кое-что настроить перед тем, как переходить к отрисовке геометрических фигур. Два наиболее важных элемента — это проекционная матрица (плюс конус отображения) и область просмотра, которая определяет размер конечного изображения и его положение при визуализации в фреймбуфере.

Определение области просмотра

OpenGL ES использует область просмотра для перевода координат, проецируемых на ближнюю плоскость отсечения, в пиксельные координаты фреймбуфера. Мы можем указать OpenGL ES использовать наш фреймбуфер частично или полностью с помощью следующего метода:

```
GL10.glViewport(int x, int y, int width, int height)
```

Координаты `x` и `y` задают верхний левый угол области просмотра в фреймбуфере, а `width` и `height` задают размер области просмотра в пикселах. Обратите внимание, что OpenGL ES ставит начало координат фреймбуфера в левом нижнем углу экрана. Обычно мы присваиваем началу координат значение ноль, а параметры `width` и `height` делаем равными высоте и ширине экрана, поскольку работаем в полноэкранном режиме. Можно было бы указать OpenGL ES использовать при таком подходе только часть фреймбуфера. Тогда отображаемая графика была бы автоматически вписана в нужные размеры.

ПРИМЕЧАНИЕ

Хотя и кажется, что данный метод устанавливает двухмерную систему координат для визуализации, на самом деле это не так. Он просто описывает часть фреймбуфера, которую использует OpenGL ES для вывода конечного изображения. Наша система координат описывается с помощью матриц проекции и матриц модели представления.

Определение матрицы проекции

Далее нам необходимо определить матрицу проекции. Поскольку в этой главе нас интересует только 2D-графика, мы будем использовать параллельную проекцию. Как это сделать?

Режимы матрицы и активные матрицы

Мы уже знаем, что OpenGL ES отслеживает три матрицы: матрицу проекции, матрицу модели представления и матрицу текстуры (которую мы продолжаем игнорировать). OpenGL ES предлагает нам несколько особых методов для изменения этих матриц. Перед тем как мы сможем использовать матрицы, нам необходимо сообщить OpenGL ES, какой матрицей мы хотим управлять. Это делается с помощью следующего метода:

```
GL10.glMatrixMode(int mode)
```

Параметр `mode` может быть выражен через `GL10.GL_PROJECTION`, `GL10.GL_MODELVIEW` или `GL10.GL_TEXTURE`. Наверное, и так понятно, какая из этих констант активизирует каждую матрицу. Все последующие вызовы методов управления матрицами будут направлены на матрицу, которую мы установили в этом методе, до тех пор, как снова не изменим активную матрицу, еще раз вызвав этот метод. Режим матрицы является одним из состояний OpenGL ES, которое также будет утеряно в случае потери контекста, (происходящей, если приложение было поставлено

на паузу и возобновлено). Для управления матрицей проекции и всех последующих вызовов мы можем вызвать метод следующим образом:

```
gl.glMatrixMode(GL10.GL_PROJECTION);
```

Ортогональная проекция с применением glOrthof

OpenGL ES предлагает следующий метод для установки в активной матрице ортографической проекции:

```
GL10.glOrthof(int left, int right, int bottom, int top, int near, int far)
```

Напоминает манипуляции, которые мы производили с плоскостями отсечения нашего конуса отображения. И действительно, сходство неслучайное. Какие же значения мы здесь определяем?

OpenGL ES имеет стандартную систему координат, как на рис. 7.5. Положительные точки по оси *x* находятся справа, положительные точки оси *y* — вверху, а положительные точки оси *z* направлены в нашу сторону. С помощью `glOrthof()` мы описываем конус отображения нашей параллельной проекции в данной системе координат. Если снова взглянуть на рис. 7.3, то мы увидим, что конус в параллельной проекции имеет форму куба. Мы можем рассматривать параметры для `glOrthof()` как определяющие два угла нашего конуса-куба (рис. 7.5).

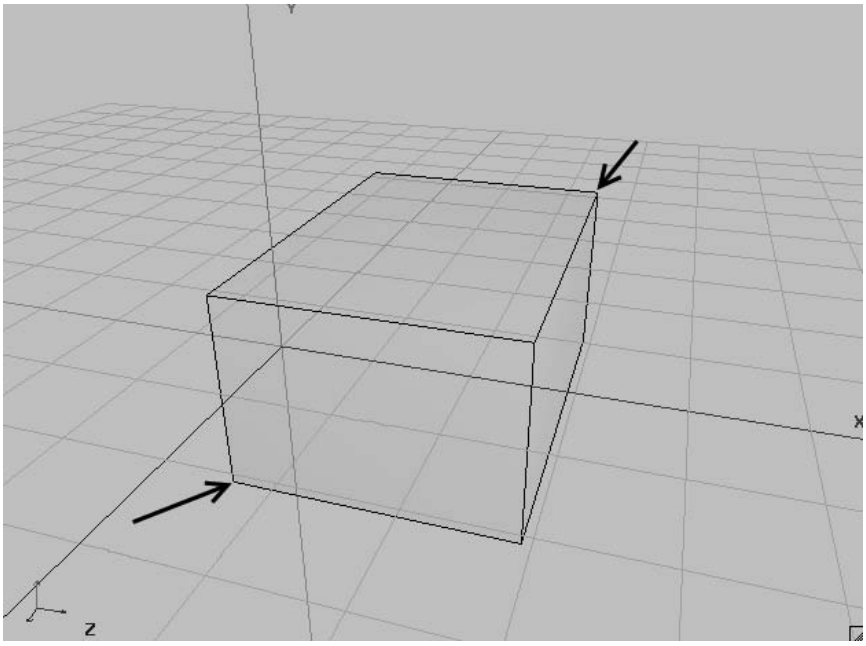

Рис. 7.5. Ортогональный конус отображения

Передняя сторона нашего конуса отображения непосредственно накладывается на область просмотра. В случае если мы работаем с полноэкранной областью про-

смотра, расположенной в диапазоне от (0; 0) до (480; 320) (например, альбомная ориентация на HTC Hero), нижний левый угол передней стороны будет находиться в нижнем левом углу экрана, а правый верхний угол передней стороны — в верхнем левом углу экрана. OpenGL выполнит растяжение автоматически.

Поскольку нас интересует 2D-графика, мы определим угловые точки (левая; нижняя; ближайшая) и (правая; верхняя; дальняя) (см. рис. 7.5) таким образом, чтобы можно было работать в своеобразной пиксельной системе координат, как мы это делали с `Canvas` и мистером Номом. Вот как создать подобную систему координат:

```
gl.glOrthof(0, 480, 0, 320, 1, -1);
```

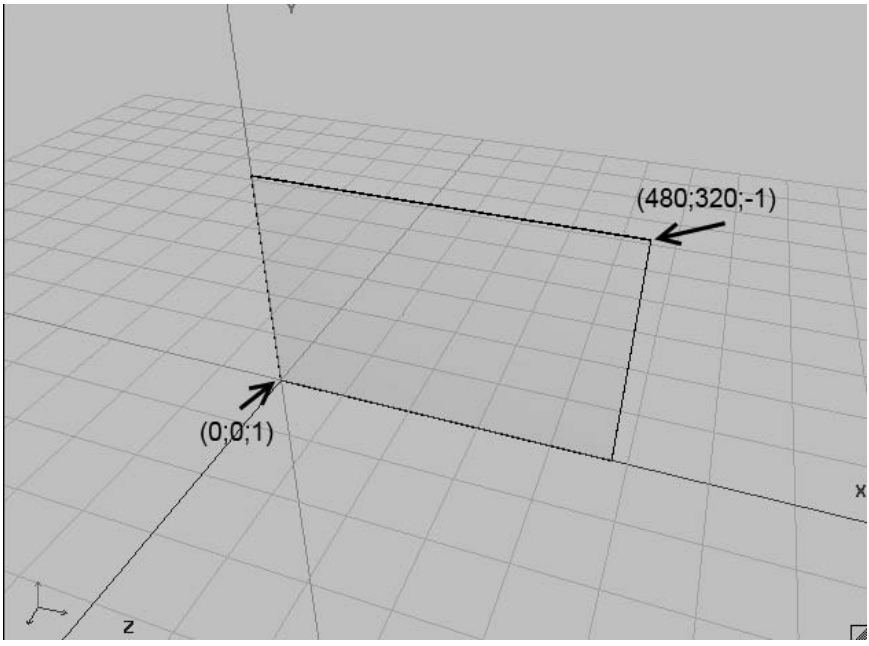

Рис. 7.6. Параллельная проекция конуса отображения для визуализации 2D с помощью OpenGL ES

Наш конус отображения достаточно тонкий, однако это нормально, поскольку мы работаем в 2D. Видимая часть системы координат находится в диапазоне от (0; 0; 1) к (480; 320; –1). Любые точки, которые мы описываем внутри данной зоны, будут также видимы на экране. Эти точки будут проецироваться на переднюю часть куба, которая является ближней плоскостью отсечения. Затем проекция будет растянута до размера области просмотра независимо от ее размеров. Допустим, у нас есть Nexus One с разрешением 800 × 480 пикселов в альбомной ориентации. Когда мы описываем конус отображения так, как только что было показано, мы можем работать в системе координат 480 × 320, а OpenGL растянет изображение до фреймбуфера 800 × 480 (если мы решили, что область просмотра должна

охватывать весь фреймбуфер). Кстати, ничто нам не мешает использовать и конусы отображения более странной формы. Например, мы могли бы взять конус, в котором углы имеют координаты (–1; –1; 100) и (2; 2; –100). Все, что попадает внутрь данной области, будет в итоге растянуто до необходимого размера и показано на экране. Достаточно удобно.

Обратите внимание, что мы также описываем ближнюю и дальнюю плоскости отсечения. Поскольку в данной главе мы собираемся полностью игнорировать ось *z*, возможно, захочется ввести 0 для обеих плоскостей. Однако этого не стоит делать по нескольким причинам. Чтобы не рисковать, мы предоставляем конусу отображения небольшой буфер в оси *z*. Все геометрические точки будут описаны на оси *x*, где *z* равна нулю, что, в принципе, и есть 2D.

ПРИМЕЧАНИЕ

Возможно, вы заметили, что ось y теперь направлена вверх, а ее начало находится в нижнем левом углу экрана. Несмотря на то что Canvas, фреймворк пользовательского интерфейса и много других API, визуализирующих 2D, используют систему, где ось y начинается в верхнем левом углу и идет вниз, для программирования игр гораздо удобнее применять данную новую систему координат. Например, если Марио прыгает, разве вы не ожидаете, что его y-координата будет возрастать вместо того, чтобы уменьшаться? Хотите работать в другой системе координат? Хорошо, просто поменяйте местами верхний и нижний параметры glOrthof(). Также, хотя изображение конуса отображения выглядит правильно с геометрической точки зрения, передняя и дальняя плоскости отсечения выглядят для glOrthof() несколько по-другому. Поскольку это немного сложно, давайте просто притворимся, что предыдущие иллюстрации выглядят правильно.

Полезный фрагмент

Вот небольшой фрагмент кода, который будет использоваться во всех наших примерах из этой главы. Он очищает экран, заполняя его черным цветом, задает область просмотра размером с весь фреймбуфер и устанавливает матрицу проекции (и следовательно, конус отображения), чтобы мы могли работать с удобной системой координат, начинающейся в нижнем левом углу экрана, и с осью *y*, которая направлена вверх.

```
gl.glClearColor(0,0,0,1);
gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
gl.glMatrixMode(GL10.GL_PROJECTION);
gl.glLoadIdentity();
gl.glOrthof(0, 320, 0, 480, 1, -1);
```

Подождите, а что же делает здесь `glLoadIdentity()`? Дело в том, что большинство методов, которые предлагает OpenGL ES для управления активной матрицей, на практике не задают матрицу. Вместо этого они создают временную матрицу с необходимыми параметрами и умножают ее на текущую матрицу. Метод `glOrthof()` — не исключение. Например, если бы мы в каждом кадре вызывали `glOrthof()`, мы умножали бы матрицу проекции очень много раз. Так что вместо этого перед тем, как умножать матрицу проекции, мы лучше проверим, есть ли у нас в запасе чистая единичная матрица. Как вы помните, умножение единичной матрицы на другую матрицу не изменяет эту неединичную матрицу. Вот зачем нам понадобится

glLoadIdentity(). Считайте, что мы сначала загружаем значение 1, а затем умножаем его на нужную матрицу (в нашем случае это матрица проекции, созданная glOrthof()).

Обратите внимание, что система координат теперь находится в диапазоне от (0; 0; 1) до (320; 480; –1) — это для визуализации в книжной ориентации.

Определение треугольников

Далее необходимо выяснить, как сообщить OpenGL ES о треугольниках, которые мы хотим визуализировать. Определим для начала, из чего состоит треугольник:

- треугольник состоит из трех точек;
- каждая точка называется вершиной;
- вершина имеет положение в 3D-пространстве;
- расположение в 3D-пространстве указано в виде трех чисел с плавающей точкой, определяющих *x*-, *y*- и *z*-координаты;
- вершина может иметь такие дополнительные атрибуты, как цвет и текстурные координаты (мы поговорим об этом чуть позже). Эта информация также может выражаться в числах с плавающей точкой.

OpenGL ES будет отсылать описания наших треугольников в виде массивов. Тем не менее, если учитывать, что OpenGL ES фактически является API на языке C, мы не можем использовать стандартные массивы Java. Вместо этого мы будем применять буферы Java NIO, которые являются обычными блоками памяти из последовательных байтов.

Небольшое лирическое отступление о буфере NIO (нового ввода-вывода)

Если быть абсолютно точным, нам необходимы *прямые* буферы NIO. Это значит, что их память находится не в куче виртуальной машины, а в нативной куче. Для создания прямого буфера NIO подойдет следующий фрагмент кода:

```
ByteBuffer buffer = ByteBuffer.allocateDirect(NUMBER_OF_BYTES); buffer.
order(ByteOrder.nativeOrder());
```

Здесь выделяется ByteBuffer, который может содержать NUMBER_OF_BYTES (общее количество байт) и проверяет, соответствует ли порядок байт используемому в центральном процессоре. Буфер NIO имеет три свойства:

- capacity — максимальное количество элементов, которое может содержать буфер;
- position — текущее местоположение, куда будет записан следующий элемент или откуда он будет прочтен;
- limit — индекс последнего определенного элемента плюс один.

Фактически вместимость буфера — это его размер. В случае ByteBuffer этот размер измеряется в байтах. Свойства position и limit можно считать определяющими

сегмент внутри буфера, который начинается с position и заканчивается на limit (исключающе).

Поскольку мы хотим определить наши вершины как числа с плавающей точкой, хорошо бы справиться с этими байтами. К счастью, мы можем преобразовать экземпляр класса ByteBuffer в экземпляр класса FloatBuffer, который позволяет нам работать с числами с плавающей точкой.

```
FloatBuffer floatBuffer = buffer.asFloatBuffer();
```

В случае с FloatBuffer свойства capacity, position и limit являются числами с плавающей точкой. Наш паттерн использования данных буферов достаточно прост. Он выглядит следующим образом:

```
float[] vertices = { ... определения положений вершин и т.д. ...; floatBuffer.
clear();
floatBuffer.put(vertices);
floatBuffer.flip();
```

Сначала определяем наши данные в стандартном массиве Java из чисел с плавающей точкой. Перед тем как вставить массив float в буфер, указываем буферу очистить себя с помощью метода clear(). Фактически так не стираются никакие данные, просто position получает значение 0, а limit становится равным capacity. Далее используем метод FloatBuffer.put(float[] array), чтобы скопировать содержимое всего массива в буфер, начиная с текущей позиции в буфере. После копирования эта позиция увеличится на длину массива. Далее вызываем метод put(), который добавляет дополнительные данные к данным предыдущего массива, который мы скопировали в буфер. Последний вызов FloatBuffer.flip() просто меняет местами capacity и limit.

Чтобы лучше понять, как это работает, предположим, что наш массив вершин имеет размер в 5 чисел с плавающей точкой и что capacity нашего FloatBuffer достаточно велика, чтобы вместить в себя эти 5 чисел. После вызова FloatBuffer.put() позиция буфера будет равна 5 (индексы от 0 до 4 заняты 5 числами с плавающей точкой из нашего массива), limit по-прежнему будет равен вместимости буфера. После вызова FloatBuffer.flip() свойству position будет задано значение 0, а limit будет установлен в 5. Любая сторона, которая попробует прочесть данные из буфера, будет знать, что ей следует читать числа с плавающей точкой из индекса 0–4 (включительно). Об этом необходимо знать и OpenGL ES. Обратите внимание, что OpenGL ES тем не менее не учитывает limit. Обычно мы сообщаем количество элементов, которое необходимо прочесть, делаем это вместе с передачей буфера. На данном этапе не проверяются ошибки, так что будьте внимательны.

Иногда лучше установить position буфера вручную после того, как мы его заполнили. Это можно сделать с помощью следующего метода:

```
FloatBuffer.position(int position)
```

Это пригодится нам позже, когда мы временно установим свойству position заполненного буфера какое-либо значение, отличное от нуля, для того чтобы OpenGL ES начал считывание данных буфера с конкретного адреса.

Отправка вершин к OpenGL ES

Как мы опишем местоположение трех вершин нашего первого треугольника? Легко — предположим, что наша система координат лежит в диапазоне от (0; 0; 1) до (320; 480; –1). Тогда мы можем сделать следующее:

```
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(3 * 2 * 4);
byteBuffer.order(ByteOrder.nativeOrder());
FloatBuffer vertices = byteBuffer.asFloatBuffer();
vertices.put(new float[] {   0.0f,   0.0f,
                           319.0f,   0.0f,
                           160.0f, 479.0f });
vertices.flip();
```

Первые три строки должны быть вам уже знакомы. Единственная интересная часть касается того, сколько байт мы выделяем. У нас есть три вершины, каждая из которых имеет положение, определяемое координатами по осям *x* и *y*. Каждая координата — это число с плавающей точкой (`float`), которое занимает 4 байта. Все вершины имеют две координаты, каждая из которых занимает 4 байта, так что в целом для нашего треугольника необходимо 24 байта.

ПРИМЕЧАНИЕ

Мы можем определить вершины только с помощью координат x и y, а OpenGL ES автоматически задаст z-координате значение ноль.

Далее копируем в буфер массив чисел типа `float`, содержащий данные о вершинах. Наш треугольник начинается в нижнем левом углу (0; 0), далее к правой границе видимой области/экрана (319; 0), а затем к середине верхней границы видимой области/экрана. Поскольку мы уже умеем пользоваться буфером NIO, мы также применяем метод `flip()` к нашему буферу. Таким образом, `position` будет 0, а `limit` будет 6 (помните, что значения свойств `limit` и `position` `FloatBuffer` задаются в числах с плавающей точкой (`float`), а не в байтах).

Как только наш буфер NIO готов, мы можем указать OpenGL ES отрисовать его в текущем состоянии (например, с областью просмотра и матрицей проекции). Это можно сделать с помощью следующего фрагмента:

```
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
gl.glVertexPointer( 2, GL10.GL_FLOAT, 0, vertices);
gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
```

Вызов метода `glEnableClientState()` можно назвать рудиментарным. Так мы сообщаем OpenGL ES, что вершины, которые мы собираемся нарисовать, имеют позиции. Это немного глупо по двум следующим причинам:

- константа называется `GL10.GL_VERTEX_ARRAY`, несколько путано. Было бы разумнее назвать ее `GL10.GL_POSITION_ARRAY`;
- невозможно нарисовать то, у чего нет позиции, так что использовать данный метод несколько излишне. Мы тем не менее применим его для удобства работы с OpenGL ES.

Обращаясь к `glVertexPointer()`, мы сообщаем OpenGL ES, где можно найти позиции вершин, а также некоторую другую дополнительную информацию. Первый параметр сообщает OpenGL ES, что каждая позиция вершины состоит из двух координат, *x* и *y*. Если бы вы ввели *x*, *y* и *z*, координат было бы три. Второй параметр сообщает OpenGL ES тип данных, в которых хранится каждая координата. В нашем случае это `GL10.GL_FLOAT`, который показывает, что мы использовали `float` по 4 байта каждый. Третий параметр, `stride` (шаг), сообщает OpenGL, насколько далеко в байтах вершины расположены друг от друга. В предыдущем случае шаг был равен нулю, так как позиции располагались очень близко друг к другу (вершина 1 (*x*; *y*), вершина 2 (*x*; *y*) и т. д.). Последним параметром является `FloatBuffer`, касательно которого следует запомнить две вещи:

- `FloatBuffer` является блоком памяти в нативной куче, так что он имеет начальный адрес;
- позиция `FloatBuffer` — смещение относительно начального адреса.

OpenGL ES берет адрес начала буфера и прибавляет к нему позицию буфера, чтобы попасть к тому числу с плавающей точкой, с которого начнется считывание вершин, когда мы укажем выполнить отрисовку содержимого буфера. Указатель на вершины (который опять-таки правильнее называть указателем на местоположение) — это состояние OpenGL ES. Пока мы не изменим его (и контекст не потеряется), OpenGL ES будет помнить и использовать его для всех последующих запросов, требующих позицию вершин.

И наконец, рассмотрим вызов `glDrawArrays()`. Этот метод отрисует наш треугольник. Первый параметр определяет, элементарную фигуру какого типа мы собираемся рисовать. В данном случае мы хотим нарисовать несколько треугольников, определяемых `GL10.GL_TRIANGLES`. Следующий параметр — это смещение относительно первой вершины на обозначенную в указателе вершину. Данное смещение измеряется в вершинах, а не байтах или числах с плавающей точкой. Если бы мы описывали более чем один треугольник, мы могли бы использовать этот фрагмент, чтобы визуализировать лишь часть списка треугольников. Наш последний аргумент сообщает OpenGL ES, сколько вершин нужно использовать для визуализации. В нашем случае это три вершины. Обратите внимание, что нам всегда необходимо определить количество, кратное трем, если мы рисуем `GL10.GL_TRIANGLES`. Каждый треугольник состоит из трех вершин, так что это целесообразно. Для других простейших фигур правила несколько отличаются.

Когда мы запускаем команду `glVertexPointer()`, OpenGL ES передает месторасположения вершины графическому процессору и сохраняет их для всех последующих команд визуализации. Каждый раз, когда мы указываем OpenGL ES визуализировать вершины, их местоположение узнается из данных, которые мы определили с помощью `glVertexPointer()`.

Каждая наша вершина может иметь и другие свойства кроме местоположения. Еще одним свойством может быть цвет вершины. Обычно эти свойства называются свойствами вершины.

Может возникнуть вопрос: как **OpenGL ES узнает, какого цвета сделать треугольник**, если мы определили только местоположение. Дело в том, что OpenGL ES содержит настройки по умолчанию для свойств вершины, которые мы сами не указали. Большинство данных свойств можно настроить напрямую. Например, чтобы установить цвет для всех вершин, которые рисуем, можно использовать следующий метод:

```
GL10.glColor4f(float r, float g, float b, float a)
```

Этот метод задаст цвет по умолчанию, используемый для всех вершин, для которых мы не указали цвет отдельно. Данный цвет указывается в значениях RGBA в диапазоне от 0,0 до 1,0 — такой же диапазон мы уже применяли при работе с цветом выше. Цвет по умолчанию, с которого начинает OpenGL ES, — это (1; 1; 1; 1), матовый белый.

Вот и весь код, который нам нужен для визуализации треугольника в параллельной проекции с помощью OpenGL ES. Эти 16 строк кода необходимы нам для очистки экрана, установки области просмотра и матрицы проекции, создания буфера NIO, в котором мы храним информацию о положении вершин, и для отрисовки треугольника. А теперь сравните это количество кода с несколькими страницами, которые ушли у меня на то, чтобы объяснить все это. Конечно, я мог бы опустить некоторые детали и выражаться попроще. Проблема в том, что OpenGL ES местами довольно сложен, и для того, чтобы не получить после работы пустой экран, лучше сначала изучить все подробно, а не просто копировать и вставлять код.

Все вместе

Чтобы подвести итоги этого раздела, объединим все с помощью реализации `GLGame` и `Screen`. В листинге 7.5 показан пример того, как это сделать.

Листинг 7.5. `FirstTriangleTest.java`

```
package com.badlogic.androidgames.glbasics;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Screen;
import com.badlogic.androidgames.framework.impl.GLGame;
import com.badlogic.androidgames.framework.impl.GLGraphics;

public class FirstTriangleTest extends GLGame {
    @Override
    public Screen getStartScreen() {
        return new FirstTriangleScreen(this);
    }
```

Класс FirstTriangleTest наследует от GLGame и, соответственно, реализует метод Game.getStartScreen(). В этом методе мы создаем новый класс FirstTriangleScreen, который впоследствии будет вызываться GLGame, обновляться и отображаться. Обратите внимание, что, когда вызывается этот метод, мы уже находимся в основном цикле или даже скорее в потоке визуализации GLSurfaceView, так что мы можем использовать методы OpenGL ES в конструкторе класса FirstTriangleScreen. Рассмотрим реализацию класса Screen подробнее:

```
class FirstTriangleScreen extends Screen {
    GLGraphics glGraphics;
    FloatBuffer vertices;

    public FirstTriangleScreen(Game game) {
        super(game);
        glGraphics = ((GLGame)game).getGLGraphics();

        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(3 * 2 * 4);
        byteBuffer.order(ByteOrder.nativeOrder());
        vertices = byteBuffer.asFloatBuffer();
        vertices.put( new float[] {   0.0f,   0.0f,
                                    319.0f,   0.0f,
                                    160.0f, 479.0f});
        vertices.flip();
    }
```

Класс FirstTriangleScreen содержит два члена: экземпляр класса GLGraphics и наш проверенный буфер FloatBuffer, который хранит 2D-положения трех вершин нашего треугольника. В конструкторе переносим экземпляр класса GLGraphics из GLGame и создаем FloatBuffer в соответствии с нашим предыдущим фрагментом кода. Поскольку конструктор Screen получает экземпляр класса Game, нам необходимо привести его к классу GLGame, чтобы иметь возможность вызвать метод GLGame.getGLGraphics().

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glClear(GL10. GL_COLOR_BUFFER_BIT);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    gl.glOrthof(0, 320, 0, 480, 1, -1);

    gl.glColor4f(1, 0, 0, 1);
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glVertexPointer( 2, GL10.GL_FLOAT, 0, vertices);
    gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
}
```

Метод present() отражает то, что мы с вами только что обсудили: мы устанавливаем область просмотра, очищаем экран, устанавливаем матрицу проекции, чтобы можно было работать в измененной системе координат, задаем цвет по умолчанию для вершин (в нашем случае красный), указываем, что у наших вершин

будут определенные положения, сообщаем OpenGL ES, где находится информация об этих положениях вершин, и, наконец, отображаем наш маленький красный треугольник.

```
        @Override
        public void update(float deltaTime) {
            game.getInput().getTouchEvents();
            game.getInput().getKeyEvents();
        }

        @Override
        public void pause() {

        }

        @Override
        public void resume() {

        }

        @Override
        public void dispose() {

        }
    }
}
```

Оставшаяся часть кода — это просто шаблон. В методе `update()` мы проверяем, не заполнился ли наш буфер событий. Оставшаяся часть кода вообще ничего не делает.

ПРИМЕЧАНИЕ

С этого момента мы сосредоточимся только на самих классах Screen, так как заключающие их производные классы GLGame, такие как FirstTriangleTest, не изменятся. Мы также немного сокращаем код, убирая все ненужные методы класса Screen. Следующие примеры будут различаться только названиями элементов, конструкторов и методов презентации.

На рис. 7.7 показано, какое изображение получилось в результате предыдущего примера.

Рассмотрим, что же мы сделали не так в этом примере с точки зрения оптимальных методов работы с OpenGL ES.

- Мы устанавливали одни и те же состояния одним и тем же значениям снова и снова без какой-либо на то нужды. Изменения состояний в OpenGL ES занимают много памяти — некоторые немного больше, некоторые немного меньше. Мы всегда должны стремиться сократить количество изменений состояния, которые мы делаем за один кадр.
- Точка наблюдения и матрица проекции никогда не изменяются с момента, как мы их установили. Мы можем переместить этот код в метод `resume()`, который вызывается всего один раз, когда создается (или создается заново) поверхность OpenGL ES. Здесь же мы справляемся с потерей контекста OpenGL ES.

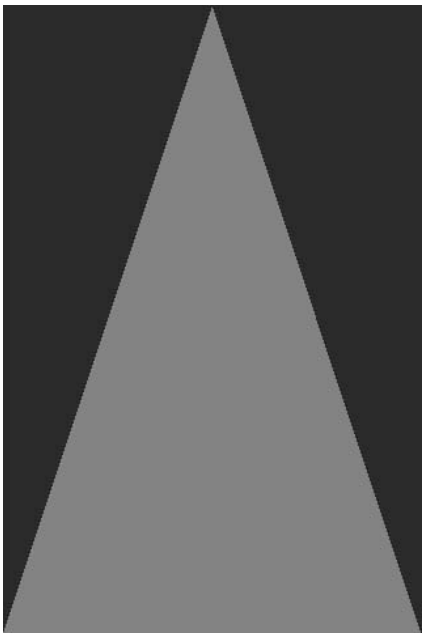

Рис. 7.7. Наш первый фрейдистский треугольник

- Мы можем также переместить настройку цвета для очистки экрана и выполняемую по умолчанию установку цвета вершины в метод resume(). Эти два цвета также не будут изменяться.
- Мы можем переместить методы `glEnableClientState()` и `glVertexPointer()` в метод `resume()`.
- Единственное, что надо вызывать в каждом кадре, — это методы `glClear()` и `glDrawArrays()`. Они оба используют текущие состояния OpenGL ES, которые остаются нетронутыми до тех пор, пока мы не изменим их или не случится потеря контекста из-за того, что `Activity` была приостановлена и запущена вновь.

Если выполнить все эти оптимизации, у нас будет только два вызова OpenGL ES в основном цикле.

Чтобы окончательно не запутаться, мы пока не будем использовать данные оптимизации. Однако когда мы перейдем к нашей первой игре на OpenGL ES, нам придется придерживаться данных рекомендаций как можно строже.

Добавим еще несколько атрибутов к вершинам нашего треугольника. Для начала займемся цветом.

ПРИМЕЧАНИЕ

Наблюдательные читатели, наверное, уже заметили, что у треугольника на рис. 7.7 не хватает пиксела в нижнем правом углу. Возможно, это выглядит, как типичная ошибка занижения на единицу, но на самом деле, это происходит из-за того, что OpenGL ES растрирует (рисует пикселы) треугольника. Существует особое правило растрирования треугольника, которое вызывает подобный огрех. Однако не стоит переживать. Нас с вами волнует визуализация двухмерных прямоугольников (состоящих из двух треугольников), где этот эффект нивелируется.

Определение цвета вершины

В предыдущем примере мы установили глобальный цвет по умолчанию для всех вершин, которые мы рисуем с помощью `glColor4f()`. Иногда нам требуется более тонкий контроль (например, возможность устанавливать цвет для каждой из вершин). OpenGL ES предлагает нам данный функционал, который по-настоящему легок в использовании. Все, что нам надо сделать, — добавить `float`-компоненты RGBA к каждой вершине и сообщить OpenGL ES, где можно найти цвет для всех вершин так же, как мы сообщали, где можно найти расположение каждой вершины. Начнем с того, что присвоим цвет каждой вершине:

```
int VERTEX_SIZE = (2 + 4) * 4;
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(3 * VERTEX_SIZE);
byteBuffer.order(ByteOrder.nativeOrder());
FloatBuffer vertices = byteBuffer.asFloatBuffer();
vertices.put( new float[] {  0.0f,   0.0f, 1, 0, 0, 1,
                           319.0f,   0.0f, 0, 1, 0, 1,
                           160.0f, 479.0f, 0, 0, 1, 1});
vertices.flip();
```

Сначала необходимо выделить `ByteBuffer` для наших трех вершин. Какой размер должен иметь этот `ByteBuffer`? У нас есть две координаты и четыре (RGBA) компонента цвета для каждой вершины, что в сумме составляет 6 чисел с плавающей точкой. Каждое значение `float` занимает 4 байта, так что каждая вершина требует 24 байта. Мы сохраняем данную информацию в `VERTEX_SIZE`. Когда мы вызываем `ByteBuffer.allocateDirect()`, мы просто умножаем `VERTEX_SIZE` на количество вершин, которое хотим сохранить в `ByteBuffer`. Оставшаяся часть понятна сама по себе. Мы получаем вид `FloatBuffer` для нашего `ByteBuffer` и помещаем вершины методом `put()` в `ByteBuffer`. Каждый ряд массива `float` содержит *x*- и *y*-координаты, а также R-, G-, B- и A-компоненты вершины именно в таком порядке.

Если мы хотим визуализировать это, нам необходимо сообщить OpenGL ES, что наши вершины не только имеют местоположение, но также и свойство цвета. Мы это делаем так же, как и раньше, с помощью вызова `glEnableClientState()`:

```
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY); gl.glEnableClientState(GL10.GL_
COLOR_ARRAY);
```

Теперь, когда OpenGL ES известно, что можно ожидать информацию о положении и цвете для каждой из вершин, мы сообщаем, где можно получить эту информацию:

```
vertices.position(0);
gl.glVertexPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
vertices.position(2);
gl.glColorPointer(4, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
```

Мы начинаем с установки положения нашего `FloatBuffer`. Позиция указывает на *x*-координату нашей первой вершины в буфере. Далее вызываем `glVertexPointer()`.

Единственное отличие от предыдущего примера состоит в том, что теперь мы также описываем размер вершины (помните, что он указывается в байтах). OpenGL ES начнет считывание месторасположения вершин, начиная с того адреса в буфере, которое мы укажем. Для описания положения второй вершины прибавим байты `VERTEX_SIZE` к данным первой вершины и т. д.

Далее мы устанавливаем местоположение буфера для компонента R нашей первой вершины и вызываем `glColorPointer()`, который сообщает OpenGL ES, где можно найти цвета вершин. Первый аргумент — это количество компонентов на цвет. Он всегда равен четырем, поскольку OpenGL ES требует от нас компоненты R, G, B и A для каждой вершины. Второй параметр описывает тип каждого компонента. Как и с координатами вершин, мы снова используем `GL10.GL_FLOAT` для указания, что каждый цветовой компонент является числом с плавающей точкой в диапазоне от 0 до 1. Третий параметр — это интервал между цветами вершин. Он, естественно, равен интервалу между местоположениями вершин. Последний параметр — это снова наш буфер вершин.

Поскольку мы вызвали `vertices.position(2)` перед `glColorPointer()`, OpenGL ES знает, что данные цвета первой вершины можно найти, начиная с третьего числа `float` в буфере. Если бы мы не указали, что местоположение буфера равно 2, OpenGL ES начал бы читать цвета с 0. Это было бы не очень хорошо, поскольку сначала у нас идут *x*-координаты нашей первой вершины.

Рисунок 7.8 демонстрирует, откуда OpenGL ES будет читать свойства вершин и как он перескакивает от одной вершины к другой для каждого из свойств.

Чтобы нарисовать наш треугольник, мы снова вызываем `glDrawElements()`, который приказывает OpenGL ES нарисовать треугольник, используя первые три вершины нашего `FloatBuffer`:

```
gl.glDrawElements(GL10.GL_TRIANGLES, 0, 3);
```

Поскольку мы использовали константы `GL10.GL_VERTEX_ARRAY` и `GL10.GL_COLOR_ARRAY`, OpenGL ES знает, что он должен применять свойства, определенные `glVertexPointer()` и `glColorPointer()`. Он будет игнорировать цвет, заданный по умолчанию, поскольку мы указываем отдельный цвет для каждой вершины.

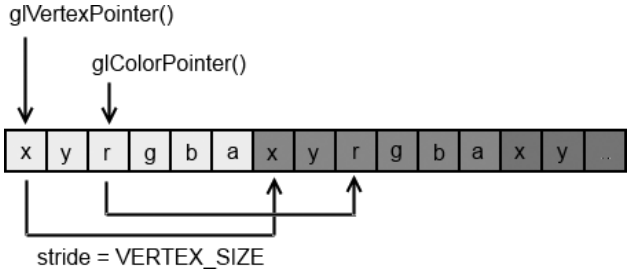

Рис. 7.8. FloatBuffer вершин, начальные адреса для OpenGL ES, чтобы прочесть местоположение/цвет, и шаг, используемый для перехода к следующему местоположению/цвету

ПРИМЕЧАНИЕ

То, как мы только что описали местоположение наших вершин, называется чередованием (interleaving). Это значит, что мы записываем все свойства вершин в один блок памяти. Достичь такого эффекта также можно, воспользовавшись массивами нечередующихся вершин (noninterleaved vertex arrays). Мы можем использовать два FloatBuffer: один для указания местоположения, другой для цвета. Тем не менее чередование гораздо удобнее применять с точки зрения экономии памяти, так что мы не будем здесь обсуждать массивы нечередующихся вершин.

Соединение всего вместе в новой реализации GLGame и Screen должно теперь показаться совсем простым. В листинге 7.6 показан фрагмент из файла ColoredTriangleTest.java. Я опустил шаблонный код.

Листинг 7.6. Отрывок из ColoredTriangleTest.java: Чередование свойств местоположения и цвета

```
class ColoredTriangleScreen extends Screen {
    final int VERTEX_SIZE = (2 + 4) * 4;
    GLGraphics glGraphics;
    FloatBuffer vertices;

    public ColoredTriangleScreen(Game game) {
        super(game);
        glGraphics = ((GLGame) game).getGLGraphics();

        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(3 * VERTEX_SIZE);
        byteBuffer.order(ByteOrder.nativeOrder());
        vertices = byteBuffer.asFloatBuffer();
        vertices.put( new float[] {  0.0f,   0.0f, 1, 0, 0, 1,
                                   319.0f,   0.0f, 0, 1, 0, 1,
                                   160.0f, 479.0f, 0, 0, 1, 1});
        vertices.flip();
        }

    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glOrthof(0, 320, 0, 480, 1, -1);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

        vertices.position(0);

        gl.glVertexPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
        vertices.position(2);
        gl.glColorPointer(4, GL10.GL_FLOAT, VERTEX_SIZE, vertices);

        gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
}
```

Здорово, получилось вполне незамысловато. Все, что мы изменили по сравнению с предыдущим примером, — добавили в наш `FloatBuffer` четыре компонента цвета для каждой из вершин и использовали `GL10.GL_COLOR_ARRAY`. Лучше всего в данных изменениях то, что любые дополнительные свойства, которые мы будем приписывать нашим вершинам позже, будут работать ровно таким же образом. Мы просто указываем OpenGL ES не использовать значения, заданные по умолчанию для данного свойства, а вместо этого брать свойства в нашем `FloatBuffer`, начиная с определенного адреса и передвигаясь от вершины к вершине по байтам `VERTEX_SIZE`.

Теперь мы можем также выключить `GL10.GL_COLOR_ARRAY`, чтобы OpenGL ES мог снова использовать цвет по умолчанию, который мы определили с помощью `glColor4f()`, как это делалось раньше. Для этого мы вызываем:

```
gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
```

OpenGL ES просто выключит свойство чтения цвета из нашего `FloatBuffer`. Если мы уже установили указатель цвета с помощью `glColorPointer()`, OpenGL ES запомнит указатель, однако мы только что приказали ему его не использовать.

Чтобы подвести итоги, рассмотрим вывод предыдущей программы (рис. 7.9).

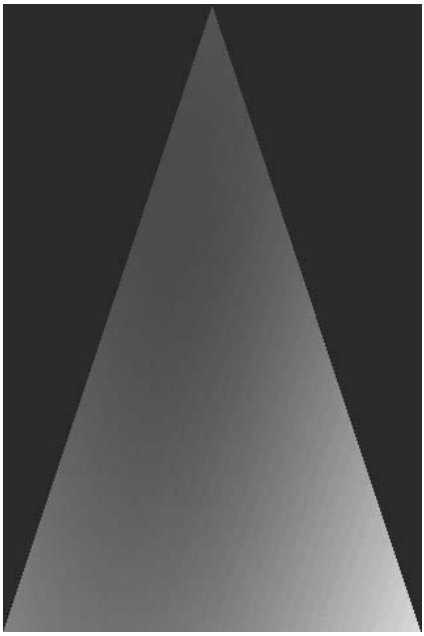

Рис. 7.9. Треугольник, где цвет каждой вершины описан отдельно

Выглядит достаточно мило. Мы не говорили о том, как OpenGL ES будет использовать те три цвета, которые мы указали (красный для нижней левой вершины, зеленый для нижней правой и синий для верхней). Как оказалось, он интерполировал цвета между вершинами. Такой подход позволяет без труда создавать хоро-

шие градиенты. Однако нам явно недостаточно только использования цвета, ведь мы собираемся рисовать изображения с помощью OpenGL ES. Вот где нам пригодится обработка текстуры.

Обработка текстур: легкая работа с фоновыми рисунками

Когда мы писали «Мистера Нома», мы загрузили ряд растровых изображений и напрямую отправили их в фреймбуфер — без поворотов, только слегка изменив масштабы, что было весьма несложно. В OpenGL ES нас в основном интересуют треугольники, которые могут иметь любую ориентацию и размер в зависимости от поставленной задачи. Как же визуализировать изображения с помощью OpenGL ES?

Все достаточно просто: загрузите изображения в OpenGL ES (точнее — передайте на обработку графическому процессору, который имеет свою выделенную RAM), добавьте новые свойства каждой вершине треугольника и прикажите OpenGL ES визуализировать наш треугольник и применить изображение (или *текстуру*, говоря на языке OpenGL ES) к нашему треугольнику. Рассмотрим сначала, что определяют эти новые свойства вершин.

Координаты текстур

Чтобы ассоциировать растровое изображение с треугольником, сначала нужно добавить к каждой вершине треугольника так называемые *координаты текстур*. Что же такое координата текстуры? Она определяет точку внутри текстуры (растрового изображения, которое мы загрузили), которая будет ассоциирована с одной из вершин треугольника. Координаты текстур, как правило, задаются для двух измерений.

Координаты месторасположения обозначают буквами x, y и z, а текстуры координат обычно называют u и v или s и t. В OpenGL ES их обозначают s и t, так что мы будем придерживаться этого названия. Если вы читаете какие-либо статьи в Интернете, которые используют названия u/v, не переживайте, это то же самое, что и s и t. Как же выглядит система координат? На рис. 7.10 показан Боб в текстурной системе координат после того, как мы загрузили его в OpenGL ES.

Здесь происходит несколько интересных вещей. Прежде всего, s равно координате x в стандартной системе координат, а t, в свою очередь, равно координате y. Ось s направлена вправо, а ось t — вниз. Начало системы координат совпадает с левым верхним углом изображения Боба. Нижний правый угол изображения имеет координаты (1; 1).

Что же случилось с пиксельной системой координат? Как оказалось, OpenGL ES не особо ее любит. Любое изображение, которое мы загружаем, вне зависимости от его ширины и высоты в пикселах, вставляется в данную систему координат. Верхний левый угол изображения всегда будет находиться в точке (0; 0), а нижний

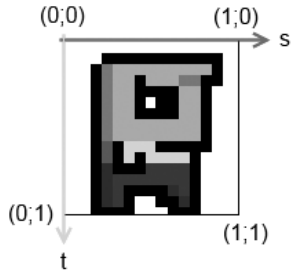

Рис. 7.10. Боб, загруженный в OpenGL ES и показанный в текстурной системе координат

правый угол — в (1; 1), даже если его ширина в два раза больше высоты. Это называется нормализованными координатами. Они на самом деле иногда значительно облегчают нам жизнь. Так как же ассоциировать Боба с нашим треугольником? Все просто, мы присваиваем каждой вершине треугольника пару текстурных координат в координатной системе Боба. На рис. 7.11 показаны несколько конфигураций.

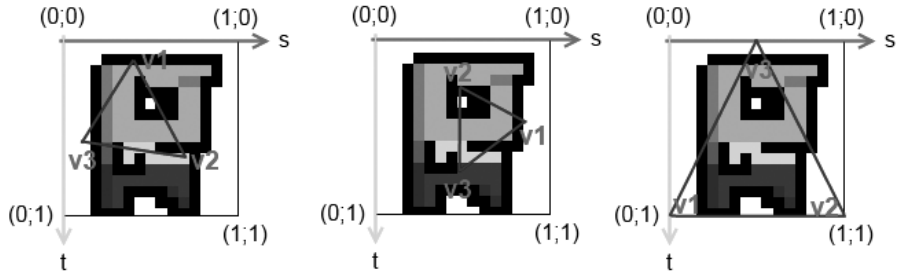

Рис. 7.11. Три разных треугольника, нанесенных на Боба; координаты v1, v2 и v3 указывают вершины треугольника

Мы можем ассоциировать вершины треугольника с системой текстурных координат так, как захотим. Обратите внимание, что ориентация треугольника в системе позиционных координат может не совпадать с ориентацией в системе текстурных координат.

Данные системы координат никак не связаны. Рассмотрим, как добавить эти координаты текстур к нашим вершинам:

```
Int VERTEX_SIZE = (2 + 2) * 4;
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(3 * VERTEX_SIZE);
byteBuffer.order(ByteOrder.nativeOrder());
vertices = byteBuffer.asFloatBuffer();
vertices.put( new float[] {   0.0f,   0.0f, 0.0f, 1.0f,
                            319.0f,   0.0f, 1.0f, 1.0f,
                            160.0f, 479.0f, 0.5f, 0.0f});
vertices.flip();
```

Несложно, правда? Все, что нам осталось сделать, — убедиться, что у нас хватает места в буфере, а затем применить координаты текстур к каждой вершине. Предыдущий код соответствует самому правому варианту ассоциации, показанному на рис. 7.10. Обратите внимание, что положения наших вершин по-прежнему даются в обычной системе координат, которую мы описали с помощью проекции. При желании мы могли бы также добавить каждой вершине цветовые атрибуты, как в предыдущем примере. Затем OpenGL ES смешает интерполированные цвета вершин с цветами пикселов той текстуры, с которой ассоциируется треугольник. Это происходит динамически. Естественно, нам нужно настроить размер буфера, а также, соответственно, константы `VERTEX_SIZE` (например, $(2 + 4 + 2) \cdot 4$)). Чтобы сообщить OpenGL ES, что у наших вершин есть текстурные координаты, мы снова используем методы `glEnableClientState()` и `glTexCoordPointer()`, которые функционируют так же, как `glVertexPointer()` и `glColorPointer()` (видите закономерность?).

```
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY); gl.glEnableClientState(GL10.GL_
TEXTURE_COORD_ARRAY);

vertices.position(0);
gl.glVertexPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
vertices.position(2);
gl.glTexCoordPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
```

Отлично, все выглядит знакомым. Остается всего лишь один вопрос: как нам загрузить текстуры в OpenGL ES и указать ассоциировать их с треугольником? Естественно, это немного сложнее, однако тоже не составляет труда.

Загрузка растровых изображений

Для начала необходимо загрузить наше растровое изображение. Вы уже знаете, как это делается в Android:

```
Bitmap bitmap =
BitmapFactory.decodeStream(game.getFileIO().readAsset("bobrgb888.png"));
```

Здесь мы загружаем Боба в конфигурации RGB888. Затем нам нужно сообщить OpenGL ES, что мы хотим создать новую текстуру. Чтобы создать объект текстуры, можно вызвать следующий метод:

```
GL10.glGenTextures(int numTextures, int[] ids, int offset)
```

Первый параметр определяет, как много объектов текстур мы хотим создать. Обычно создаем всего одну. Следующий параметр — это массив `int`, куда OpenGL ES будет записывать ID сгенерированных объектов текстур.

Последний параметр просто сообщает OpenGL ES, с какой точки массива нужно начинать записывать ID.

Вы уже знаете, что OpenGL ES — это API на языке C. Естественно, он не может вернуть нам Java-объект, соответствующий новой текстуре. Вместо этого он дает ID этой текстуры. Каждый раз, когда мы хотим, чтобы OpenGL ES что-то сделал с данной текстурой, мы указываем ее ID. Вот более сложный фрагмент кода, который демонстрирует, как сгенерировать новый текстурный объект и получить его ID:

```
int textureIds[] = new int[1];
gl.glGenTextures(1, textureIds, 0);
int textureId = textureIds[0];
```

Объект текстуры по-прежнему пуст. Это значит, что у него по-прежнему нет никаких графических данных. Загрузим наше растровое изображение. Для этого нам необходимо сначала привязать текстуру. В OpenGL ES под привязкой чего-либо понимается, что мы хотим, чтобы OpenGL ES использовал данный конкретный объект для всех последующих вызовов до тех пор, пока мы снова не изменим привязку. В данном случае мы хотим привязать текстуру объекта. Для этого мы используем метод `glBindTexture()`. Как только привяжем текстуру, сможем управлять ее свойствами, такими как данные об изображении. Вот как загрузить Боба в наш новый объект текстуры:

```
gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId); GLUtils.texImage2D(GL10.GL_
TEXTURE_2D, 0, bitmap, 0);
```

Сначала мы привязываем объект текстуры при помощи `glBindTexture()`. Первый параметр определяет тип текстуры, которую мы хотим привязать. Наше изображение Боба — двухмерное, поэтому мы используем `GL10.GL_TEXTURE_2D`. Существуют и другие типы текстур, но в этой книге они нам не понадобятся. Мы всегда применяем `GL10.GL_TEXTURE_2D` для методов, которым необходима информация, с каким типом текстуры мы хотим работать. Второй параметр данного метода — ID текстуры. Когда метод выполнится, все последующие методы, которые работают с двухмерной текстурой, будут работать с нашим объектом текстуры.

При следующем вызове мы активируем метод класса `GLUtils`, этот класс предоставляется во фреймворке Android. Обычно задача загрузки изображения текстуры достаточно сложна, однако этот небольшой вспомогательный класс значительно все облегчает. Все, что нужно сделать, — определить тип текстуры (`GL10.GL_TEXTURE_2D`), уровень детализации (мы подробно рассмотрим его в главе 11, по умолчанию он равен нулю), изображение, которое мы хотим загрузить, и еще один аргумент, который должен быть равен нулю во всех случаях. После этого вызова к объекту текстуры будут прикреплены графические данные.

ПРИМЕЧАНИЕ

Объект текстуры и его графические данные фактически хранятся в видеопамяти, а не в обычной оперативной памяти. Объект текстуры (и графические данные) будут утеряны при потере контекста OpenGL ES (когда наша активность ставится на паузу, а затем возобновляется). Это значит, что надо будет заново воссоздавать объект текстуры и заново загружать графические данные каждый раз, когда создается контекст OpenGL ES. Если мы этого не сделаем, то увидим только белый треугольник.

Фильтрация текстур

Есть еще одна деталь, которую нам необходимо определить перед тем, как мы сможем использовать объект текстуры. Она связана с тем, что наш треугольник может занимать больше или меньше пикселов на экране по сравнению с тем, сколько пикселов есть в обозначенной зоне текстуры. Например, изображение Боба на рис. 7.10 имеет размер 128×128 пикселов. Наш треугольник занимает половину этого изображения, так что он использует $128 \cdot 128 / 2$ пиксела из текстуры (текстурные пикселы также называются «текселы»). Когда мы нарисуем на экране треугольник с координатами, которые описали в предыдущем фрагменте кода, он будет занимать $320 \cdot 480 / 2$ пикселов. На экране мы используем гораздо больше пикселов по сравнению с тем, что перенесли из зоны текстуры. Естественно, все может быть и наоборот: мы используем меньше пикселов на экране, чем на выделенной зоне текстуры. Первый случай называется *магнификацией*, а второй — *минификацией*. В каждом из них нам необходимо сообщить OpenGL ES, как нужно увеличивать или уменьшать текстуру. В терминологии OpenGL ES соответствующие механизмы называются фильтрами минификации и магнификации. Эти фильтры являются свойствами объекта текстуры, как и сами данные изображения. Чтобы их установить, необходимо сначала проверить, привязан ли объект текстуры с помощью `glBindTexture()`. Если это так, устанавливаем их следующим образом:

```
gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER, GL10.GL_NEAREST);
gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MAG_FILTER, GL10.GL_NEAREST);
```

В обоих случаях используем метод `GL10.glTexParameterf()`, который устанавливает свойства текстуры. В первом случае мы определяем фильтр минификации, а в другом — фильтр магнификации. Первый параметр в этом методе — тип текстуры, который в нашем случае по умолчанию равен `GL10.GL_TEXTURE_2D`. Второй параметр сообщает методу, какое свойство мы хотим установить. В нашем случае это `GL10.GL_TEXTURE_MIN_FILTER` и `GL10.GL_TEXTURE_MAG_FILTER`. Последний параметр указывает тип фильтра, который должен быть использован. Здесь у нас есть два варианта: `GL10.GL_NEAREST` и `GL10.GL_LINEAR`.

Первый тип фильтра всегда выбирает ближайший текстурный пиксел в зоне текстуры, который будет ассоциирован с пикселом из изображения. Второй тип фильтра будет использовать четыре ближних текстурных пиксела для ассоциирования с пикселом треугольника, а затем находить их среднее значение для конечного цвета. Мы применяем фильтр первого типа, если нам нужно мозаичное изображение, и второй тип, если мы хотим более сглаженный вариант.

На рис. 7.12 показана разница между этими двумя типами фильтров.

Теперь наш объект текстуры полностью определен: мы создали ID, установили графические данные и определили фильтры, которые должны быть использованы, если наше изображение неидеально. Как правило, мы отвязываем текстуру после того, как закончим ее описывать. Нам также необходимо переработать растровый

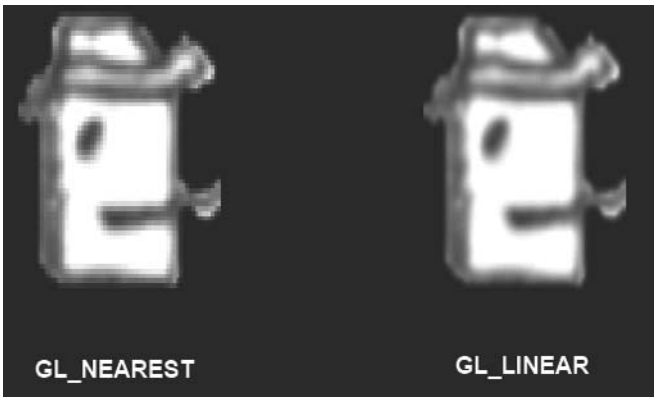

Рис. 7.12. Фильтр GL10.GL_NEAREST делает изображение мозаичным *(слева)*, фильтр GL10.GL_LINEAR немного сглаживает изображение *(справа)*

рисунок `Bitmap`, который мы загрузили, поскольку он нам больше не нужен. Зачем тратить память зря? Отвязывание можно сделать с помощью следующего фрагмента кода:

```
gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
bitmap.recycle();
```

`0` — это специальный ID, который сообщает OpenGL ES, что следует отвязать текущий привязанный объект. Если мы хотим использовать текстуру для рисования треугольников, нам, естественно, нужно заново ее привязать.

Удаление текстур

Полезно также знать, как удалить объект текстуры из видеопамяти, если он нам больше не нужен (так же, как мы используем `Bitmap.recycle()`, чтобы освободить память рисунка). Это можно сделать с помощью следующего фрагмента:

```
gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
int textureIds = { textureid };
gl.glDeleteTextures(1, textureIds, 0);
```

Обратите внимание: сначала нам нужно удостовериться, что объект текстуры, который мы хотим удалить, не привязан. Оставшаяся часть кода напоминает этап с использованием `glGenTextures()`для создания объекта текстуры.

Полезный фрагмент кода

Вот (для справки) полный пример кода, предназначенный для объекта текстуры, загрузки графических данных и установки фильтров на Android:

```
Bitmap bitmap = BitmapFactory.decodeStream(game.getFileIO().readAsset("bobrgb888.png"));
int textureIds[] = new int[1];
gl.glGenTextures(1, textureIds, 0);
```

```
int textureId = textureIds[0];
gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER, GL10.GL_NEAREST);
gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MAG_FILTER, GL10.GL_NEAREST);
gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
bitmap.recycle();
```

Совсем неплохо. Самое важное здесь — заново использовать рисунок `Bitmap`, когда мы закончим. Иначе мы будем впустую тратить память. Данные нашего изображения надежно хранятся в видеопамяти в объекте текстуры (до тех пор пока контекст не будет утерян и нам не понадобится заново все загрузить).

Активация текстурирования

Еще кое-что, что нам необходимо сделать перед тем, как мы нарисуем наш треугольник с текстурой. Нужно привязать текстуру, а также сообщить OpenGL ES, что он должен применить текстуру ко всем треугольникам, которые мы визуализируем. Еще одно состояние OpenGL ES касается того, нужно ли ассоциировать текстуру. Мы можем включать и выключать этот механизм с помощью следующих методов:

```
GL10.glEnable(GL10.GL_TEXTURE_2D);
GL10.glDisable(GL10.GL_TEXTURE_2D);
```

Это должно быть вам уже знакомо. Когда мы включали/выключали свойства вершины ранее, мы использовали метод `glEnableClientState()/glDisableClientState()`. Как я говорил раньше, это все особенности OpenGL. На самом деле есть причины, почему они не включены в `glEnable()/glDisable()`, но не будем в это углубляться. Просто запомните, что надо использовать `glEnableClientState()/glDisableClientState()`, чтобы включить/выключить свойства вершины, а `glEnable()/glDisable()` для всех других состояний OpenGL, например текстурирования.

Все вместе

Теперь мы наконец-то можем написать небольшой пример, который соединит все вместе. В листинге 7.7 показан отрывок из файла с исходным кодом `TexturedTriangleTest.java`. Здесь приведена только нужная нам часть с классом `TexturedTriangleScreen`.

Листинг 7.7. Отрывок из `TexturedTriangleTest.java`; текстурирование треугольника

```
class TexturedTriangleScreen extends Screen {
    final int VERTEX_SIZE = (2 + 2) * 4;
    GLGraphics glGraphics;
    FloatBuffer vertices;
    int textureId;

    public TexturedTriangleScreen(Game game) {
        super(game);
```

```
        glGraphics = ((GLGame) game).getGLGraphics();

        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(3 * VERTEX_SIZE);
        byteBuffer.order(ByteOrder.nativeOrder());
        vertices = byteBuffer.asFloatBuffer();
        vertices.put( new float[] {   0.0f,   0.0f, 0.0f, 1.0f,
                                    319.0f,   0.0f, 1.0f, 1.0f,
                                    160.0f, 479.0f, 0.5f, 0.0f});
        vertices.flip();
        textureId = loadTexture("bobrgb888.png");
    }

    public int loadTexture(String fileName) {
        try {
            Bitmap bitmap =
    BitmapFactory.decodeStream(game.getFileIO().readAsset(fileName));
            GL10 gl = glGraphics.getGL();
            int textureIds[] = new int[1];
            gl.glGenTextures(1, textureIds, 0);
            int textureId = textureIds[0];
            gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
            GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
            gl.glTexParameterf(GL10.GL_TEXTURE_2D,
            GL10.GL_TEXTURE_MIN_FILTER, GL10.GL_NEAREST);
            gl.glTexParameterf(GL10.GL_TEXTURE_2D,
            GL10.GL_TEXTURE_MAG_FILTER, GL10.GL_NEAREST);
            gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
            bitmap.recycle();
            return textureId;
        } catch(IOException e) {
            Log.d("TexturedTriangleTest", "couldn't load asset
                    'bobrgb888.png'!");
            throw new RuntimeException("couldn't load asset '"
                    + fileName + "'");
        }
    }

    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glOrthof(0, 320, 0, 480, 1, -1);

        gl.glEnable(GL10.GL_TEXTURE_2D);
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
```

```
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

        vertices.position(0);
        gl.glVertexPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
        vertices.position(2);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
        gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
    }
```

Я решил поместить загрузку текстуры в метод `loadTexture()`, который просто использует имя загружаемого изображения. Метод возвращает ID-объект текстуры, сгенерированный OpenGL ES, который мы будем прменять в методе `present()` для привязки текстуры.

Описание нашего треугольника вряд ли вас удивит, мы просто добавили координаты текстур к каждой вершине.

Метод `present()` делает то же, что и всегда: очищает экран и устанавливает матрицу проекции. Затем активируем ассоциирование текстур с помощью вызова `glEnable()` и привязываем наш объект текстуры. Оставшаяся часть такая же, как всегда: включение свойств вершины, которые мы хотим использовать, сообщение OpenGL ES, где их можно найти и какие шаги при этом использовать, и, наконец, рисование треугольника с вызовом `glDrawArrays()`. На рис. 7.13 показаны результаты выполнения предыдущего кода.

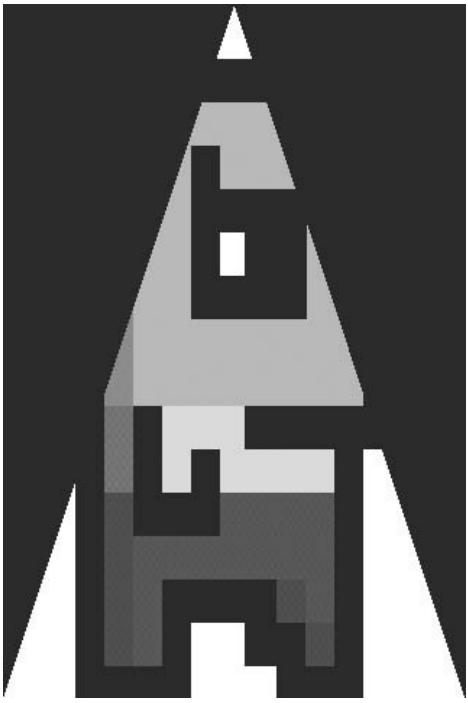

Рис. 7.13. Ассоциирование текстуры Боба с треугольником

Осталась еще одна очень важная вещь, которую мы не обсудили. *Все растровые изображения, которые мы загружаем, должны иметь ширину и высоту в степени 2*. Это аксиома, и если ей не следовать, возникнут проблемы.

Что это значит? Изображение Боба, которое мы использовали в нашем примере, имеет размер 128 × 128 пикселов. 128 — это 2 в седьмой степени (2 · 2 · 2 · 2 · 2 · 2 · 2). Другие размеры изображения могут быть 2 × 8, 32 × 16, 128 × 256 и т. д. Существует также предел того, насколько велики могут быть изображения. К сожалению, он варьируется в зависимости от возможностей аппаратуры, на которой запущено изображение. Насколько мне известно, стандарт OpenGL ES 1.x не описывает минимальный поддерживаемый размер текстуры. Тем не менее, исходя из моего опыта, текстура 512 × 512 пикселов работает на всех современных устройствах Android (и, скорее всего, будет работать на всех будущих). Я даже позволю себе предположить, что размер 1024 × 1024 также подойдет.

Еще один момент, который мы до сих пор игнорировали, — это глубина цвета текстур. К счастью, метод `GLUtils.texImage2D()`, который мы использовали для загрузки данных нашего изображения в центральный процессор, достаточно успешно справляется с этим. OpenGL ES может работать с такой глубиной цвета, как RGBA8888, RGB565 и т. д. Мы всегда должны пытаться использовать наименьшую возможную глубину цвета, чтобы уменьшить необходимую пропускную способность. Для этого мы можем использовать класс `BitmapFactory.Options`, как в предыдущих главах, чтобы, например, загрузить изображение RGB888 или изображение RGB565. Когда мы загрузили экземпляр класса `Bitmap` с той глубиной цвета, которая нам нужна, мы используем `GLUtils.texImage2D()`, который проверяет, получил ли OpenGL ES данные изображения в верном формате. Конечно, вы всегда должны проверять, сказывается ли уменьшение глубины цвета негативно на внешнем виде игры.

Класс Texture

Чтобы уменьшить количество кода, необходимого для следующих примеров, я написал небольшой вспомогательный класс `Texture`. Он загрузит изображение из ресурса и создаст из него объект текстуры. У него также есть несколько удобных методов, чтобы привязать текстуру и чтобы избавиться от нее. Код приведен в листинге 7.8.

Листинг 7.8. Texture.java, небольшой класс Texture для OpenGL ES

```
package com.badlogic.androidgames.framework.gl;

import java.io.IOException;
import java.io.InputStream;

import javax.microedition.khronos.opengles.GL10;

import android.graphics.Bitmap;
```

```
import android.graphics.BitmapFactory;
import android.opengl.GLUtils;

import com.badlogic.androidgames.framework.FileIO;
import com.badlogic.androidgames.framework.impl.GLGame;
import com.badlogic.androidgames.framework.impl.GLGraphics;

public class Texture {
    GLGraphics glGraphics;
    FileIO fileIO;
    String fileName;
    int textureId;
    int minFilter;
    int magFilter;

    public Texture(GLGame glGame, String fileName) {
        this.glGraphics = glGame.getGLGraphics();
        this.fileIO = glGame.getFileIO();
        this.fileName = fileName;
        load();
    }

    private void load() {
        GL10 gl = glGraphics.getGL();
        int[] textureIds = new int[1];
        gl.glGenTextures(1, textureIds, 0);
        textureId = textureIds[0];

        InputStream in = null;
        try {
            in = fileIO.readAsset(fileName);
            Bitmap bitmap = BitmapFactory.decodeStream(in);
            gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
            GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
            setFilters(GL10.GL_NEAREST, GL10.GL_NEAREST);
            gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
        } catch(IOException e) {
            throw new RuntimeException("Couldn't load texture '"
                                       + fileName +"'", e);
        } finally {
            if(in != null)
                try { in.close(); } catch (IOException e) { }
        }
    }
    public void reload() {
        load();
        bind();
        setFilters(minFilter, magFilter);
```

```
        glGraphics.getGL().glBindTexture(GL10.GL_TEXTURE_2D, 0);
    }

    public void setFilters(int minFilter, int magFilter) {
        this.minFilter = minFilter;
        this.magFilter = magFilter;
        GL10 gl = glGraphics.getGL();
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER,
                           minFilter);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MAG_FILTER,
                           magFilter); }

    public void bind() {
        GL10 gl = glGraphics.getGL();
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
    }

    public void dispose() {
        GL10 gl = glGraphics.getGL();
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
        int[] textureIds = { textureId };
        gl.glDeleteTextures(1, textureIds, 0);
    }
}
```

Единственный интересный компонент этого класса — метод `reload()`, который мы можем использовать, когда контекст OpenGL ES утерян. Обратите также внимание, что метод `setFilters()` будет работать только в том случае, если `Texture` привязан. В противном случае он установит фильтры текущей привязанной текстуры.

Мы могли бы также написать небольшой вспомогательный метод для обработки буфера наших вершин. Но перед этим нужно обсудить еще одну деталь: индексированные вершины.

Индексированные вершины: используем повторно

До сих пор мы всегда описывали список треугольников, где каждый треугольник имеет собственный набор вершин. В действительности мы успели нарисовать только один треугольник, но добавить еще несколько не составит особого труда.

Тем не менее существуют случаи, когда два или больше треугольников могут делить некоторые вершины. Рассмотрим, как мы можем визуализировать прямоугольник с помощью уже изученных приемов. Мы просто определим два треугольника, которые будут иметь две вершины с одинаковыми координатами места, цветами и текстурами.

Однако мы можем сделать еще лучше. На рис. 7.14 показаны старый и новый способы визуализации прямоугольника.

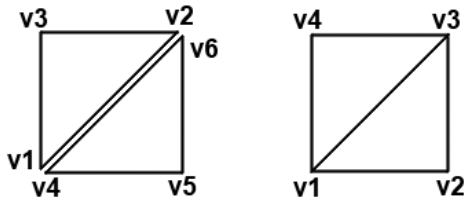

Рис. 7.14. Визуализация прямоугольника как двух треугольников с шестью вершинами *(слева)* и его визуализация с четырьмя вершинами *(справа)*

Вместо дублирования вершин `v1` и `v2` с вершинами `v4` и `v6` мы просто описываем эти вершины один раз. Мы по-прежнему визуализируем два треугольника в данном случае, однако явно сообщаем OpenGL ES, какие вершины использовать для каждого треугольника (например, `v1`, `v2` и `v3` для первого треугольника и `v3`, `v4` и `v1` для второго). То, какие вершины применять для каждого треугольника, определяется по индексам в массиве вершин. Первая вершина в нашем массиве имеет индекс 0, вторая вершина имеет индекс 1 и т. д. Для предыдущего прямоугольника у нас будет следующий список индексов:

```
short[] indices = { 0, 1, 2,
                    2, 3, 0 };
```

Кстати, OpenGL ES предпочитает определять индексы в формате чисел `short` (правда, здесь мы также можем использовать байты). Тем не менее, как и с данными вершин, мы не можем просто передать OpenGL ES массив `short`. Ему нужен прямой `ShortBuffer`. Вы уже знаете, как это делать:

```
ByteBuffer byteBuffer = ByteBuffer.allocate(indices.length * 2);
byteBuffer.order(ByteOrder.nativeOrder());
ShortBuffer shortBuffer = byteBuffer.asShortBuffer();
shortBuffer.put(indices);
shortBuffer.flip();
```

`Short` нужно 2 байта памяти, так что выделяем `indices.length` · 2 байтов для нашего `ShortBuffer`. Снова устанавливаем изначальный порядок и получаем вид `ShortBuffer`, чтобы нам было проще обработать базовый `ByteBuffer`. Все, что осталось, — собрать все индексы в `ShortBuffer` и вызвать `flip()`, чтобы все позиции были установлены правильно.

Если бы нам было нужно нарисовать Боба как прямоугольник с двумя индексированными треугольниками, мы могли бы описать вершины следующим образом:

```
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(4 * VERTEX_SIZE);
byteBuffer.order(ByteOrder.nativeOrder());
vertices = byteBuffer.asFloatBuffer();
vertices.put(new float[] { 100.0f, 100.0f, 0.0f, 1.0f,
                           228.0f, 100.0f, 1.0f, 1.0f,
                           228.0f, 229.0f, 1.0f, 0.0f,
                           100.0f, 228.0f, 0.0f, 0.0f });
vertices.flip();
```

Порядок вершин полностью совпадает с правой частью рис. 7.14. Мы сообщаем OpenGL ES, что у нас есть координаты положения, а также текстуры для наших вершин. Еще сообщаем, где можно найти эти свойства вершин с помощью обычного вызова `glEnableClientState()` и `glVertexPointer()`/`glTexCoordPointer()`. Единственная разница состоит в методе, который мы вызываем для рисования двух треугольников:

```
gl.glDrawElements(GL10.GL_TRIANGLES, 6, GL10.GL_UNSIGNED_SHORT, indices);
```

Все это очень похоже на `glDrawArrays()`. Первый параметр определяет тип элемента, который мы хотим визуализировать, — в нашем случае это список треугольников. Следующий параметр задает, сколько вершин мы хотим использовать (в нашем случае — шесть). Третий параметр указывает, какой тип индексов мы задействуем (у нас это числа `short` без знака). Обратите внимание, что в Java нет типов беззнаковых типов. Тем не менее, учитывая кодирование чисел со знаками с поразрядным дополнением до единицы, вполне уместно использовать `ShortBuffer`, который работает с числами `short` со знаками. Последний параметр — наш `ShortBuffer`, содержащий шесть индексов.

Что же делает OpenGL ES? Системе известно, что мы хотим визуализировать два треугольники, поскольку мы указали, что собираемся визуализировать шесть вершин. Но вместо того, чтобы перенести шесть вершин по очереди из массива вершин, он последовательно проходит через буфер индексов и использует индексированные вершины.

Все вместе

Когда мы все объединим, получим код, как в листинге 7.9.

Листинг 7.9. Отрывок из IndexedTest.java; рисуем два индексированных треугольника

```
class IndexedScreen extends Screen {
    final int VERTEX_SIZE = (2 + 2) * 4;
    GLGraphics glGraphics;
    FloatBuffer vertices;
    ShortBuffer indices;
    Texture texture;

    public IndexedScreen(Game game) {
        super(game);
        glGraphics = ((GLGame) game).getGLGraphics();

        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(4 * VERTEX_SIZE);
        byteBuffer.order(ByteOrder.nativeOrder());
        vertices = byteBuffer.asFloatBuffer();
        vertices.put(new float[] { 100.0f, 100.0f, 0.0f, 1.0f,
                                   228.0f, 100.0f, 1.0f, 1.0f,
                                   228.0f, 228.0f, 1.0f, 0.0f,
                                   100.0f, 228.0f, 0.0f, 0.0f });
```

```
        vertices.flip();

        byteBuffer = ByteBuffer.allocateDirect(6 * 2);
        byteBuffer.order(ByteOrder.nativeOrder());
        indices = byteBuffer.asShortBuffer();
        indices.put(new short[] { 0, 1, 2,
                                  2, 3, 0 });
        indices.flip();

        texture = new Texture((GLGame)game, "bobrgb888.png");
    }

    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glOrthof(0, 320, 0, 480, 1, -1);

        gl.glEnable(GL10.GL_TEXTURE_2D);
        texture.bind();

        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

        vertices.position(0);
        gl.glVertexPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);
        vertices.position(2);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, VERTEX_SIZE, vertices);

        gl.glDrawElements(GL10.GL_TRIANGLES, 6, GL10.GL_UNSIGNED_SHORT,
                          indices);
    }
```

Обратите внимание, как удобно использовать класс `Texture`, который значительно сокращает количество кода. На рис. 7.15 показан результат.

Теперь все очень похоже на то, как мы работали с `Canvas`. У нас также гораздо больше гибкости сейчас, поскольку мы больше не ограничены осями.

Этот пример описывает все, что нам на данный момент надо знать о вершинах. Мы увидели, что каждая вершина должна иметь как минимум положение, но она также может обладать дополнительными свойствами, такими как цвет, данный в четырех значениях с плавающей точкой (формат RGBA), и текстурные координаты. Мы также увидели, что можем повторно использовать вершины с помощью индексации в случае, если хотим избежать дублирования. Это позволяет нам немного ускорить работу, поскольку OpenGL ES придется умножать вершины на проекцию и матрицы вида модели не более, чем требуется (давайте остановимся на такой интерпретации, хотя она и не совсем верна).

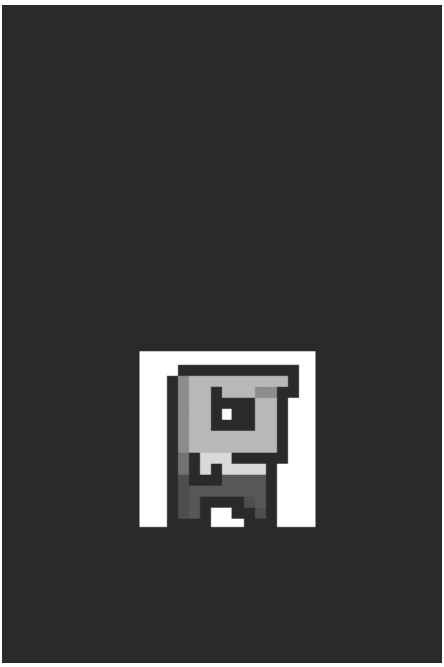

Рис. 7.15. Индексированный Боб

Класс Vertices

Давайте упростим код, создав класс `Vertices`, который может содержать максимальное количество вершин и при необходимости индексов, используемых при визуализации. Он должен также обеспечивать все состояния, нужные для визуализации, а также очищать состояния после визуализации, чтобы другой код мог основываться на чистом наборе состояний OpenGL ES. В листинге 7.10 показан класс `Vertices`.

Листинг 7.10. `Vertices.java`: инкапсулированные (индексированные) вершины

```
package com.badlogic.androidgames.framework.gl;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;

import java.nio.ShortBuffer;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.impl.GLGraphics;

public class Vertices {
    final GLGraphics glGraphics;
    final boolean hasColor;
```

```
    final boolean hasTexCoords;
    final int vertexSize;
    final FloatBuffer vertices;
    final ShortBuffer indices;
```

Класс `Vertices` содержит ссылку на экземпляр класса `GLGraphics`, поэтому мы можем получить экземпляр класса `GL10`, как только он понадобится. Мы также сохраняем информацию о том, имеют ли вершины цвет и текстурные координаты. Так достигается значительная гибкость, поскольку мы можем выбрать минимальный набор свойств, которые нам нужны для визуализации. Мы также сохраняем `FloatBuffer`, который содержит наши вершины, и `ShortBuffer`, включающий в себя дополнительные индексы.

```
    public Vertices(GLGraphics glGraphics, int maxVertices, int maxIndices,
boolean hasColor, boolean hasTexCoords) {
        this.glGraphics = glGraphics;
        this.hasColor = hasColor;
        this.hasTexCoords = hasTexCoords;
        this.vertexSize = (2 + (hasColor?4:0) + (hasTexCoords?2:0)) * 4;

        ByteBuffer buffer = ByteBuffer.allocateDirect(maxVertices *
                                                      vertexSize);
        buffer.order(ByteOrder.nativeOrder());
        vertices = buffer.asFloatBuffer();

        if(maxIndices > 0) {
            buffer = ByteBuffer.allocateDirect(maxIndices * Short.SIZE / 8);
            buffer.order(ByteOrder.nativeOrder());
            indices = buffer.asShortBuffer();
        } else {
            indices = null;
        }
    }
```

В конструкторе определяем, какое максимальное количество вершин и индексов может содержать экземпляр класса `Vertices`, а также имеют ли вершины цвет и текстурные координаты. Внутри конструктора устанавливаем соответствующие элементы и указываем значения буферам. Обратите внимание: если `maxIndices` равен нулю, `ShortBuffer` получит значение `null`. В таком случае визуализация будет выполняться без индексирования.

```
    public void setVertices(float[] vertices, int offset, int length) {
        this.vertices.clear();
        this.vertices.put(vertices, offset, length);
        this.vertices.flip();
    }

    public void setIndices(short[] indices, int offset, int length) {
        this.indices.clear();
        this.indices.put(indices, offset, length);
        this.indices.flip();
    }
```

Переходим к методам setVertices() и setIndices(). Последний будет выдавать исключение NullPointerException, если экземпляр класса Vertices не сохраняет индексы. Все, что нам надо делать, — очищать буферы и копировать содержимое массивов.

```
    public void draw(int primitiveType, int offset, int numVertices) {
        GL10 gl = glGraphics.getGL();

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        vertices.position(0);
        gl.glVertexPointer(2, GL10.GL_FLOAT, vertexSize, vertices);

        if(hasColor) {
            gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
            vertices.position(2);
            gl.glColorPointer(4, GL10.GL_FLOAT, vertexSize, vertices);
        }

        if(hasTexCoords) {
            gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
            vertices.position(hasColor?6:2);
            gl.glTexCoordPointer(2, GL10.GL_FLOAT, vertexSize, vertices);
        }

        if(indices!=null) {
            indices.position(offset);
            gl.glDrawElements(primitiveType, numVertices,
                              GL10.GL_UNSIGNED_SHORT, indices);
        } else {
            gl.glDrawArrays(primitiveType, offset, numVertices);
        }

        if(hasTexCoords)
            gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

        if(hasColor)
            gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
    }
}
```

Последний метод класса Vertices — draw(). Он использует тип элемента (например, GL10.GL_TRIANGLES), смещение в буфере вершин (или буфере индексов, если мы пользуемся индексами) и количество вершин, применяемых для визуализации. В зависимости от того, имеют ли вершины цвет и текстурные координаты, включаем соответствующие состояния OpenGL ES и сообщаем OpenGL ES, где найти данные. Естественно, мы делаем то же самое для положения вершин, поскольку эта информация будет нам требоваться постоянно. В зависимости от того, используются ли индексы, вызываем glDrawElements() или glDrawArrays() с соответствующими параметрами. Обратите внимание, что параметр смещения

может также применяться при индексированной визуализации: мы просто устанавливаем положение буфера индексов таким образом, что OpenGL ES начинает читать индексы с точки, где произошло смещение, а не с первого индекса буфера. В методе `draw()` осталось только очистить состояния OpenGL ES. Вызываем `glDisableClientState()` с `GL10.GL_COLOR_ARRAY` или `GL10.GL_TEXTURE_COORD_ARRAY`, если у вершин есть эти атрибуты. Нам нужно это сделать, поскольку другой экземпляр класса `Vertices` может не использовать эти атрибуты. Если бы мы визуализировали другой экземпляр класса `Vertices`, OpenGL ES по-прежнему искал бы координаты цвета и текстур.

Мы можем заменить весь сложный код конструктора из предыдущего примера следующим фрагментом:

```
Vertices vertices = new Vertices(glGraphics, 4, 6, false, true);
vertices.setVertices(new float[] { 100.0f, 100.0f, 0.0f, 1.0f,
                                   228.0f, 100.0f, 1.0f, 1.0f,
                                   228.0f, 228.0f, 1.0f, 0.0f,
                                   100.0f, 228.0f, 0.0f, 0.0f }, 0, 16);
vertices.setIndices(new short[] { 0, 1, 2, 2, 3, 0 }, 0, 6);
```

Мы также можем заменить все вызовы, занятые установкой наших массивов с атрибутами, а также занятые отображением, следующим вызовом:

```
vertices.draw(GL10.GL_TRIANGLES, 0, 6);
```

Вместе с классом `Texture` у нас может получиться достаточно хорошая основа для двухмерной визуализации в OpenGL ES. Одна из деталей, которых нам до сих пор не хватает для полного воссоздания возможностей нашего `Canvas`, — это смешивание. Давайте им займемся.

Альфа-смешивание: я вижу тебя насквозь

Реализовать альфа-смешивание в OpenGL ES достаточно просто. Нужно вызвать всего два метода:

```
gl.glEnable(GL10.GL_BLEND);
gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
```

Вызов первого метода должен быть вам уже знаком: он просто сообщает OpenGL ES, что тот должен применить альфа-смешивание ко всем треугольникам, которые мы визуализируем с данного момента. Второй метод немного сложнее. Он определяет, как должны сочетаться исходный и конечный цвета. В главе 3 мы обсуждали, что сочетание исходного и конечного цветов управляется простым уравнением смешивания. Метод `glBlendFunc()` просто сообщает OpenGL ES, какой тип уравнения использовать. Предыдущие параметры описывают, что мы хотим смешивать исходный и конечный цвета так, как это указано в уравнении из главы 3. Аналогичным образом `Canvas` выполняет смешивание с растровыми рисунками `Bitmap`.

Смешивание в OpenGL ES — мощный и одновременно сложный механизм. Нас пока не интересуют подробности. Просто используйте приведенную выше функцию смешивания везде, где хотите смешать треугольники с информацией из фреймбуфера, так же, как мы смешивали `Bitmap` с `Canvas`.

Второй вопрос касается того, откуда поступают исходный и конечный цвета. Последнее достаточно просто объяснить: это цвет пиксела в фреймбуфере, который мы собираемся заменить при отрисовке треугольника. Исходный цвет фактически является сочетанием двух цветов.

- *Цвет вершины.* Это цвет, который мы определяем с помощью `glColor4f()` для всех вершин или повершинно, добавляя свойство цвета к каждой вершине.
- *Цвет текстурных пикселов.* Как мы уже говорили, при отрисовке текстур применяются текселы — текстурные пикселы. Когда треугольник визуализируется вместе с ассоциированной с ним текстурой, OpenGL ES смешивает цвета текселов с цветами вершины для каждого пиксела треугольника.

Так что если треугольник не ассоциируется с определенной текстурой, исходный цвет для смешивания будет равен цвету вершины. Если треугольник ассоциируется с текстурой, исходный цвет для каждого пиксела треугольника представляет собой смесь цвета вершины и цвета текстурного пиксела. Мы можем определить, как сочетаются цвета вершины и текселов, используя метод `glTexEnv()`. По умолчанию цвет вершины *модулируется* цветом текселя. Это означает, что два цвета умножаются друг на друга покомпонентно (вершина $r \cdot$ тексел r и т. д.) Для всех случаев, описанных в этой книге, такой метод подходит оптимально, так что не будем углубляться в работу с `glTexEnv()`. Существует также несколько особенных ситуаций, в которых может потребоваться изменить принцип смешивания цвета вершины и текселов. Что касается `glBlendFunc()`, мы игнорируем детали и воспользуемся установками, заданными по умолчанию.

Когда мы загружаем изображение текстуры, у которого нет альфа-канала, OpenGL ES автоматически принимает, что значение альфа для каждого пиксела равно 1. Если мы загружаем изображение в формате RGBA8888, OpenGL ES будет использовать для смешивания предоставленные альфа-значения.

Для цвета вершин мы всегда определяем альфа-компонент отдельно или с помощью `glColor4f()`, где последний аргумент является альфа-значением или с помощью определения четырех компонентов для каждой вершины, где опять-таки последний компонент является альфа-значением.

Применим полученные знания на практике, обратившись к небольшому примеру. Мы хотим нарисовать Боба дважды: один раз — используя изображение `bobrgb888.png`, в котором нет альфа-канала на пиксел, а второй раз — применяя изображение `bobargb8888.png`, содержащее информацию в альфа-канале. Обратите внимание, что формат PNG сохраняет пикселы в формате ARGB8888, а не в RGBA8888. К счастью, метод `GLUtils.texImage2D()`, который мы используем для загрузки данных о изображении текстуры, автоматически выполнит преобразование. В листинге 7.11 дан код нашего небольшого эксперимента с использованием классов `Texture` и `Vertices`.

Листинг 7.11. Фрагмент из BlendingTest.java: смешивание в действии

```
class BlendingScreen extends Screen {
    GLGraphics glGraphics;
    Vertices vertices;
    Texture textureRgb;
    Texture textureRgba;

    public BlendingScreen(Game game) {
        super(game);
        glGraphics = ((GLGame)game).getGLGraphics();

        textureRgb = new Texture((GLGame)game, "bobrgb888.png");
        textureRgba = new Texture((GLGame)game, "bobargb8888.png");

       vertices = new Vertices(glGraphics, 8, 12, true, true);       float[] rects
= new float[] {
                100, 100, 1, 1, 1, 0.5f, 0, 1,
                228, 100, 1, 1, 1, 0.5f, 1, 1,
                228, 228, 1, 1, 1, 0.5f, 1, 0,
                100, 228, 1, 1, 1, 0.5f, 0, 0,

               100, 300, 1, 1, 1, 1, 0, 1,                228, 300, 1, 1, 1, 1, 1,
1,                228, 428, 1, 1, 1, 1, 1, 0,                100, 428, 1, 1, 1, 1,
0, 0
        };
        vertices.setVertices(rects, 0, rects.length);
        vertices.setIndices(new short[] {0, 1, 2, 2, 3, 0,
                                         4, 5, 6, 6, 7, 4 }, 0, 12);
    };
```

Наша небольшая реализация BlendingScreen содержит один экземпляр класса Vertices, где мы будем хранить два прямоугольника, а также два экземпляра класса Texture — один, включающий в себя RGBA8888-изображение Боба, и второй, содержащий RGB888-версию Боба. Мы загружаем обе текстуры из файлов bobrgb888.png и bobargb8888.png в конструктор и пользуемся классами Texture и GLUtils.texImag2D(), которые переводят ARGB8888 PNG в RGBA8888, как того требует OpenGL ES. Затем определяем наши вершины и индексы. Первый прямоугольник, имеющий четыре вершины, ассоциируется с текстурой Боба RGB888. Второй прямоугольник ассоциируется с версией Боба RGBA8888 и визуализируется на 200 единиц выше прямоугольника RGB888. Обратите внимание, что все вершины первого прямоугольника имеют цвет (1; 1; 1; 0,5f), а вершины второго прямоугольника имеют цвет (1; 1; 1; 1).

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glClearColor(1,0,0,1);
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    gl.glMatrixMode(GL10.GL_PROJECTION);
```

```
        gl.glLoadIdentity();
        gl.glOrthof(0, 320, 0, 480, 1, -1);

        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);

        gl.glEnable(GL10.GL_TEXTURE_2D);
        textureRgb.bind();
        vertices.draw(GL10.GL_TRIANGLES, 0, 6 );

        textureRgba.bind();
        vertices.draw(GL10.GL_TRIANGLES, 6, 6 );
    }
```

В методе `present()` очищаем экран, заполняя его красным цветом, и задаем матрицу проекции, как делали это раньше. Затем включаем альфа-смешивание и задаем соответствующее уравнение. Наконец, включаем нанесение текстуры и визуализируем два прямоугольника. Первый прямоугольник визуализируется с текстурой RGB888, а второй — с текстурой RGBA8888. Сохраняем оба прямоугольника в одном и том же экземпляре класса `Vertices` и используем смещение вместе с методами `vertices.draw()`. На рис. 7.16 показан результат нашей работы.

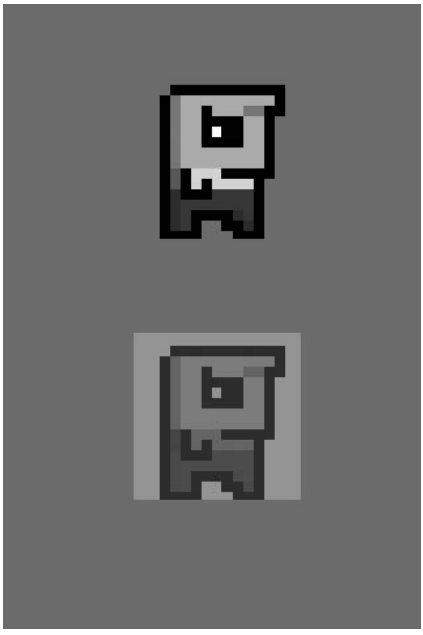

Рис. 7.16. Боб: смешивание цвета вершин *(сверху)* и смешивание текстур *(снизу)*

В случае с RGB888-Бобом смешивание происходит с примнением альфа-значений цветов каждой вершины. Поскольку мы установили их как 0,5f, Боб на 50 % прозрачный.

В случае с RGBA8888-Бобом альфа-значения цветов каждой вершины равны 1. Тем не менее, поскольку фоновые пикселы текстуры имеют альфа-значение 0 и поскольку цвета вершин и текселы модулируются, фон в этом варианте Боба не виден.

Если бы мы также установили альфа-значение цветов для каждой вершин 0,5f, то сам Боб тоже был бы на 50 % прозрачный, как и его копия в нижней части экрана. На рис. 7.17 показано, как бы это выглядело.

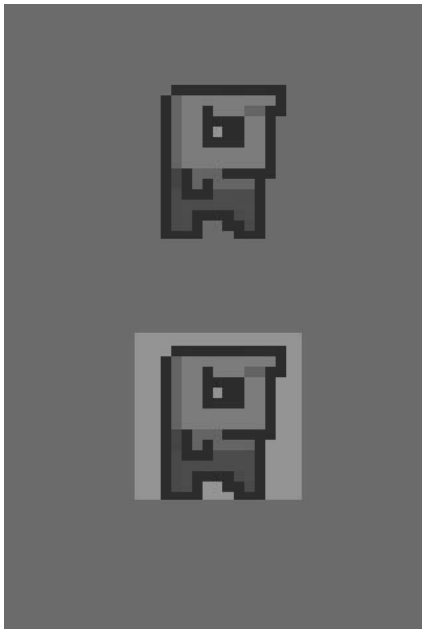

Рис. 7.17. Альтернативная версия RGBA8888 Боба, где альфа-значение, примененное поочередно к каждой из вершин, равно 0,5f *(вверху)*

Вот в общем-то и все, что нам надо знать о смешивании в 2D в OpenGL ES.

Однако есть еще один очень важный момент, на который я бы хотел обратить ваше внимание: смешивание потребляет очень много памяти! Не злоупотребляйте им. Современные мобильные графические процессоры не всегда хорошо справляются со смешиванием большого количества пикселов. Используйте смешивание только тогда, когда без него не обойтись.

Другие примитивы: точки, линии, полоски и конусы

Когда я говорил, что OpenGL ES — это огромная машина для визуализации треугольников, я немного недоговаривал. На самом деле OpenGL ES может также визуализировать точки и линии. Более того, они тоже описываются с помощью

вершин, и, как следствие, все вышеупомянутое применимо и к ним (текстуры, цвета для вершин и т. д.). Все, что нам надо сделать, чтобы визуализировать эти примитивы, — использовать при вызове `glDrawArrays()/glDrawElements()` что-то кроме `GL10.GL_TRIANGLES`. Эти примитивы также поддаются визуализации с индексированием, хотя это и несколько излишне (как минимум, в случае с точками). На рис. 7.18 показан список всех типов примитивов, которые нам предлагает OpenGL ES.

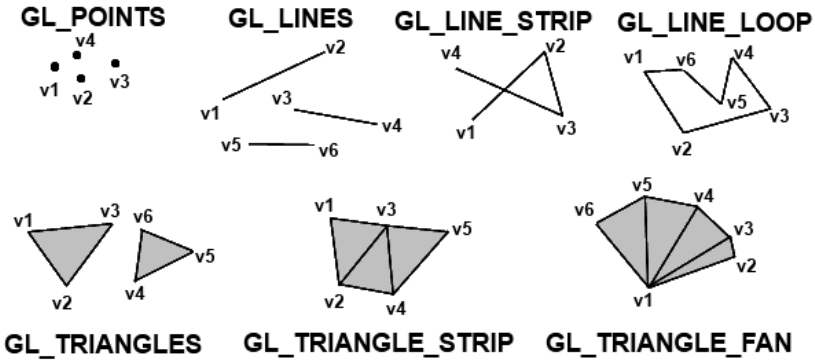

Рис. 7.18. Все примитивы, которые может визуализировать OpenGL ES

Кратко рассмотрим все эти примитивы.

- *Точка*. В случае с точкой каждая вершина — это отдельный примитив.
- *Линия*. Состоит из двух вершин. Как и в случае с треугольниками, достаточно иметь $2 \cdot n$ вершин, чтобы описать n линий.
- *Ломаная линия*. Все вершины воспринимаются как принадлежащие одной длинной линии.
- *Замкнутая ломаная линия*. Похожа на ломаную линию, единственная разница состоит в том, что OpenGL ES автоматически нарисует ломаную линию от последней вершины к первой.
- *Треугольник*. Он нам уже знаком. Каждый треугольник состоит из трех вершин.
- *Лента треугольников*. Вместо описания трех вершин мы просто описываем количество треугольников + одна вершина. Затем OpenGL ES создаст первый треугольник из вершин (`v1`; `v2`; `v3`), следующий треугольник из вершин (`v2`; `v3`; `v4`) и т. д.
- *Веер треугольников*. Веер имеет одну основную вершину (`v1`), которая является общей для всех треугольников. Первый треугольник будет (`v1`; `v2`; `v3`), следующий (`v1`; `v3`; `v4`) и т. д.

Ленты и вееры треугольников чуть менее гибки по сравнению с обычными списками треугольников. Однако они могут ускорить работу программы, поскольку нам придется умножать на матрицу проекции и модельно-видовую матрицу меньшее количество вершин. В нашем коде будем работать со списком треугольн-

ников, поскольку их проще использовать, а применение индексов поможет нам достичь довольно высокой производительности.

Точки и линии в OpenGL ES немного странные. Когда мы используем идеальную пиксельную ортогональную проекцию (например, когда разрешение экрана равно 320×480 пикселов и наш `glOrthof()` применяет такие же значения), мы по-прежнему не получаем идеальную пиксельную визуализацию. Местоположения вершин точки и линии должны быть смещены на 0,375f из-за так называемого правила ромбического выхода (diamond exit). Помните об этом, если вы хотите получить идеальную визуализацию точек и линий. Мы уже видели что-то похожее, когда рассматривали треугольники. Тем не менее, принимая во внимание, что мы обычно рисуем треугольники в 2D, мы вряд ли встретимся с этой проблемой.

Допустим, все, что вам нужно, — визуализировать примитивы, отличные от `GL10.GL_TRIANGLES`. В этом случае используйте одну из констант с рис. 7.18. Далее мы рассмотрим пример программы. В большинстве случаев будем применять список треугольников, особенно работая с графическим программированием 2D.

Теперь изучим в еще одну вещь, которую нам предлагает OpenGL ES: модельно-видовую матрицу.

2D-преобразования: матрица «Модель — представление»

До этого мы описывали статическую геометрию в виде списка треугольников. Здесь не было ни передвижения, ни поворотов, ни изменения масштаба. Однако даже когда сами параметры вершины оставались прежними (например, не изменялись ширина и высота прямоугольника, состоящего из двух треугольников вместе с координатами текстуры и цвета), нам по-прежнему приходилось создавать копию вершин, если мы хотели нарисовать такой же прямоугольник в другом месте. Посмотрите снова на листинг 7.11 и пока не учитывайте цветовые атрибуты вершин. Два прямоугольника различаются только значением y-координаты — на 200 единиц. Если бы мы могли передвигать эти вершины без фактического изменения их значений, мы могли бы описать прямоугольник Боба только один раз и просто отрисовывать его в разных местах. И это как раз то, для чего мы будем использовать матрицу «Модель — представление».

Пространство мира и модели

Чтобы понять, как это работает, нам нужно воспринимать ортографический отображаемый конус немного по-другому. Он располагается в особой системе координат, которая называется пространством мира. Это пространство, куда в итоге будут попадать все наши вершины.

До этого момента мы описывали местоположение всех вершин в абсолютной системе координат, относящейся к области мира, с которым мы работаем (сравните с рис. 7.5). На самом деле мы хотим сделать определение местоположения

независимым от системы координат области мира. Этого можно достичь, приписав каждой из наших моделей (например, прямоугольник Боба, космический корабль и т. д.) свою отдельную систему координат.

Это то, что мы обычно называем областью модели, системой координат, внутри которой мы описываем местоположение вершин модели. На рис. 7.19 показана эта концепция в 2D. Это также верно для 3D (просто добавьте ось *z*).

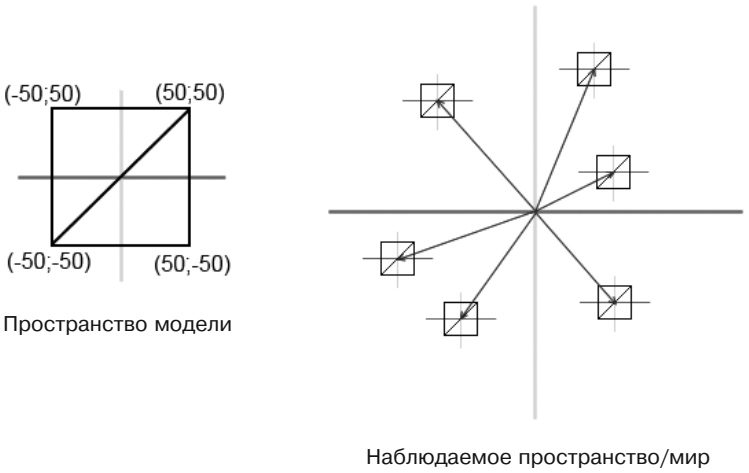

Рис. 7.19. Описание нашей модели в области модели, повторное использование и визуализация в разных местах области мира

На рис. 7.19 изображена одна модель, описанная с помощью экземпляра класса `Vertices`, к примеру, следующим образом:

```
Vertices vertices = new Vertices(glGraphics, 4, 12, false, false); vertices.
setVertices(new float[] { -50, -50,
                                  50, -50,
                                  50,  50,
                                 -50,  50 }, 0, 8);
vertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);
```

В данном контексте мы пока опустим координаты цвета и текстуры. Теперь, когда мы визуализируем эту модель без каких-либо последующих изменений, она будет размещена в начале координат мира на конечном изображении. Если мы хотим визуализировать ее в другом месте, например, чтобы ее центр был в точке (200; 300) области мира, мы можем повторно описать положение вершин следующим образом:

```
vertices.setVertices(new float[] { -50 + 200, -50 + 300,
                                    50 + 200, -50 + 300,
                                    50 + 200,  50 + 300,
                                   -50 + 200,  50 + 300 }, 0, 8);
```

При следующем вызове `vertices.draw()` модель будет визуализирована с центром в точке (200; 300). Но так работать немного неудобно, не правда ли?

Снова матрицы

Помните, мы вкратце уже говорили о матрицах? Мы обсуждали, как матрицы могут задавать такие трансформации, как перемещение, поворот и изменение размеров. Проекционная матрица, которую мы используем для переноса наших вершин на плоскость проекции, задает особый тип преобразования: проекцию.

Матрицы имеют определяющее значение при решении нашей предыдущей проблемы, помогая справиться с ней более аккуратно. Вместо того чтобы вручную перемещать вершины, заново описывая их местоположение, мы просто устанавливаем матрицу, которая кодирует перемещение. Поскольку матрица проекции OpenGL ES уже занята матрицей ортогональной проекции, которую мы определили через `glOrthof()`, мы будем использовать другую матрицу OpenGL ES: матрицу «Модель — представление». Вот как можно визуализировать нашу модель, перемещая ее вместе с началом координат в наблюдаемом пространстве:

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();
gl.glTranslatef(200, 300, 0);
vertices.draw(GL10.GL_TRIANGLES, 0, 6);
```

Однако сначала нужно сообщить OpenGL ES, какой матрицей мы хотим управлять. В нашем случае это матрица «Модель — представление», которая определена константой `GL10.GL_MODELVIEW`. Затем мы проверяем, установлена ли единичная матрица в качестве модельно-видовой. Фактически мы просто перезаписываем все, что уже было, — просто очищаем матрицу. Обратите внимание на вызов, описанный далее, — здесь происходит самое интересное.

Метод `glTranslatef()` принимает три аргумента: перемещение по осям x, y и z. Поскольку мы хотим, чтобы начало координат модели было перемещено в точку (200; 300) наблюдаемого пространства мира, мы задаем перемещение на 200 единиц по оси x и перемещение на 300 единиц по оси y. Поскольку мы работаем в 2D, просто игнорируем ось z и устанавливаем компонент перемещения равным нулю. Мы не описывали z-координату для наших вершин, так что она по умолчанию будет равна нулю. Если прибавить к нулю ноль, все равно получится ноль, так что наши вершины останутся в плоскости xy.

С этого момента матрица «Модель — представление» OpenGL ES кодирует перемещение на (200; 300; 0), которое будет применено ко всем вершинам, проходящим через конвейер OpenGL ES. Если вы снова посмотрите на рис. 7.4, то увидите, что OpenGL ES просто перемножит каждую вершину сначала на матрицу «Модель — представление», а затем применит к вершинам проекционную матрицу. До сих пор матрица «Модель — представление» представляла собой единичную матрицу (настройка OpenGL ES по умолчанию). Следовательно, она не влияла на наши вершины. Наш небольшой вызов `glTranslatef()` спровоцирует эти изменения и переместит все вершины перед тем, как они будут спроецированы.

Конечно, все это делается динамически, а значения в экземпляре класса `Vertices` не изменяются вообще. Мы бы заметили какие-либо постоянные изменения в экземпляре класса `Vertices`, поскольку проекционная матрица уже бы его изменила.

Первый пример с использованием переноса

Для чего нам требуется перенос? Допустим, мы хотим визуализировать 100 копий Боба в разных местах мира. Более того, нам нужно, чтобы они двигались по экрану и изменяли направление каждый раз, когда столкнутся с границами экрана (или, вернее, границами нашей параллельной проекции конуса отображения, которые совпадают с границами экрана). Это можно было бы сделать с помощью одного большого экземпляра класса `Vertices`, который содержит вершины 100 прямоугольников — одного для каждого Боба — и пересчитывает местоположение вершин для каждого кадра. Проще было бы использовать небольшой экземпляр класса `Vertices`, содержащий только один прямоугольник (модель Боба) и применяющий его многократно, перенося его по матрице «Модель — представление». Опишем нашу модель Боба:

```
Vertices bobModel = new Vertices(glGraphics, 4, 12, false, true); bobModel.
setVertices(new float[] { -16, -16, 0, 1,
                                    16, -16, 1, 1,
                                    16,  16, 1, 0,
                           -16,  16, 0, 0, }, 0, 8); bobModel.setIndices(new short[]
{0, 1, 2, 2, 3, 0}, 0, 6);
```

Размер каждого Боба — 32×32 единицы. Мы также ассоциируем его с текстурой (будем использовать `bobrgb888.png`, чтобы видеть границы каждого Боба).

Боб становится классом

Опишем простой класс `Bob`, который будет отвечать за местоположение Боба и его перемещение в текущем направлении, основываясь на дельте времени, так же, как мы передвигали мистера Нома (единственная разница состоит в том, что мы больше не передвигаемся по сетке). Метод `update()` будет следить за тем, чтобы Боб не вышел за видимые границы. В листинге 7.12 показан класс `Bob`.

Листинг 7.12. `Bob.java`

```
package com.badlogic.androidgames.glbasics;

import java.util.Random;

class Bob {
    static final Random rand = new Random();
    public float x, y;
    float dirX, dirY;

    public Bob() {
        x = rand.nextFloat() * 320;
        y = rand.nextFloat() * 480;
        dirX = 50;
        dirY = 50;
    }

    public void update(float deltaTime) {
```

```
        x = x + dirX * deltaTime;
        y = y + dirY * deltaTime;

        if (x < 0) {
            dirX = -dirX;
            x = 0;
        }

        if (x > 320) {
            dirX = -dirX;
            x = 320;
        }

        if (y < 0) {
            dirY = -dirY;
            y = 0;
        }

        if (y > 480) {
            dirY = -dirY;
            y = 480;
        }
    }
}
```

Каждый Боб помещается в случайное место в созданном нами мире. Все Бобы будут изначально двигаться в одинаковом направлении: 50 единиц направо и 50 единиц вверх в секунду (поскольку происходит умножение на `deltaTime`). В методе `update()` просто перемещаем Боба в текущем направлении, основываясь на времени, а затем проверяем, покинул ли он границы конуса отображения. Если покинул, меняем направление и проверяем, находится ли он по-прежнему внутри конуса отображения.

Теперь предположим, что мы создаем 100 Бобов следующим образом:

```
Bob[] bobs = new Bob[100];
for(int i = 0; i < 100; i++) {
    bobs[i] = new Bob();
}
```

Для визуализации каждого Боба делаем следующее (допустим, мы уже очистили экран, установили матрицу проекции и привязали текстуру):

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
for(int i = 0; i < 100; i++) {
    bob.update(deltaTime);
    gl.glLoadIdentity();
    gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
    bobModel.render(GL10.GL_TRIANGLES, 0, 6);
}
```

Все достаточно просто, не правда ли? Для каждого Боба вызываем его метод `update()`, который будет перемещать его и проверять, остался ли он в границах нашего маленького мира. Затем загружаем единичную матрицу в модельно-видовую матрицу OpenGL ES, чтобы у нас всегда было чистое состояние. Затем используем текущие *x*- и *y*-координаты Боба в вызове `glTransltef()`. Когда мы затем визуализируем модель Боба в следующем вызове, все вершины будут перенесены относительно текущего месторасположения Боба — именно этого мы и добивались.

Все вместе

Сделаем общий пример (листинг 7.13).

Листинг 7.13. BobTest.java: 100 Moving Bobs!

```
package com.badlogic.androidgames.glbasics;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Screen;
import com.badlogic.androidgames.framework.gl.FPSCounter;
import com.badlogic.androidgames.framework.gl.Texture;
import com.badlogic.androidgames.framework.gl.Vertices;
import com.badlogic.androidgames.framework.impl.GLGame;
import com.badlogic.androidgames.framework.impl.GLGraphics;

public class BobTest extends GLGame {

    @Override
    public Screen getStartScreen() {
        return new BobScreen(this);
    }

    class BobScreen extends Screen {
        static final int NUM_BOBS = 100;
        GLGraphics glGraphics;
        Texture bobTexture;
        Vertices bobModel;
        Bob[] bobs;
```

Наш класс `BobScreen` содержит текстуру `Texture` (загруженную из `bobrbg888.png`), экземпляр класса `Vertices`, который содержит модель Боба (простой текстурированный прямоугольник) и массив экземпляров класса Боб. Мы также определяем константу `NUM_BOBS`, чтобы можно было изменять количество Бобов на экране.

```
        public BobScreen(Game game) {
            super(game);
            glGraphics = ((GLGame)game).getGLGraphics();

            bobTexture = new Texture((GLGame)game, "bobrgb888.png");

            bobModel = new Vertices(glGraphics, 4, 12, false, true);
            bobModel.setVertices(new float[] { -16, -16, 0, 1,
```

```
                                        16, -16, 1, 1,
                                        16,  16, 1, 0,
                                       -16,  16, 0, 0, }, 0, 16);
        bobModel.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);

        bobs = new Bob[100];
        for(int i = 0; i < 100; i++) {
            bobs[i] = new Bob();
        }
    }
```

Конструктор просто загружает текстуру и модель и приписывает константу `NUM_BOBS` экземплярам класса `Bob`.

```
@Override
    public void update(float deltaTime) {
        game.getInput().getTouchEvents();
        game.getInput().getKeyEvents();

        for(int i = 0; i < NUM_BOBS; i++) {
            bobs[i].update(deltaTime);
        }
    }
```

В методе `update()` Бобы обновляются. Мы также проверяем, очищены ли буферы событий ввода.

```
@Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glClearColor(1,0,0,1);
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glOrthof(0, 320, 0, 480, 1, -1);

        gl.glEnable(GL10.GL_TEXTURE_2D);
        bobTexture.bind();

        gl.glMatrixMode(GL10.GL_MODELVIEW);
        for(int i = 0; i < NUM_BOBS; i++) {
            gl.glLoadIdentity();
            gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
            gl.glRotatef(45, 0, 0, 1);
            gl.glScalef(2, 0.5f, 0);
            bobModel.draw(GL10.GL_TRIANGLES, 0, 6);
        }
    }
```

В методе `render()` очищаем экран, устанавливаем проекционную матрицу, активируем текстурирование и привязываем текстуру Боба. Последние две строки отвечают за визуализацию каждого экземпляра класса `Bob`. Поскольку OpenGL ES запоминает их состояния, нам нужно устанавливать активную матрицу только один раз

(в этом случае мы будем изменять матрицу «Модель — представление» в оставшейся части кода). Затем мы проходим по всем Бобам, устанавливаем матрицу «Модель — представление», основываясь на текущем положении Боба, и визуализируем модель, которая будет автоматически смещена по матрице «Модель — представление».

```
        @Override
        public void pause() {
        }

        @Override
        public void resume() {
        }

        @Override
        public void dispose() {
        }
    }
}
```

Вот и все. Лучше всего то, что мы снова использовали паттерн MVC, который уже известен нам из работы с «Мистером Номом». Он действительно очень удобен при программировании игр. Логическая сторона Боба полностью отделена от его внешнего вида, что весьма неплохо, так как мы можем легко заменить его вид чем-то посложнее. На рис. 7.20 показан результат работы нашей программы после нескольких секунд работы.

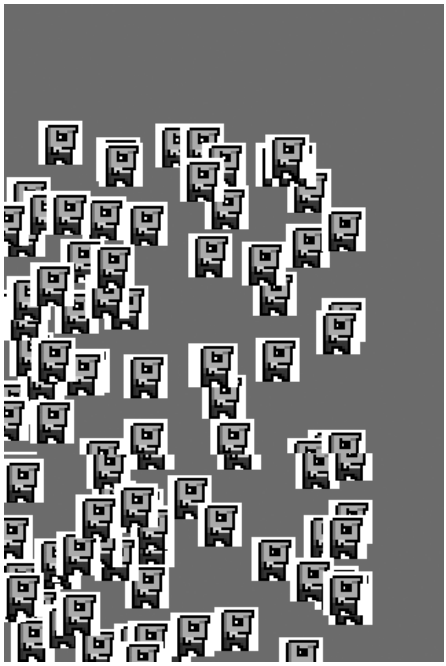

Рис. 7.20. Очень много Бобов

Но на этом интересности с преобразованиями не заканчиваются. Помните, несколько страниц назад я упоминал, что мы еще поговорим о вращении и масштабировании?

Другие преобразования

Кроме метода `glTranslatef()` OpenGL ES также предлагает два других метода для преобразований: `glRotatef()` и `glScalef()`.

Вращение

Вот сигнатура метода `glRotatef()`:

```
GL10.glRotatef(float angle, float axisX, float axisY, float axisZ);
```

Первый параметр — это угол в градусах, на который мы хотим повернуть вершины. Но что же значат остальные параметры?

Когда мы что-то поворачиваем, делаем это по какой-либо оси. Что такое ось? Мы уже знаем три оси: x, y и z. Мы можем выразить эти три оси в виде векторов. Положительная ось x будет описываться как (1; 0; 0), положительная ось y — (0; 1; 0), а положительная ось z — (0; 0; 1). Как вы можете видеть, вектор фактически кодирует направление, в нашем случае в 3D-пространстве. Направление Боба — это тоже вектор, но в 2D-пространстве. Векторы также кодируют местоположение, например место Боба в двухмерном пространстве.

Чтобы описать ось, по которой будем поворачивать модель Боба, нужно вернуться к 3D-пространству. На рис. 7.21 показана модель Боба (с наложенной текстурой для ориентации), как она описана в 3D с помощью предыдущего кода.

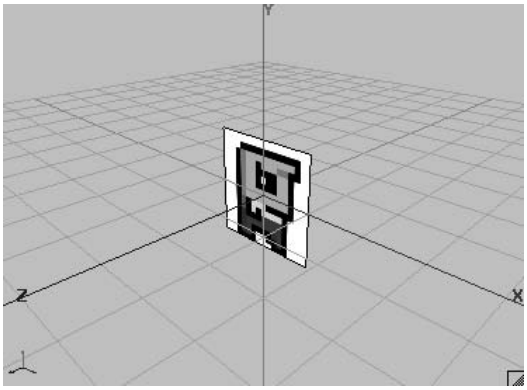

Рис. 7.21. Боб в 3D

Поскольку мы не описали z-координаты для вершин Боба, он находится в xy-пространстве нашего 3D-мира, который фактически является пространством модели. Если мы захотим повернуть Боба, можем сделать это по любой оси (x, y, z) или даже

по каким-то невообразимым осям, например с координатами (0,75; 0,75; 0,75). Тем не менее в случае с нашим графическим программированием в двух измерениях целесообразно вращать Боба в плоскости *xy*. Следовательно, мы будем использовать положительную ось *z*, которая может быть описана как (0; 0; 1) в качестве оси вращения. Поворот будет проходить против часовой стрелки по оси *z*. Вызов `glRotatef()` приведет к тому, что вершины модели Боба будут повернуты так, как это показано на рис. 7.22:

```
gl.glRotatef(45, 0, 0, 1);
```

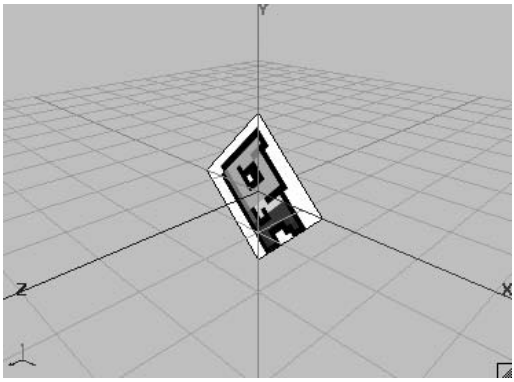

Рис. 7.22. Боб, повернутый по оси *z* на 45°

Масштабирование

Мы также можем изменить размеры Боба с помощью `glScalef` следующим образом:

```
glScalef(2, 0.5f, 1);
```

Исходя из начальной позиции Боба, мы получим ориентацию, как на рис. 7.23.

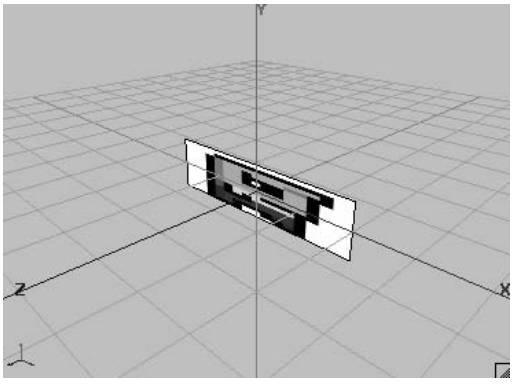

Рис. 7.23. Боб, размеры которого изменены в 2 раза по оси *x* и в 0,5 раза по оси *y*

Сочетание преобразований

Мы можем сочетать эффекты множества матриц, перемножая их, чтобы получить новую матрицу. Фактически все это делают методы `glTranslatef()`, `glScalef()`, `glRotatef()` и `glOrthof()`. Они умножают текущую активную матрицу на временную матрицу, которую создают внутрисистемно, основываясь на параметрах, которые мы им передаем. Соединим поворот и изменение размеров Боба:

```
gl.glRotatef(45, 0, 0, 1);
gl.glScalef(2, 0.5f, 1);
```

В реультате модель Боба будет выглядеть, как на рис. 7.24 (помните, мы по-прежнему в пространстве модели).

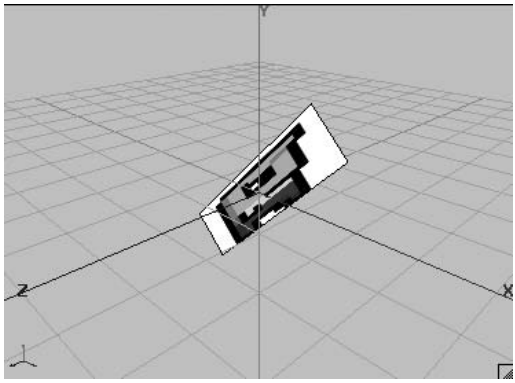

Рис. 7.24. Боб, сначала измененный в размере, а затем повернутый

Что случится, если мы применим изменения в другом порядке:

```
gl.glScalef(2, 0.5, 0);
gl.glRotatef(45, 0, 0, 1)
```

На рис. 7.25 показан результат.

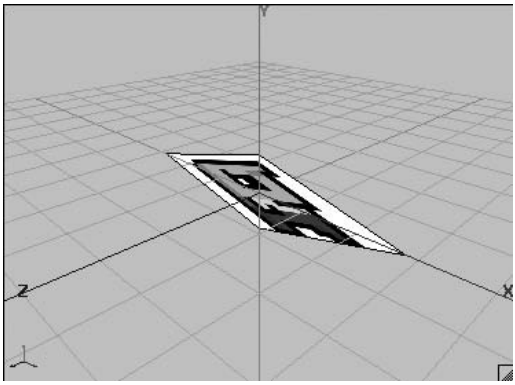

Рис. 7.25. Боб, сначала повернутый, а затем измененный в размерах

Ого, это не тот Боб, к которому мы привыкли. Что же здесь произошло? Исходя из кода, логично предположить, что рис. 7.24 и 7.25 будут выглядеть одинаково. В первом фрагменте мы сначала производим поворот, а затем изменяем размеры Боба, правильно?

Нет, неправильно. Порядок, в котором преобразования применяются к модели, соответствует порядку, в котором OpenGL ES перемножает матрицы. Последняя матрица, на которую мы умножаем текущую активную матрицу, будет первой, которая будет применена к вершинам.

Если мы хотим изменить размер, повернуть и перенести Боба именно в таком порядке, нам нужно вызвать методы следующим образом:

```
glTranslatef(bobs[i].x, bobs[i].y, 0);
glRotatef(45, 0, 0, 1);
glScalef(2, 0.5f, 1);
```

Изменим цикл в нашем методе `BobScreen.present()` вот так:

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
for(int i = 0; i < NUM_BOBS; i++) {
    gl.glLoadIdentity();
    gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
    gl.glRotatef(45, 0, 0, 1);
    gl.glScalef(2, 0.5f, 0);
    bobModel.draw(GL10.GL_TRIANGLES, 0, 6); }
```

Результат будет выглядеть, как на рис. 7.26.

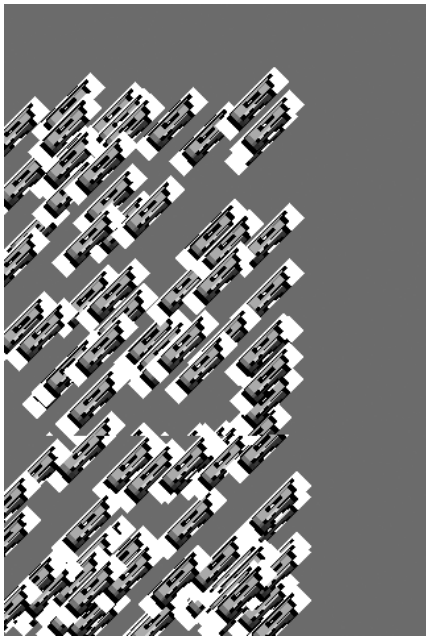

Рис. 7.26. Сто Бобов, измененных в размере, повернутых и перемещенных (в таком порядке)

Когда я только начинал работать с OpenGL, я постоянно путал порядок операций с матрицами. Чтобы запомнить, как это правильно делать, я изобрел мнемоническое слово ПООПЕП: последним описан, первым применен (да, мне есть еще над чем работать в мнемонике).

Самый простой способ научиться правильно работать с модельно-видовыми преобразованиями — использовать их как можно чаще. Я предлагаю вам взять исходный файл `BobTest.java` и немного поизменять внутренний цикл для того, чтобы посмотреть различные эффекты.

Обратите внимание, что для визуализации каждой модели вы можете описать столько изменений, сколько захотите. Добавьте повороты, перемещения и изменения размеров. Поиграйте.

После рассмотрения этого примера мы теперь знаем практически все, что нужно, об OpenGL ES для написания игр 2D. Или нет?

Оптимизация производительности

Когда мы запускаем подобный пример на мощном устройстве второго поколения, как, например, Droid или Nexus One, все работает быстро и правильно. Если мы запустим его на Hero, все начнет тормозить, что весьма неприятно. Но разве мы с вами не говорили, что OpenGL ES — это как раз то, что нам нужно для быстрой визуализации графики? Да, это действительно так. Но только, если мы все делаем так, как этого хочет OpenGL ES.

Измерение частоты кадров

`BobTest` — это отличный материал для того, чтобы начать оптимизацию. Перед тем как мы приступим, нам нужно сначала оценить работу. Осмотр вручную (то есть простая оценка скорости работы на глаз) недостаточно точен. Лучший способ измерить то, как быстро работает программа, — подсчитать количество кадров, которые мы визуализируем в секунду. Помните, в главе 3 мы говорили о вертикальной синхронизации, или vsync. Эта функция есть на всех устройствах с Android, которые сейчас присутствуют на рынке. Она сокращает максимальное количество кадров в секунду, которые мы можем получить, до 60. Как мы знаем, подобная частота кадров вполне подходит для нашего кода.

ПРИМЕЧАНИЕ

Хотя неплохо было бы иметь частоту 60 кадров в секунду, на самом деле этого не так просто достичь. У таких устройств, как Nexus One и Droid, слишком много пикселов, которые надо заполнить, даже если мы просто очищаем экран. Нас вполне устроит, если игра будет визуализировать мир на частоте более 30 кадров в секунду. Хотя, конечно, и лишние кадры не помешают.

Напишем небольшой вспомогательный класс, который считает кадры в секунду и периодически выводит это значение. В листинге 7.14 показан код класса `FPSCounter`.

Листинг 7.14. FPSCounter.java: считаем фреймы и записываем их в LogCat каждую секунду

```
package com.badlogic.androidgames.framework.gl;

import android.util.Log;

public class FPSCounter {
    long startTime = System.nanoTime();
    int frames = 0;

    public void logFrame() {
        frames++;
        if(System.nanoTime() - startTime >= 1000000000) {
            Log.d("FPSCounter", "fps: " + frames);
            frames = 0;
            startTime = System. nanoTime();
        }
    }
}
```

Мы можем поместить экземпляр этого класса в наш класс BobScreen и один раз вызвать метод logFrame() в методе BobScreen.present(). Я только что сделал это, и вот результат для Hero (работает на Android 1.5), Droid (работает на Android 2.2) и Nexus One (работает на Android 2.2.1):

```
Hero:
12-10 03:27:05.230: DEBUG/FPSCounter(17883): fps: 22
12-10 03:27:06.250: DEBUG/FPSCounter(17883): fps: 22
12-10 03:27:06.820: DEBUG/dalvikvm(17883): GC freed 21818 objects / 524280 bytes
in 132ms
12-10 03:27:07.270: DEBUG/FPSCounter(17883): fps: 20
12-10 03:27:08.290: DEBUG/FPSCounter(17883): fps: 23

Droid:
12-10 03:29:44.825: DEBUG/FPSCounter(8725): fps: 39
12-10 03:29:45.864: DEBUG/FPSCounter(8725): fps: 38
12-10 03:29:46.879: DEBUG/FPSCounter(8725): fps: 38
12-10 03:29:47.879: DEBUG/FPSCounter(8725): fps: 39
12-10 03:29:48.887: DEBUG/FPSCounter(8725): fps: 40

Nexus One:
12-10 03:28:05.923: DEBUG/FPSCounter(930): fps: 43
12-10 03:28:06.933: DEBUG/FPSCounter(930): fps: 43
12-10 03:28:07.943: DEBUG/FPSCounter(930): fps: 44
12-10 03:28:08.963: DEBUG/FPSCounter(930): fps: 44
12-10 03:28:09.973: DEBUG/FPSCounter(930): fps: 44
12-10 03:28:11.003: DEBUG/FPSCounter(930): fps: 43
12-10 03:28:12.013: DEBUG/FPSCounter(930): fps: 44
```

С первого взгляда мы видим следующее:

- HTC Hero в два раза медленнее, чем Droid и Nexus One;
- Nexus One немного быстрее по сравнению с Droid;
- мы генерируем мусор на HTC Hero в нашем процессе (17883).

Последний пункт в этом списке не очень понятен. Мы запускаем один и тот же код на трех устройствах. Но мы не создаем никаких временных объектов ни в методе `present()`, ни в методе `update()`. Что же происходит на HTC Hero?

Любопытный случай с Hero на Android 1.5

Как оказалось, в Android 1.5 есть небольшой баг. Вообще-то это даже не баг, а просто пример неаккуратного программирования. Помните, мы используем NIO-буферы для наших вершин и индексов? Фактически они являются блоками памяти в нативной куче памяти. Каждый раз, когда мы вызываем `glVertexPointer()`, `glColorPointer()` или любой другой метод `glXXXPointer()`, OpenGL ES пытается получить адрес памяти нативной кучи буфера, чтобы найти вершины для перевода данных в видеопамять. Проблема на Android 1.5 заключается в том, что каждый раз, когда мы запрашиваем адрес памяти из буфера NIO, он генерирует временный объект `PlatformAddress`. Поскольку у нас множество вызовов методов `glXXXPointer()` и `glDrawElements()` (помните, последний переносит адрес из `ShortBuffer`), Android создает множество временных экземпляров класса `PlatformAdress`, и мы ничего не можем с этим поделать. (Вообще-то выход есть, но мы не будем его сейчас обсуждать.) Давайте просто учтем тот факт, что пользование буферами NIO в Android 1.5 доставляет массу проблем, и пойдем дальше.

Почему OpenGL ES-рендеринг такой медленный?

То, что HTC Hero медленнее, чем устройства второго поколения, не секрет. Тем не менее чип PowerVR в Droid чуть-чуть быстрее, чем чип Adreno в Nexus One, так что предыдущие результаты на первый взгляд достаточно странные. При дальнейшем исследовании мы можем, наверное, объяснить разницу не различной мощностью графического процессора, а тем, что вызываем множество методов OpenGL ES в каждом фрейме и эти методы занимают очень много памяти. Это значит, что они фактически вызывают C-код, который требует больше памяти, чем вызов метода Java на Dalvik. У Nexus One есть динамический компилятор, так что он имеет некоторое пространство для оптимизации. Поэтому давайте будем считать, что разница обусловлена динамическим компилятором. (что, возможно, не совсем правильно).

А теперь посмотрим, что в OpenGL ES оставляет желать лучшего:

- множество изменений состояний в каждом кадре (например, смешивание, включение/выключение нанесения текстуры и т. д.);

- большое количество изменений матриц в каждом кадре;
- множество операций привязывания текстур в каждом кадре;
- многократные изменения вершин, цвета и координат текстуры в каждом кадре.

Все эти проблемы фактически связаны с изменениями состояний. Почему они требуют так много памяти? Графический процессор работает как сборочная линия на фабрике. Пока первая линия обрабатывает поступающие товары, конечная линия заканчивает уже обработанные на всех предыдущих стадиях продукты. Попробуем провести параллель с автомобильным конвейером.

Продукция на производственной линии проходит через несколько состояний. Сначала применяются инструменты, с которыми работают специалисты завода, потом крепления, используемые для соединения частей машины, потом в ход идет краска, которой покрывают машины, и т. д. Конечно, настоящие автомобильные заводы имеют в своем распоряжении множество линий, но давайте предположим, что линия всего одна. В таком случае каждая стадия линии будет занята до тех пор, пока мы не изменим состояние. Когда мы изменяем один этап, линия будет продолжать работать, пока все собираемые машины не будут закончены. Только после этого мы можем перейти к другому состоянию и собирать машины нового цвета или с новыми креплениями.

Ключевой момент состоит в том, что вызов `glDrawElements()` или `glDrawArrays()` не выполняется сразу. Вместо этого команда помещается в буфер, который асинхронно обрабатывается графическим процессором. Это значит, что вызов методов рисования не блокируется. Поэтому вряд ли стоит измерять, сколько времени займет вызов `glDrawElements()`, поскольку сама работа может быть проделана в будущем.

Именно поэтому мы все измеряем в кадрах в секунду. Когда фреймбуферы меняются местами (да, и в OpenGL ES применяется двойная буферизация), OpenGL ES выполняет все отложенные операции.

Вновь сравним OpenGL ES с автомобилестроительным заводом. В то время как новые треугольники поступают в командный буфер с помощью вызова `glDrawElements()` или `glDrawArrays()`, конвейер графического процессора должен закончить визуализацию текущих треугольников из предыдущих вызовов (например, треугольник может находиться на стадии растеризации). Этим обусловлены следующие особенности.

- На замену актуальной привязанной текстуры уходит много памяти. Любые еще не обработанные, но использующие текстуру треугольники в командном буфере должны сначала быть визуализированы.
- Изменение вершин, цвета или текстурных координат также требует много памяти. Любые еще не визуализированные треугольники, расположенные в командном буфере и применяющие старые указатели, должны сначала быть визуализированы.
- Изменение состояния смешивания требует много памяти. Любые еще не визуализированные, но требующие/не требующие смешивания треугольники

в командном буфере, к которым должны быть применены старые матрицы, должны сначала быть визуализированы.

- Наконец, много памяти требуется на изменение модели-вида или проекционной матрицы. Любые находящиеся в командном буфере треугольники, которые пока не были обработаны и к которым должны применяться старые матрицы, придется отображать в первую очередь. Конвейер застынет.

Суть всего это проста — сократите изменения состояний до минимума.

Убираем ненужные изменения состояний

Рассмотрим метод `present()BobTesta`, чтобы узнать, где что можно урезать. Вот фрагмент кода (я добавил `FPSCounter`, а также `glRotatef()` и `glScalef()`):

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glClearColor(1,0,0,1);
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    gl.glOrthof(0, 320, 0, 480, 1, -1);

    gl.glEnable(GL10.GL_TEXTURE_2D);
    bobTexture.bind();

    gl.glMatrixMode(GL10.GL_MODELVIEW);
    for(int i = 0; i < NUM_BOBS; i++) {
        gl.glLoadIdentity();
        gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
        gl.glRotatef(45, 0, 0, 1);
        gl.glScalef(2, 0.5f, 1);
        bobModel.draw(GL10.GL_TRIANGLES, 0, 6);
    }
    fpsCounter.logFrame();
}
```

Первое, что мы можем сделать, — переместить вызовы `glViewport()` и `glClearColor()`, а также вызовы методов, которые устанавливают проекционную матрицу в метод `BobScreen.resume()`. Цвет очистки никогда не будет меняться, равно как область просмотра и матрица проекции. Вы спросите, почему не поместить код для установки всех постоянных состояний OpenGL, таких как область просмотра или матрица проекции, в конструктор `BobScreen`? Дело в потере контекста. Все изменения состояний OpenGL ES будут утеряны, а когда вызывается наш экранный метод `resume()`, мы знаем, что контекст, как и все утерянные состояния, был восстановлен. Мы также можем переместить в метод `resume()` вызов `glEnable()` и вызов привязки текстуры. Наконец, мы хотим, чтобы текстурирование было включено все время,

однако мы используем только одну текстуру для Боба. На всякий случай мы перемещаем texture.reload() в метод resume(), чтобы в случае потери контекста наши данные изображения текстуры были заново загружены. Вот как будут выглядеть измененные методы present() и resume():

```
@Override
public void resume() {
    GL10 gl = glGraphics.getGL();
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glClearColor(1, 0, 0, 1);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    gl.glOrthof(0, 320, 0, 480, 1, -1);

    bobTexture.reload();
    gl.glEnable(GL10.GL_TEXTURE_2D);
    bobTexture.bind();
}

@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

    gl.glMatrixMode(GL10.GL_MODELVIEW);
    for(int i = 0; i < NUM_BOBS; i++) {
        gl.glLoadIdentity();
        gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
        gl.glRotatef(45, 0, 0, 1);
        gl.glScalef(2, 0.5f, 0);
        bobModel.draw(GL10.GL_TRIANGLES, 0, 6);
    }

    fpsCounter.logFrame();
}
```

Применение этой «улучшенной» версии приводит к следующим результатам на трех устройствах:

```
Hero:
12-10 04:41:56.750: DEBUG/FPSCounter(467): fps: 23
12-10 04:41:57.770: DEBUG/FPSCounter(467): fps: 23
12-10 04:41:58.500: DEBUG/dalvikvm(467): GC freed 21821 objects / 524288 bytes in 133ms
12-10 04:41:58.790: DEBUG/FPSCounter(467): fps: 19
12-10 04:41:59.830: DEBUG/FPSCounter(467): fps: 23

Droid:
12-10 04:45:26.906: DEBUG/FPSCounter(9116): fps: 39
12-10 04:45:27.914: DEBUG/FPSCounter(9116): fps: 41
12-10 04:45:28.922: DEBUG/FPSCounter(9116): fps: 41
```

```
12-10 04:45:29.937: DEBUG/FPSCounter(9116): fps: 40

Nexus One:
12-10 04:37:46.097: DEBUG/FPSCounter(2168): fps: 43
12-10 04:37:47.127: DEBUG/FPSCounter(2168): fps: 45
12-10 04:37:48.147: DEBUG/FPSCounter(2168): fps: 44
12-10 04:37:49.157: DEBUG/FPSCounter(2168): fps: 44
12-10 04:37:50.167: DEBUG/FPSCounter(2168): fps: 44
```

Как видите, все устройства немного выиграли от наших оптимизаций. Конечно, эффект не так уж велик из-за того, что мы изначально вызываем все эти методы в начале фрейма, а ведь мы еще не приступили к треугольникам.

Уменьшение размера текстуры — выбираем меньше пикселов

Что еще можно изменить? Кое-что, что не так уж очевидно. Наши экземпляры класса `Bob` имеют размер 32×32 единицы. Мы используем плоскость проекции размером 320×480 единиц. На Hero получаем идеальную визуализацию с точностью до пиксела. На Nexus One или Droid одна единица в нашей системе координат будет занимать чуть меньше пиксела. В любом случае наша текстура фактически имеет размер 128×128 пикселов. Нам не нужно такое большое разрешение, уменьшим размер изображения текстуры `bobrgb888.png` до 32×32 пиксела. Назовем новое изображение `bobrgb888-32x32.png`. Используя эту меньшую по размеру текстуру, получим следующие значения кадровой частоты для каждого устройства:

```
Hero:
12-10 04:48:03.940: DEBUG/FPSCounter(629): fps: 23
12-10 04:48:04.950: DEBUG/FPSCounter(629): fps: 23
12-10 04:48:05.860: DEBUG/dalvikvm(629): GC freed 21812 objects / 524256 bytes in 134ms
12-10 04:48:05.990: DEBUG/FPSCounter(629): fps: 21
12-10 04:48:07.030: DEBUG/FPSCounter(629): fps: 24

Droid:
12-10 04:51:11.601: DEBUG/FPSCounter(9191): fps: 56
12-10 04:51:12.609: DEBUG/FPSCounter(9191): fps: 56
12-10 04:51:13.625: DEBUG/FPSCounter(9191): fps: 55
12-10 04:51:14.641: DEBUG/FPSCounter(9191): fps: 55

Nexus One:
12-10 04:48:18.067: DEBUG/FPSCounter(2238): fps: 53
12-10 04:48:19.077: DEBUG/FPSCounter(2238): fps: 56
12-10 04:48:20.077: DEBUG/FPSCounter(2238): fps: 53
12-10 04:48:21.097: DEBUG/FPSCounter(2238): fps: 54
```

Оказывается, на устройствах второго поколения разница весьма велика. Дело в том, что их графические процессоры совершенно не приспособлены к сканированию большого количества пикселов. Это же справедливо для выбора текселов

из текстуры и визуализации треугольников на экране. Скорость, на которой графические процессоры могут загружать текселы и визуализировать пикселы во фреймбуфере, называется *скоростью заполнения* (fill rate). Все графические процессоры второго поколения характеризуются крайне низкой скоростью заполнения, так что нужно стараться использовать как можно более маленькие текстуры (или наносить на наши треугольники только малую часть) и не визуализировать очень большие треугольники на экране. Мы также должны следить за наложением: чем меньше треугольники пересекаются, тем лучше.

ПРИМЕЧАНИЕ

Вообще-то наложение не такая уж и большая проблема таких графических процессоров, как PowerVR SGX 350 на Droid. Эти графические процессоры имеют специальный механизм отложенной визуализации частей, который при определенных условиях справляется с большинством наложений. Однако приходится учитывать и те пикселы, которые не будут отображаться на экране.

HTC Hero выиграл от уменьшения размера изображения текстуры совсем немного. В чем же дело?

Уменьшаем количество вызовов методов OpenGL ES/JNI

Прежде всего это могло произойти из-за большого количества вызовов OpenGL ES на кадр, когда мы визуализируем модель для каждого из Бобов. У нас происходит четыре матричные операции с каждым экземпляром Боба. Если нам не нужно поворачивание или изменение размеров, можем сократить количество вызовов до двух. Вот количество кадров в секунду, когда мы используем только `glLoadIdentity()` и `glTranslatef()` во внутреннем цикле:

```
Hero:
12-10 04:57:49.610: DEBUG/FPSCounter(766): fps: 27
12-10 04:57:49.610: DEBUG/FPSCounter(766): fps: 27
12-10 04:57:50.650: DEBUG/FPSCounter(766): fps: 28
12-10 04:57:50.650: DEBUG/FPSCounter(766): fps: 28
12-10 04:57:51.530: DEBUG/dalvikvm(766): GC freed 22910 objects / 568904 bytes in 128ms

Droid:
12-10 05:08:38.604: DEBUG/FPSCounter(1702): fps: 56
12-10 05:08:39.620: DEBUG/FPSCounter(1702): fps: 57
12-10 05:08:40.628: DEBUG/FPSCounter(1702): fps: 58
12-10 05:08:41.644: DEBUG/FPSCounter(1702): fps: 57

Nexus One:
12-10 04:58:01.277: DEBUG/FPSCounter(2509): fps: 54
12-10 04:58:02.287: DEBUG/FPSCounter(2509): fps: 54
12-10 04:58:03.307: DEBUG/FPSCounter(2509): fps: 55
12-10 04:58:04.317: DEBUG/FPSCounter(2509): fps: 55
```

Итак, мы немного улучшили работу HTC Hero, в свою очередь Droid и Nexus One также выиграли от уменьшения количества матричных операций до двух. Конечно, здесь мы идем на хитрость: ведь если нам нужно повернуть и изменить размеры Боба, мы не можем убрать два дополнительных вызова. Тем не менее, когда мы работаем с 2D-визуализацией, мы можем применить небольшой маневр, который поможет нам избавиться от всех матричных операций (мы рассмотрим такой маневр в следующей главе).

OpenGL ES — это API на языке C, его применение в языке Java обеспечивается при помощи обертки JNI. Это значит, что любой метод OpenGL ES, который мы вызываем, проходит через JNI-обертку, чтобы вызвать саму C-нативную функцию. В более ранних версиях Android при этом потреблялась масса памяти, но в более поздних версиях все стало гораздо лучше. Как вы видите сами, это влияние не так уж заметно, особенно если операции занимают больше времени, чем запуск вызова.

Концепция связывания вершин

Можем ли мы еще что-то оптимизировать? Еще раз посмотрим на наш текущий метод `present()` (с убранными `glRotatef()` и `glScalef()`):

```
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

    gl.glMatrixMode(GL10.GL_MODELVIEW);
    for(int i = 0; i < NUM_BOBS; i++) {
        gl.glLoadIdentity();
        gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
        bobModel.draw(GL10.GL_TRIANGLES, 0, 6); }
    }

    fpsCounter.logFrame();
}
```

Выглядит гораздо лучше, не правда ли? Но на самом деле метод еще не идеален. Для начала мы можем переместить вызов `gl.glMatrixMode()` в метод `resume()`, однако это не даст значительных улучшений. Вот что еще можно оптимизировать.

Мы используем класс `Vertices` для хранения и визуализации моделей Бобов. Помните метод `Vertices.draw()`? Вот он:

```
public void draw(int primitiveType, int offset, int numVertices) {
    GL10 gl = glGraphics.getGL();

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    vertices.position(0);
    gl.glVertexPointer(2, GL10.GL_FLOAT, vertexSize, vertices);

    if(hasColor) {
```

```
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
        vertices.position(2);
        gl.glColorPointer(4, GL10.GL_FLOAT, vertexSize, vertices);
    }

    if(hasTexCoords) {
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        vertices.position(hasColor?6:2);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, vertexSize, vertices);
    }

    if(indices!=null) {
        indices.position(offset);
        gl.glDrawElements(primitiveType, numVertices,
                          GL10.GL_UNSIGNED_SHORT, indices);
    } else {
        gl.glDrawArrays(primitiveType, offset, numVertices);
    }

    if(hasTexCoords)
        gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

    if(hasColor)
        gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
}
```

Теперь давайте снова рассмотрим предыдущий цикл. Ничего не заметили? Для каждого Боба используем одни и те же свойства вершин снова и снова с помощью `glEnableClientState()`. На самом деле нам нужно всего лишь установить их один раз, поскольку каждый Боб применяет одну и ту же модель, которая всегда задействует одни и те же свойства вершин. Следующая проблема связана с вызовом `glXXXPointer()` для каждого экземпляра Боба. Поскольку эти указатели также являются состояниями OpenGL, нам нужно установить их всего один раз, так как после установки они уже не меняются. Как мы можем это исправить? Немного перепишем метод `Vertices.draw()`:

```
public void bind() {
    GL10 gl = glGraphics.getGL();

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    vertices.position(0);
    gl.glVertexPointer(2, GL10.GL_FLOAT, vertexSize, vertices);

    if(hasColor) {
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
        vertices.position(2);
        gl.glColorPointer(4, GL10.GL_FLOAT, vertexSize, vertices);
    }

    if(hasTexCoords) {
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        vertices.position(hasColor?6:2);
```

```
            gl.glTexCoordPointer(2, GL10.GL_FLOAT, vertexSize, vertices);
        }
    }

    public void draw(int primitiveType, int offset, int numVertices) {
        GL10 gl = glGraphics.getGL();

        if(indices!=null) {
            indices.position(offset);
            gl.glDrawElements(primitiveType, numVertices,
                              GL10.GL_UNSIGNED_SHORT, indices);
        } else {
            gl.glDrawArrays(primitiveType, offset, numVertices);
        }
    }

    public void unbind() {
        GL10 gl = glGraphics.getGL();

        if(hasTexCoords)
            gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

        if(hasColor)
            gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
    }
```

Видите, что мы здесь сделали? Мы можем обращаться с нашими вершинами и другими указателями так же, как и с текстурой. Привязываем указатели вершин с помощью одного вызова `Vertices.bind()`. С этого момента каждый вызов `Vertices.draw()` будет работать с этими привязанными вершинами так же, как вызов рисования всегда использует текущую привязанную текстуру. Когда мы закончили визуализацию с экземплярами класса `Vertices`, вызываем `Vertices.unbind()`, чтобы отменить все свойства вершин, которые не нужны другим экземплярам класса `Vertices`. Целесообразно не вносить лишней информации в состояние OpenGL ES. Вот как теперь выглядит наш метод `present()` (я также переместил вызов `glMatrixMode(GL10.GL_MODELVIEW)` в `resume()`):

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

    bobModel.bind();
    for(int i = 0; i < NUM_BOBS; i++) {
        gl.glLoadIdentity();
        gl.glTranslatef(bobs[i].x, bobs[i].y, 0);
        bobModel.draw(GL10.GL_TRIANGLES, 0, 6);
    }
    bobModel.unbind();

    fpsCounter.logFrame();
}
```

Фактически мы вызываем методы `glXXXPointer()` и `glEnableClientState()` только один раз за кадр. Таким образом мы сэкономили около 100 · 6 вызовов OpenGL ES. Это должно значительно повлиять на работу устройств, правильно? Действительно.

```
Hero:
12-10 05:16:59.710: DEBUG/FPSCounter(865): fps: 51
12-10 05:17:00.720: DEBUG/FPSCounter(865): fps: 46
12-10 05:17:01.720: DEBUG/FPSCounter(865): fps: 47
12-10 05:17:02.610: DEBUG/dalvikvm(865): GC freed 21815 objects / 524272 bytes in
131ms
12-10 05:17:02.740: DEBUG/FPSCounter(865): fps: 44
12-10 05:17:03.750: DEBUG/FPSCounter(865): fps: 50

Droid:
12-10 05:22:27.519: DEBUG/FPSCounter(2040): fps: 57
12-10 05:22:28.519: DEBUG/FPSCounter(2040): fps: 57
12-10 05:22:29.526: DEBUG/FPSCounter(2040): fps: 57
12-10 05:22:30.526: DEBUG/FPSCounter(2040): fps: 55

Nexus One:
12-10 05:18:31.915: DEBUG/FPSCounter(2509): fps: 56
12-10 05:18:32.935: DEBUG/FPSCounter(2509): fps: 56
12-10 05:18:33.935: DEBUG/FPSCounter(2509): fps: 55
12-10 05:18:34.965: DEBUG/FPSCounter(2509): fps: 54
```

Теперь все устройства примерно сравнялись. Droid работает лучше всех, за ним следует Nexus One. Наш маленький Hero также работает весьма неплохо. Мы можем гордиться собой. Наш оптимизированный тест Боба теперь достаточно хорош.

Новый привязываемый класс `Vertices`, правда, не лишен некоторых ограничений.

- Мы можем устанавливать данные вершин и индексов только тогда, когда экземпляры класса `Vertices` не привязаны, поскольку загрузка информации о вершине осуществляется в `Vertices.bind()`.
- Мы не можем одновременно привязать два экземпляра класса `Vertices`. Это значит, что в каждом моменте мы можем визуализировать только один экземпляр класса `Vertices`. Как правило, это не большая проблема, особенно если учесть значительные улучшения в работе, так что давайте смиримся с этим.

В заключение

Есть еще один вариант оптимизации, который можно применить и при программировании 2D-графики в планиметрии — в частности, прямоугольников. Об этом мы поговорим в следующей главе. Ключевой аспект такой оптимизации —

объединение в группы (batching), а следовательно, и сокращение количества вызовов `glDrawElements()`/`glDrawArrays()`. Аналогичный процесс применяется и в 3D-графике, он называется *клонированием* (instancing), однако в OpenGL ES 1.x он не работает.

Перед тем как закончить эту главу, я хочу упомянуть еще две вещи. Прежде всего, когда вы запускаете `BobText` или `OptimizedBobTest` (который содержит супероптимизированный код, который мы только что написали), обратите внимание, что Бобы перемещаются по экрану с характерным подергиванием. Это объясняется тем, что местоположение фигурок передается методу `glTranslatef()` в формате чисел с плавающей точкой. Проблема с попиксельной визуализацией связана с тем, что среда OpenGL ES действительно очень чувствительна к месторасположению вершин, чьи координаты имеют дробные значения. Мы не можем достаточно эффективно обойти эту проблему, однако в игре такой эффект будет несущественным или по крайней мере не слишком выраженным. В этом мы сможем убедиться далее, при разработке нашей следующей игры. Чтобы еще сильнее сгладить данный эффект, можно использовать более разнообразный фон.

Еще один важный момент. Он связан с тем, как мы интерпретируем данные о кадровой частоте. Как показывают приведенные выше результаты, количество кадров в секунду немного колеблется. Это можно объяснить фоновыми процессами, которые выполняются параллельно с приложением. Мы никогда не сможем потратить на игру всех системных ресурсов, с этим нужно смириться. Когда вы оптимизируете программу, не имитируйте такой нереальной среды, в которой отсутствуют любые фоновые процессы. Запустите приложение на телефоне в нормальном состоянии, в котором вы используете его в течение дня. Это отразит ситуацию, с которой столкнется пользователь.

Хочу также дать еще один совет. Начинайте оптимизировать код визуализации только после того, как закончите работу над ним и только если у вас возникнут проблемы с работой устройства. Преждевременная оптимизация часто приводит к необходимости переписать весь код визуализации, поскольку в некоторых случаях код после оптимизации становится невозможно поддерживать.

Подводя итог

OpenGL ES — это действительно очень сложная штука. Мы умудрились сократить все до размеров, которые приемлемы для нашего игрового программирования. Обсудили, что такое OpenGL ES (попросту говоря, машина для визуализации треугольников) и как она работает. Затем изучили, как использовать функционал OpenGL ES, определяя вершины, как создавать текстуры и как использовать такие состояния, как смешивание, для получения нескольких неплохих эффектов.

Мы также немного поговорили о проекциях и том, как они связаны с матрицами. Хотя мы не рассматривали, что происходит внутри матрицы, мы изучили, как использовать ее для того, чтобы повернуть, масштабировать и перевести модели из пространства модели в пространство мира. Когда мы будем позже применять OpenGL ES для 3D-программирования, вы заметите, что вы уже выучили 90 % того, что вам необходимо знать. Мы изменим проекцию и добавим к нашим вершинам *z*-координату. Перед этим мы напишем милую 2D-игру с использованием OpenGL ES. В следующей главе вы узнаете о некоторых технологиях 2D-программирования, которые могут вам для этого понадобиться.

8 Трюки при разработке 2D-игр

В главе 7 мы узнали, что OpenGL ES предлагает множество функций для графического 2D-программирования. Среди них — вращение, масштабирование и автоматическое растяжение конуса отображения до размеров области просмотра. OpenGL ES также позволяет работать быстрее, чем при использовании `Canvas`.

Теперь давайте изучим более сложные темы, связанные с 2D-программированием игр. Некоторые из этих концепций мы использовали интуитивно, когда писали «Мистера Нома», как, например, хронологически-зависимые обновления состояний и работа с атласами текстур. Многое из того, что нам предстоит изучить, интуитивно понятно, и вполне вероятно, что рано или поздно вы сами пришли бы к подобному решению.

Мы рассмотрим удобные и наиболее важные концепции 2D-программирования игр. Некоторые из них будут касаться графики, другие будут посвящены тому, как мы представляем и эмулируем наш игровой мир. В основе всей этой работы лежат линейная алгебра и тригонометрия. Но не бойтесь, чтобы написать игру уровня «Супер Марио», математики требуется совсем немного. Так что приступим к изучению некоторых концепций двухмерной линейной алгебры и тригонометрии.

Перед стартом

Как и в предыдущих «теоретических» главах, мы собираемся создать несколько примеров, чтобы лучше понять, что происходит. В этой главы мы вновь используем материалы, подготовленные в предыдущей: в основном это будут классы `GLGame`, `GLGraphics`, `Texture` и `Vertices`, а также остальные классы фреймворка.

Наш демонстрационный проект, который мы рассмотрим для начала, называется `GameDev2DStarter` и содержит список тестов для запуска. Мы можем повторно использовать код `GLBasicsStarter` и просто заменить названия классов в тестах. Кроме того, понадобится добавить все тесты в файл манифеста в виде элементов `<activity>`.

Каждый из этих тестов — это опять-таки экземпляр интерфейса `Game`, а сама логика тестов реализована в виде `Screen`, содержащегося в `Game` реализации теста, как и в предыдущей главе. Я продемонстрирую лишь те части `Screen`, которые важны в конкретном примере, чтобы вы лучше понимали процесс. Что касается

названий, мы снова используем `XXXTest` и `XXXScreen` для реализаций `GLGame` и `Screen` каждого теста.

Теперь поговорим о векторах.

Сначала был вектор

В предыдущей главе мы говорили, что не надо путать векторы и позиции. Это не совсем так, поскольку мы можем (и будем) представлять местоположение в пространстве с помощью вектора. Вектор можно интерпретировать по-разному.

- *Позиция.* Мы уже использовали такую интерпретацию в предыдущих главах для кодирования координат объектов относительно начала координат.
- *Скорость и ускорение.* Об этих физических величинах мы поговорим в следующем разделе. Хотя в обыденном понимании скорость и ускорение — это скалярные величины, в 2D или 3D они выражаются в виде векторов. Они кодируют не только скорость объекта (например, машины, едущей со скоростью 100 км/ч), но также направление, в котором движется объект. Обратите внимание, что такая интерпретация не означает, что вектор находится в начале системы координат. Это логично, поскольку скорость и направление движения не зависят от местоположения объекта. Представьте себе машину, едущую на северо-запад по прямой трассе со скоростью 100 км/ч. Если скорость и направление не будут меняться, вектор скорости также останется неизменным.
- *Направления и расстояния.* Направление схоже со скоростью, но не имеет каких-либо физических характеристик. Мы можем использовать подобную векторную интерпретацию направлений, чтобы описывать ситуации вида «данная сущность указывает на юго-восток». Расстояния просто сообщают, как далеко и в каком направлении одна позиция находится относительно другой.

На рис. 8.1 показаны эти интерпретации в действии.

Конечно, рисунок охватывает не все случаи. У вектора может быть еще множество интерпретаций. Тем не менее для интересующей нас разработки игр этих четырех базовых интерпретаций вполне достаточно.

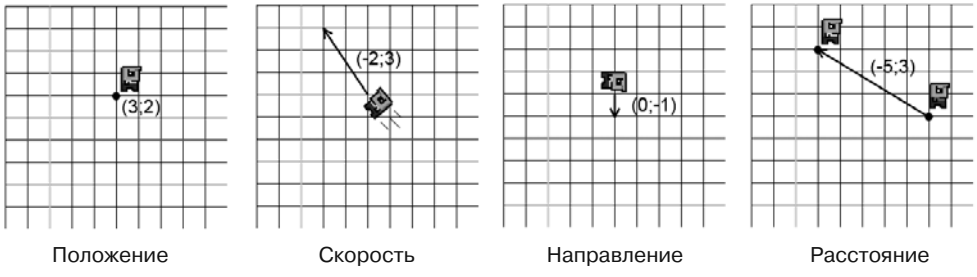

Рис. 8.1. Базовые интерпретации вектора

Еще одна деталь, опущенная на рис. 8.1, — информация о том, в каких единицах измеряются компоненты вектора. Это всегда следует проверять. Например, скорость

Боба может измеряться в метрах в секунду, и он будет передвигаться со скоростью 2 м влево, 3 м вверх за одну секунду. Это касается и позиций, и расстояний, которые также могут быть выражены в метрах. Направление Боба — это особый случай, поскольку это безразмерная величина. Безразмерность удобна, если нужно определить общее направление объекта, держа физические величины направления отдельно. Такая же операция применима и для скорости Боба: можно сохранить направление его скорости в качестве вектора направления, а скорость сохранить как скалярное значение. Для этого вектор направления должен иметь длину 1, однако мы обсудим это позже.

Работа с векторами

Потенциал векторов основывается на том факте, что мы можем легко управлять ими и комбинировать их. Перед тем как приступить к этому, давайте определимся с представлением векторов:

```
v = (x,y)
```

Ничего удивительного, мы это уже сто раз делали. Каждый вектор имеет компоненты x и y в двухмерном пространстве (да, в этой главе мы по-прежнему останемся в 2D). Мы также можем складывать векторы:

```
c = a + b = (a.x, a.y) + (b.x, b.y) = (a.x + b.x, a.y + b.y)
```

Чтобы получить конечный вектор, нужно сложить компоненты. Попробуйте это сделать с векторами на рис. 8.1. Допустим, местоположение Боба p = (3; 2), а его скорость v = (–2; 3). Мы перемещаемся в новое место p' = (3 + –2; 2 + 3) = (1,5). Не обращайте внимания на апостроф после p, он просто показывает, что у нас есть новый вектор p. Конечно, это маленькая операция целесообразна лишь тогда, когда местоположение и скорость измеряются в одних и тех же единицах. В этом случае допустим, что местоположение измеряется в метрах (м), а скорость — в метрах в секунду (м/с).

Естественно, мы можем также вычитать векторы:

```
c = a – b = (a.x, a.y) – (b.x, b.y) = (a.x – b.x, a.y – b.y)
```

Опять-таки мы просто комбинируем компоненты двух векторов. Обратите внимание на то, что порядок, в котором мы вычитаем один вектор из другого, весьма важен. Возьмите, к примеру, рис. 8.1. Зеленый Боб (на крайнем правом рисунке сверху) находится на `pg` = (1; 4), а красный Боб (на крайнем правом рисунке снизу) — на `pr` = (6; 1), где `pg` и `pr` означают местоположения зеленого и красного соответственно[1]. Когда мы вычитаем вектор расстояния красного Боба от зеленого Боба, выполняются следующие вычисления:

```
d = pg – pr = (1, 4) – (6, 1) = (-5, 3)
```

[1] pg содержит в названии букву g — от green (с англ. — «зеленый»), а pr включает в название r — от red (с англ. — «красный»). — *Примеч. ред.*

Вот что странно. Этот вектор фактически направлен от красного Боба к зеленому. Чтобы получить вектор направления от зеленого Боба к красному, нужно поменять порядок вычитания:

```
d = pr – pg = (6, 1) – (1, 4) = (5, -3)
```

Если мы хотим найти вектор расстояния между точками *a* и *b*, мы используем следующую общую формулу:

```
d = b – a
```

Другими словами, мы всегда отнимаем стартовую позицию от конечной. Сначала это кажется немного путаным, но если вы задумаетесь об этом, то поймете, что это правильно. Попробуйте сделать это на миллиметровой бумаге.

Мы также можем умножать вектор на скаляр:

```
a' = a * scalar = (a.x * scalar, a.y * scalar)
```

Мы умножаем каждый компонент вектора на скаляр. Это позволяет нам изменять длину вектора. Возьмите в качестве примера вектор направления из рис. 8.1. Он определен как $d = (0; -1)$. Если мы умножим его на скаляр, равный 2, мы вдвое увеличим его длину: $d \cdot s = (0; -1 \cdot 2) = (0; -2)$. Естественно, мы можем таким путем и уменьшить его длину, используя значения скаляра меньше единицы, например *d*, умноженное на $s = 0{,}5$, приводит к созданию нового вектора $d' = (0; -0{,}5)$.

Говоря о длине, мы также можем подсчитать длину вектора (в тех единицах, в которых она измеряется):

```
|a| = sqrt(a.x*a.x + a.y*a.y)
```

Запись `|a|` просто означает, что мы говорим о длине вектора. Если вы не прогуляли уроки линейной алгебры в школе, вам должна быть известна формула длины вектора. Это теорема Пифагора, примененная к 2D-вектору. Компоненты *x* и *y* вектора образуют две стороны треугольника, третья сторона — это длина вектора (рис. 8.2).

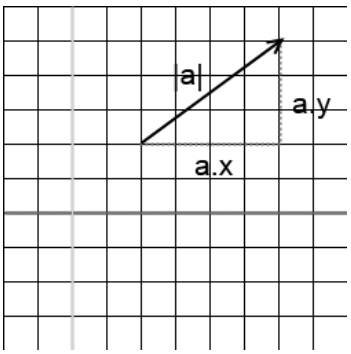

Рис. 8.2. Пифагору бы понравились векторы

Длина вектора всегда больше нуля, как и квадратный корень. Если применить это к вектору расстояния между красным и зеленым Боба, то выясним, как далеко они находятся друг от друга (если оба местоположения даны в метрах).

```
|pr – pg| = sqrt(5*5 + -3*-3) = sqrt(25 + 9) = sqrt(34) ~= 5.83m
```

Обратите внимание, что если бы отнимали `|pg – pr|`, получили бы такой же результат, поскольку длина не зависит от направления вектора. Однако отсюда мы можем вынести еще один урок: когда мы умножаем вектор на скаляр, длина вектора изменяется соответственно. Вектор $d = (0; -1)$ с начальной длиной в 1 единицу, умноженный на 2,5, даст нам новый вектор с длиной 2,5 единицы.

Мы уже обсуждали, что, как правило, направление векторов является безразмерной величиной. Мы можем сделать эту величину и размерной, умножив вектор на скаляр — например, чтобы получить вектор скорости v, мы можем умножить вектор направления $d = (0; 1)$ на константу скорости $s = 100$ м/с: $v = (0 \cdot 100; 1 \cdot 100) = (0; 100)$. Поэтому удобно, чтобы длина векторов была равна 1. Векторы, чья длина равна 1, называются единичными (unit vectors). Мы можем превратить любой вектор в единичный, разделив каждый его компонент на его длину:

```
d' = (d.x/|d|, d.y/|d|)
```

Запомните, что `|d|` означает только длину вектора d. Допустим, нам нужен вектор направления, который указывает ровно на северо-восток: $d = (1; 1)$. Нам может казаться, что этот вектор уже единичный, поскольку оба его компонента равны 1, правильно? Нет, неправильно:

```
|d| = sqrt(1*1 + 1*1) = sqrt(2) ~= 1.44
```

Мы можем легко это исправить, превратив вектор в единичный.

```
d' = (d.x/|d|, d.y/|d|) = (1/|d|, 1/|d|) ~= (1/1.44, 1/1.44) = (0.69, 0.69)
```

Такая операция также называется *нормализацией* вектора, то есть мы приводим его к значению 1. Такая небольшая уловка позволяет нам создать единичный вектор направления из вектора расстояния. Конечно, нельзя допускать появления нулевых векторов, ведь на ноль делить нельзя!

Немного тригонометрии

Давайте на минутку обратимся к тригонометрии. В тригонометрии есть две основополагающие функции: косинус и синус. Каждая из них принимает один аргумент: угол. Мы привыкли измерять углы в градусах (например, 45° или 360°). Однако в большинстве математических библиотек тригонометрические функции измеряются в радианах. Можно легко перевести градусы в радианы с помощью следующего уравнения:

```
degreesToRadians(angleInDegrees) = angleInDegrees / 180 * pi
radiansToDegrees(angle) = angleInRadians / pi * 180
```

Здесь используется π, суперконстанта, которая примерно равна 3,14159265. π радиан равно 180°.

Что же подсчитывают функции косинуса и синуса, измеряя углы? Они подсчитывают *x*- и *y*-компоненты единичного вектора относительно начала (рис. 8.3).

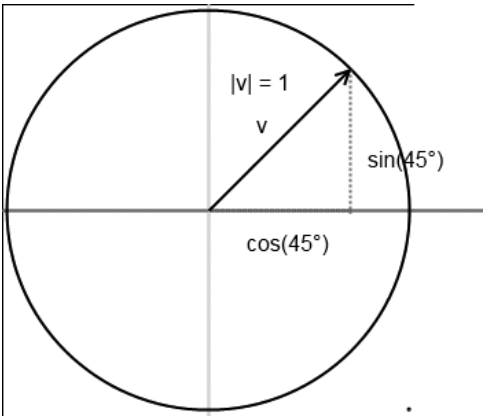

Рис. 8.3. Косинус и синус создают единичный вектор с его конечной точкой, лежащей на единичном круге

Имея угол, мы можем легко создать единичный вектор направления:

```
v = (cos(angle), sin(angle))
```

Мы можем поступить иначе: подсчитать угол вектора относительно оси *x*:

```
angle = atan2(v.y, v.x)
```

Функция `atan2` — это искусственная конструкция. Она использует функцию арктангенса (что является обратной функцией тангенса, еще одной фундаментальной функцией в тригонометрии), чтобы создать угол от −180° до 180° (или от `-pi` до `pi`, если угол измеряется в радианах). Это все достаточно сложно и не относится к теме нашей дискуссии. *y*- и *x*-компоненты нашего вектора являются аргументами функции. Обратите внимание, что вектор необязательно должен быть единичным, чтобы функция `atan2` работала. Кроме того, заметьте, что *y*-компонент, как правило, дается первым, а *x*-компонент — вторым, однако это зависит от математической библиотеки, которую мы используем. Это один из наиболее частых источников ошибок.

Рассмотрим несколько примеров. Имея вектор v = (cos(97°); sin(97°)), мы получим результат atan2(sin(97°); cos(97°)), равный 97°. Отлично, это было достаточно просто. Используя вектор v = (1; −1), мы получаем atan2(−1; 1) = −45°. Так что если *y*-компонент вектора отрицательный, мы получим отрицательный угол, равный от 0° до −180°. Мы можем исправить это, прибавив 360° (или `2pi`), если результат `atan2` отрицательный. В предыдущем примере мы тогда получили бы 315°.

Конечная операция, которую мы хотим применить к нашим векторам, — поворот их на какой-либо угол. Процедуры вывода уравнений, которые мы сейчас рассмотрим, опять-таки достаточно сложные. К счастью, мы можем просто использовать их, не углубляясь в ортогональный базис векторов (подсказка: о том, что это такое, можно спросить в поисковиках). Вот наш магический псевдокод:

```
v.x' = cos(angle) * v.x – sin(angle) * v.y
v.y' = sin(angle) * v.x + cos(angle) * v.y
```

Ух ты, это было гораздо проще, чем мы думали. Так мы повернем любой вектор против часовой стрелки вне зависимости от его интерпретации.

Вместе со сложением, вычитанием и умножением векторов на скаляр мы фактически можем сами реализовать все матричные операции OpenGL. Это одна из частей комплексного решения для последующей оптимизации работы нашего `BobTest` в последней главе. Мы поговорим об этом в следующих разделах. Теперь сконцентрируемся на том, что мы обсудили, и переведем все это в код.

Реализация класса Vector

Мы хотим создать удобный в использовании векторный класс для 2D-векторов. Назовем его `Vector2`. Он должен состоять из двух элементов, для записи x- и y-компонентов вектора. Дополнительно он должен иметь несколько удобных методов, которые позволят нам выполнять следующие операции:

- складывать и вычитать векторы;
- умножать векторные компоненты на скаляр;
- измерять длину векторов;
- нормализировать векторы;
- вычислять угол между вектором и осью x;
- поворачивать вектор.

В Java не применяется перегрузка операторов, так что нам придется найти механизм, который поможет сделать работу с классом `Vector2` не такой громоздкой. В идеале у нас должно получиться что-то в таком роде:

```
Vector2 v = new Vector2();
v.add(10,5).mul(10).rotate(54);
```

Мы можем легко этого достичь, позволив каждому из методов `Vector2` возвратить ссылку на сам вектор. Конечно, мы также хотим перегрузить такие методы, как `Vector2.add()`, чтобы мы могли использовать или два числа с плавающей точкой, или экземпляр класса другого `Vector2`. В листинге 8.1 показан класс `Vector2`.

Листинг 8.1. `Vector2.java: реализация функционала 2D-вектора`

```
package com.badlogic.androidgames.framework.math;

import android.util.FloatMath;

public class Vector2 {
```

```
    public static float TO_RADIANS = (1 / 180.0f) * (float) Math.PI;
    public static float TO_DEGREES = (1 / (float) Math.PI) * 180;
    public float x, y;

    public Vector2() {
    }

    public Vector2(float x, float y) {
        this.x = x;
        this.y = y;
    }

    public Vector2(Vector2 other) {
        this.x = other.x;
        this.y = other.y;
    }
```

Мы перемещаем этот класс в пакет `com.badlogic.androidgames.framework.math`, где будем также хранить все математические классы.

Начинаем с описания двух статических констант: `TO_RADIANS` и `TO_DEGREES`. Чтобы перевести угол, данный в радианах, нужно просто умножить его на `TO_DEGREES`; чтобы перевести в радианы угол, указанный в градусах, мы умножаем его на `TO_RADIANS`. Вы можете перепроверить это, посмотрев на два предыдущих описанных равенства, которые предназначены для перевода из градусов в радианы и обратно. Эта небольшая уловка позволяет обойтись без деления, тем самым немного ускорив работу.

Затем опишем два элемента x и y, которые хранят компоненты вектора и еще пару конструкторов, — ничего сложного:

```
public Vector2 cpy() {
    return new Vector2(x, y);
}
```

У нас также есть метод `cpy()`, который создаст дубликат экземпляра класса текущего вектора и вернет его. Это может быть достаточно удобно, если мы хотим управлять копией вектора, сохраняя значение начального вектора.

```
public Vector2 set(float x, float y) {
    this.x = x;
    this.y = y;
    return this;
}

public Vector2 set(Vector2 other) {
    this.x = other.x;
    this.y = other.y;
    return this;
}
```

Методы set() позволяют нам установить x- и y-компоненты вектора, беря за основу либо два аргумента, выраженных числами с плавающей точкой, либо другой вектор. Методы возвращают ссылку на этот вектор, так что мы можем создавать цепи операций, как мы это обсуждали выше.

```
public Vector2 add(float x, float y) {
    this.x += x;
    this.y += y;
    return this;
}

public Vector2 add(Vector2 other) {
    this.x += other.x;
    this.y += other.y;
    return this;
}

public Vector2 sub(float x, float y) {
    this.x -= x;
    this.y -= y;
    return this;
}

public Vector2 sub(Vector2 other) {
    this.x -= other.x;
    this.y -= other.y;
    return this;
}
```

Методы add() и sub() бывают двух разновидностей: в одном случае они работают с двумя аргументами в виде чисел с плавающими точками, в другом — используют еще один экземпляр класса Vector2. Все четыре метода возвращают ссылку на этот вектор, чтобы мы могли создать цепь операций.

```
public Vector2 mul(float scalar) {
    this.x *= scalar;
    this.y *= scalar;
    return this;
}
```

Метод mul() просто умножает компоненты вектора x и y на данную скалярную величину и снова возвращается к вектору для цепи.

```
public float len() {
    return FloatMath.sqrt(x * x + y * y);
}
```

Метод len() подсчитывает длину вектора ровно так же, как мы описывали это выше. Обратите внимание, что мы используем класс FloatMath вместо обычного

Math, предлагаемого Java SE. Это специальный класс Android API, который работает с float, он немного быстрее, чем его аналог Math.

```
public Vector2 nor() {
    float len = len();
    if (len != 0) {
        this.x /= len;
        this.y /= len;
    }
    return this;
}
```

Метод nor() нормализирует вектор до единичной длины. Мы используем метод len() внутрисистемно, чтобы сначала подсчитать длину. Если она равна нулю, мы выходим из операции раньше, чтобы избежать деления на ноль. В другом случае мы делим каждый компонент вектора на его длину, чтобы получить единичный вектор. Для объединения действий в цепочку снова возвращаем ссылку на вектор.

```
public float angle() {
    float angle = (float) Math.atan2(y, x) * TO_DEGREES;
    if (angle < 0)
        angle += 360;
    return angle;
}
```

Метод angle() вычисляет угол между вектором и осью x, используя метод atan2(), что мы и обсудили выше. Нам необходимо применить метод Math.atan2(), поскольку у класса FastMath нет этого метода. Возвращенный угол дается в радианах, так что мы переводим его в градусы, умножая его на TO_DEGREES. Если угол меньше нуля, прибавляем к нему 360°, чтобы он имел значение от 0° до 360°.

```
public Vector2 rotate(float angle) {
    float rad = angle * TO_RADIANS;
    float cos = FloatMath.cos(rad);
    float sin = FloatMath.sin(rad);

    float newX = this.x * cos – this.y * sin;
    float newY = this.x * sin + this.y * cos;

    this.x = newX;
    this.y = newY;

    return this;
}
```

Метод rotate() просто поворачивает вектор вокруг начала координат на данный угол. Поскольку методы FloatMath.cos() и FloatMath.sin() требуют указания угла в радианах, переводим сначала градусы в радианы. Затем используем ранее описанные равенства, чтобы подсчитать новые компоненты вектора x и y, и наконец возвращаем вектор для добавления в цепь.

```
    public float dist(Vector2 other) {
        float distX = this.x – other.x;
        float distY = this.y – other.y;
        return FloatMath.sqrt(distX * distX + distY * distY);
    }

    public float dist(float x, float y) {
        float distX = this.x – x;
        float distY = this.y – y;
        return FloatMath.sqrt(distX * distX + distY * distY);
    }
}
```

В итоге у нас есть два метода, которые подсчитывают расстояния между этим и другим вектором.

Вот наш замечательный класс `Vector2`, который мы будем применять для представления месторасположения, скорости, расстояния и направления в последующем коде. Чтобы лучше понять этот класс, используем его на простом примере.

Простой пример использования

Вот мой вариант простого теста.

- Мы создадим своего рода пушку, представленную треугольником, который имеет фиксированное местоположение в мире. Центр треугольника будет находиться в координатах (2,4; 0,5).
- Каждый раз, когда мы касаемся экрана, мы хотим повернуть треугольник в сторону точки касания.
- Конус отображения показывает зону нашего мира между (0; 0) и (4,8; 3,2). Мы не работаем с пиксельными координатами, но вместо этого описываем свою систему координат, где одна единица равна одному метру. Мы также будем работать в альбомной ориентации.

Существует несколько нюансов, заслуживающих особого внимания. Мы уже знаем, как описать треугольник в пространстве модели: для этого можно использовать экземпляр класса `Vertices`. По умолчанию наша пушка должна указывать направо под углом 0°. На рис. 8.4 показана пушка-треугольник в пространстве модели.

Когда мы визуализируем этот треугольник, мы просто используем `glTranslatef()`, чтобы передвинуть его на его место в мире, а именно на (2,4; 0,5).

Мы также хотим повернуть пушку таким образом, чтобы ее конец указывал в сторону точки, где произошло касание. Для этого нам необходимо выяснить, где в нашем мире произошло последнее событие касания. Методы `GLGame.getInput().getTouchX()` и `getTouchY()` вернут точку касания в **экранных координатах с началом** в верхнем левом углу. Кроме того, напоминаю, что экземпляр класса `Input` не преобразует координаты событий в фиксированную систему координат, как это было в «Мистере Номе». Вместо этого мы получаем координаты (479; 319),

когда касаемся нижнего правого угла экрана (ландшафтная ориентация) на Her, и (799; 479) на Nexus One. Нам нужно перевести эти координаты касания в координаты нашего мира. Мы уже делали это в обработчике касаний в «Мистере Номе» и основанном на Canvas фреймворке игры. Единственная разница состоит в том, что наша система координат немного меньше, а ось y направлена вверх. Вот псевдокод, показывающий, как мы можем достичь перевода в общем случае, что представляет собой практически то же самое, что и обработчики касаний из главы 5:

```
worldX = (touchX / Graphics.getWidth()) * viewFrustmWidth
worldY = (1 – touchY / Graphics.getHeight()) * viewFrustumHeight
```

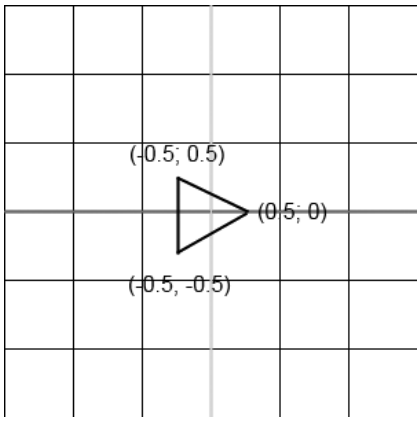

Рис. 8.4. Пушка-треугольник в пространстве модели

Мы нормализуем координаты касания до диапазона (0; 1), деля их на разрешение экрана. В случае с y-координатой вычитаем нормализованную y-координату события касания от 1, чтобы изменить по модулю направление по оси y. Осталось масштабировать координаты x- и y-координат относительно высоты и ширины конуса отображения — в нашем случае это 4,8 и 3,2. Из координат worldX и worldY можно создать величину Vector2, где будет сохранена позиция точки касания в системе координат игрового мира.

Наконец, нужно вычислить угол, на который следует повернуть пушку. На рис. 8.5 показаны пушка и точка касания в системе координат игрового мира.

Необходимо создать вектор расстояния между центром пушки (2,4; 0,5) и точкой касания (помните, что нам надо вычесть координаты центра пушки из координат точки касания, а не наоборот). Когда мы получим этот вектор расстояния, можем вычислить угол с помощью метода Vector2.angle(). Этот угол затем можно использован для поворота нашей модели с помощью glRotatef().

Запрограммируем это. В листинге 8.2 показан соответствующий фрагмент нашего класса CannonScreen, входящего в класс CannonTest.

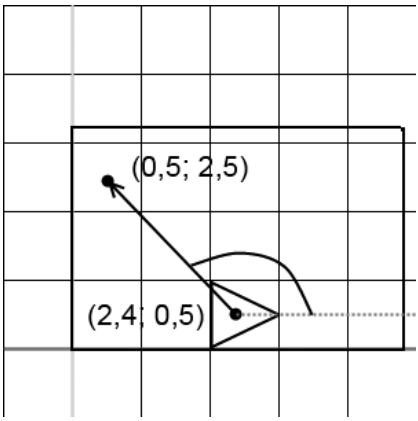

Рис. 8.5. Пушка в состоянии, задаваемом по умолчанию, указывает направо (угол = 0°), точка касания и угол, на который нужно повернуть пушку. Прямоугольник — это область мира, которая будет показана в конусе отображения на экране: диапазон от (0; 0) до (4,8; 3,2)

Листинг 8.2. Отрывок из CannonTest.java; прикосновение к экрану повернет пушку

```
class CannonScreen extends Screen {
    float FRUSTUM_WIDTH = 4.8f;
    float FRUSTUM_HEIGHT = 3.2f;
    GLGraphics glGraphics;
    Vertices vertices;
    Vector2 cannonPos = new Vector2(2.4f, 0.5f);
    float cannonAngle = 0;
    Vector2 touchPos = new Vector2();
```

Начинаем с двух констант, которые описывают ширину и высоту конуса отображения, как это обсуждалось выше. Затем используем экземпляр класса `GLGraphics`, а также экземпляр класса `Vertices`. Мы также сохраняем положения пушки в `Vector2`, а ее угол выражает в числах с плавающей точкой. Наконец, применяем еще один `Vector2`, который нужен для того, чтобы вычислить угол между началом вектора и точкой касания по оси x.

Зачем сохранять экземпляры класса `Vector2` как элементы класса? Мы могли бы создавать их экземпляры каждый раз, когда они нам нужны, однако так получается слишком много мусора. Как правило, нужно стараться создавать экземпляры `Vector2` только по одному разу и как можно чаще использовать их снова.

```
public CannonScreen(Game game) {
    super(game);
    glGraphics = ((GLGame) game).getGLGraphics();
    vertices = new Vertices(glGraphics, 3, 0, false, false);
    vertices.setVertices(new float[] { -0.5f, -0.5f,
                                        0.5f,  0.0f,
                                       -0.5f,  0.5f }, 0, 6);
}
```

В конструкторе выбираем экземпляр класса GLGraphics и создаем треугольник в соответствии с рис. 8.4.

```
@Override
public void update(float deltaTime) {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    game.getInput().getKeyEvents();

    int len = touchEvents.size();
    for (int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);

        touchPos.x = (event.x / (float) glGraphics.getWidth())
                * FRUSTUM_WIDTH;
        touchPos.y = (1 - event.y / (float) glGraphics.getHeight())
                * FRUSTUM_HEIGHT;
        cannonAngle = touchPos.sub(cannonPos).angle();
    }
}
```

Далее следует метод update(). Мы просто перебираем в цикле все TouchEvent и подсчитываем угол пушки. Это делается в несколько шагов. Сначала преобразуем координаты событий касания в точки координатной системы мира, как обсуждали это выше. Сохраняем координаты событий касания в координатной системе мира в элементе touchPoint. Затем вычитаем положение пушки из вектора touchPoint, в результате получаем вектор, показанный на рис. 8.5. Затем вычисляем угол между этим вектором и осью *x*. Вот и все!

```
@Override
public void present(float deltaTime) {

    GL10 gl = glGraphics.getGL();
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    gl.glOrthof(0, FRUSTUM_WIDTH, 0, FRUSTUM_HEIGHT, 1, -1);
    gl.glMatrixMode(GL10.GL_MODELVIEW);
    gl.glLoadIdentity();

    gl.glTranslatef(cannonPos.x, cannonPos.y, 0);
    gl.glRotatef(cannonAngle, 0, 0, 1);
    vertices.bind();
    vertices.draw(GL10.GL_TRIANGLES, 0, 3);

    vertices.unbind();
}
```

Метод present() занят все той же скучной работой, что и раньше. Мы задаем область просмотра, очищаем экран, устанавливаем матрицу ортогональной проекции, используя ширину и высоту конуса отображения, и сообщаем OpenGL ES, что

все последующие операции матрицы будут применяться к модельно-видовой матрице. Мы также загружаем единичную матрицу в модельно-видовую матрицу, чтобы «очистить» ее. Затем умножаем (единичную) модельно-видовую матрицу на матрицу трансляции, которая перенесет вершины нашего треугольника из пространства модели в пространство мира. Также вызываем `glRotatef()` с углом, который мы подсчитали в методе `update()`, чтобы повернуть наш треугольник в пространстве модели, перед тем как переместить его в пространство мира. Не забывайте, что преобразования применяются в обратном порядке — последнее указанное изменение будет выполнено первым. Наконец, привязываем вершины треугольника, визуализируем его и отвязываем.

```
    @Override
    public void pause() {

    }

    @Override
    public void resume() {

    }

    @Override
    public void dispose() {

    }
}
```

Теперь у нас есть треугольник, который будет следовать за каждым касанием. На рис. 8.6 показан результат после касания верхнего левого угла экрана.

Рис. 8.6. Треугольник-пушка, реагирующий на событие касания в верхнем левом углу

Обратите внимание: неважно, визуализируем мы на месте пушки треугольник или прямоугольную текстуру, ассоциированную с изображением пушки, — для OpenGL ES это не играет роли. Мы также снова используем матричные операции в методе `present()`.

Дело в том, что таким образом гораздо проще следить за состояниями OpenGL ES. Достаточно часто мы будем использовать несколько конусов отображения в одном вызове `present()` (например, расчет мира в метрах для визуализации нашего мира и еще один расчет мира в пикселах для визуализации элементов пользовательского интерфейса). Это не так сильно влияет на быстродействие программы, как факторы, описанные в предыдущей главе, так что вполне можно пользоваться этими методами постоянно. Просто запомните, что мы можем при необходимости оптимизировать этот функционал.

С данного момента мы уже не сможем жить без векторов. С их помощью мы будем указывать практически все элементы игрового мира. Кроме того, с их помощью мы реализуем простейшую физику. Куда годится пушка, если она не стреляет, правда?

Немного 2D-физики

В этом разделе мы используем очень простую и достаточно ограниченную физику. Разработчики игр идут на все, чтобы избавиться от сложных вычислений. Поведение объектов в игре не должно быть на 100 % физически точным, оно просто должно быть достаточно реалистичным, чтобы выглядеть правдоподобно. Иногда мы даже не хотим, чтобы поведение было полностью физически достоверным (например, один набор объектов должен падать вниз, а другой такой же — вверх).

Даже в классической игре «Супер Марио» используется несколько простых принципов ньютоновской физики. Эти принципы действительно достаточно просты и легки в применении. Мы говорим только об абсолютном минимуме, необходимом для реализации очень простой физической модели для наших игровых объектов.

Ньютон и Эйлер — друзья навек

Нас в основном интересует физика перемещения, которая касается изменения положения, скорости, ускорения объекта во времени. Мы работаем с «точечной массой», то есть приравниваем все объекты к крайне малым точкам, имеющим ассоциированную с ними массу. Нас, например, не интересует крутящий момент — вращательная скорость объекта, движущегося вокруг центра массы, — поскольку это достаточно сложная проблема, о которой была написана не одна толстая книга.

- Рассмотрим свойства нашего объекта.
- Позиция объекта — это просто вектор в пространстве, в нашем случае в 2D-пространстве. Мы представляем его в качестве вектора. Обычно местоположение дается в метрах.

- Скорость объекта — это изменение местоположения в секунду. Скорость дается как двухмерный вектор скорости, который представляет собой сочетание вектора направления (длиной в единицу) в сторону, в которую направляется объект, и скорости в метрах в секунду, с которой движется объект. Обратите внимание, что скорость управляет только длиной вектора скорости. Если мы нормализуем вектор скорости, мы получим вектор направления длиной в единицу.
- Ускорение объектов — это изменение их скорости в секунду. Мы можем выражать это или в скалярах, которые просто отображают скорость (длину вектора скорости) или как 2D-вектор, чтобы у нас было разное ускорение по осям x и y. В данном случае мы выберем второй вариант, поскольку он упрощает работу с такими вещами, как баллистика. Фактически ускорение измеряется в метрах в секунду за секунду (м/с^2). Нет, это не опечатка, мы действительно изменяем скорость на какое-то значение, измеряющееся в метрах в секунду каждую секунду.

Когда у нас есть эти свойства объекта для заданного момента времени, мы можем совместить их, чтобы эмулировать путь объекта в мире с течением времени. Может показаться, что звучит это страшновато, но мы уже поступали так с «Мистером Номом» и нашем `BobTest`. В тех случаях мы просто не использовали ускорение, мы устанавливали скорость с фиксированным вектором. Вот как можно интегрировать ускорение, скорость и местоположение объекта:

```
Vector2 position = new Vector2();
Vector2 velocity = new Vector2();
Vector2 acceleration = new Vector2(0, -10);
while(simulationRuns) {
    float deltaTime = getDeltaTime();
    velocity.add(acceleration.x * deltaTime, acceleration.y * deltaTime);
    position.add(velocity.x * deltaTime, velocity.y * deltaTime);
}
```

Этот феномен также называется *Эйлерова числовая интеграция*, и это наиболее понятный метод интеграции из всех, используемых в играх. Мы начинаем с точки (0; 0), при этом скорость равна (0; 0), а ускорение — (0; –10). Это означает, что скорость будет возрастать на 1 метр в секунду по оси y. По оси x движения не будет. Перед тем как начнется цикл интеграции, объекты будут находиться в состоянии покоя. Внутри цикла сначала обновляем скорость, основанную на ускорении, умноженном на дельту времени. Вот и все, что скрывается за таким умным словом *интеграция*.

ПРИМЕЧАНИЕ

Как правило, это даже не полдела. Эйлерова интеграция — это нестабильный метод, которого по возможности следует избегать. Обычно используется другой вариант интеграции, лишь немного более сложный, — интеграция Верле. Однако для наших целей упрощенной интеграции Эйлера вполне достаточно.

Сила и масса

Может возникнуть вопрос: а откуда берется ускорение? Вопрос хороший, и ответов на него много. Ускорение машины берется из двигателя. Двигатель применяет силу к машине, что вызывает ее ускорение. Однако это еще не все. Машина также ускоряется в сторону центра Земли из-за гравитации. Единственная вещь, которая препятствует тому, чтобы она провалилась к центру Земли, это то, что она не может пройти сквозь толщу земной коры. Земля нивелирует силу гравитации. Общая идея выглядит так:

```
Сила = масса * ускорение
```

Мы можем превратить это в следующее уравнение:

```
Ускорение = сила / масса
```

Сила выражается в единицах СИ — ньютонах (догадайтесь, кто их придумал). Если мы определяем ускорение как вектор, мы также должны определять силу как вектор. Таким образом, сила может иметь направление. Например, сила гравитации направлена вниз (0; –1). Ускорение также зависит от массы объекта. Чем больше масса объекта, тем большую силу нам нужно применить, чтобы она ускорялась так же быстро, как и объект меньшего веса. Это прямое следствие предыдущего равенства.

Однако для простых игр мы можем игнорировать массу и силу и работать непосредственно со скоростью и ускорением. В предыдущем псевдокоде мы задали ускорение на (0; –10) метров в секунду за секунду (опять повторяю, это не опечатка), что примерно равно ускорению, которое испытает объект, падая на землю вне зависимости от его массы (мы не принимаем во внимание сопротивление воздуха). Это действительно так, Галилей подтвердит!

Манипулируем теоретически

Используем наш предыдущий пример, чтобы поэкспериментировать с объектом, падающим вниз на Землю. Предположим, что мы прошли цикл 10 раз и что `getDeltaTime()` всегда будет равно 0,1 секунды. Мы получим следующие местоположения и скорости для каждого раза:

```
time=0.1, position=(0.0,-0.1), velocity=(0.0,-1.0)
time=0.2, position=(0.0,-0.3), velocity=(0.0,-2.0)
time=0.3, position=(0.0,-0.6), velocity=(0.0,-3.0)
time=0.4, position=(0.0,-1.0), velocity=(0.0,-4.0)
time=0.5, position=(0.0,-1.5), velocity=(0.0,-5.0)
time=0.6, position=(0.0,-2.1), velocity=(0.0,-6.0)
time=0.7, position=(0.0,-2.8), velocity=(0.0,-7.0)
time=0.8, position=(0.0,-3.6), velocity=(0.0,-8.0)
time=0.9, position=(0.0,-4.5), velocity=(0.0,-9.0)
time=1.0, position=(0.0,-5.5), velocity=(0.0,-10.0)
```

За 1 секунду объект падает на 5,5 м и имеет скорость (0; –10) м/с, направляясь прямо к коре Земли (конечно, до тех пор, как он ударится о землю).

Объект постоянно увеличивает скорость падения, поскольку мы не принимаем во внимание фактор сопротивления воздуха. (Как я говорил раньше, в нашей системе нам никто не запрещает немного жульничать). Мы можем легко узнать текущую максимальную скорость, посмотрев на длину вектора скорости.

Всезнающая «Википедия» сообщает нам, что человек в свободном падении может достичь максимальной скорости 125 миль в час. Если перевести это в метры в секунду ($125 \cdot 1{,}6 \cdot 1000 / 3600$), мы получим 55,5 м/с. Чтобы сделать нашу эмуляцию более реалистичной, мы можем изменить цикл следующим образом:

```
while(simulationRuns) {
    float deltaTime = getDeltaTime();
    if(velocity.len() < 55.5)
        velocity.add(acceleration.x * deltaTime, acceleration.y *
                deltaTime);
    position.add(velocity.x * deltaTime, velocity.y * deltaTime);
}
```

Поскольку скорость нашего объекта (длина вектора скорости) меньше, чем 55,5 м/с, мы увеличим скорость с помощью ускорения. Когда мы достигли максимальной скорости, мы просто больше не увеличиваем ее ускорением. Такое простое лимитирование скоростей используется во множестве игр.

Мы могли бы добавить ветер в уравнение, указав еще одно ускорение по оси x, допустим $(-1; 0)$ м/с^2. Для этого нам нужно просто добавить гравитационное ускорение и ускорение ветра перед тем, как мы прибавим все это к скорости:

```
Vector2 gravity = new Vector2(0,-10);
Vector2 wind = new Vector2(-1,0);
while(simulationRuns) {
    float deltaTime = getDeltaTime();
    acceleration.set(gravity).add(wind);
    if(velocity.len() < 55.5)
        velocity.add(acceleration.x * deltaTime, acceleration.y *
                deltaTime);
    position.add(velocity.x * deltaTime, velocity.y * deltaTime);
}
```

Мы можем также не учитывать вообще никакого ускорения и задать объектам фиксированную скорость. Именно так мы раньше поступали в `BobTest`. Мы изменяли скорость каждого Боба, если он касался края.

Манипулируем на практике

Возможности, предоставляемые даже этой простой моделью, практически бесконечны. Дополним наш небольшой класс `CannonTest`, чтобы можно было стрелять ядрами. Вот, что мы хотим сделать.

- Когда пользователь водит пальцем по экрану, пушка следует за ним. Так мы определяем угол, под которым выстрелим ядром.
- Когда мы получаем событие касания, выстреливаем ядром в сторону, куда направлена пушка. Начальная скорость ядра будет следовать из направления

пушки и изначальной скорости ядра. Скорость равна расстоянию между пушкой и точкой касания. Чем дальше мы касаемся, тем быстрее ядро будет лететь.

- Ядро будет лететь до тех пор, пока не появится новое событие касания.
- Мы вдвое увеличим размер конуса отображения от (0; 0) до (9,6; 6,4), чтобы мы могли видеть большую часть мира. Кроме того, разместим пушку в (0; 0). Обратите внимание, что все единицы в игровом мире даются в метрах.
- Мы визуализируем ядро как красный квадрат размером 0,2 × 0,2 м, что равно 20 × 20 см. Как мне кажется, достаточно похоже на настоящее ядро. Конечно, если среди вас есть канониры, вы можете выбрать размер пореалистичнее.

Изначально местоположение ядра будет (0; 0) — таким же, как у пушки. Скорость также будет равна (0; 0). Поскольку мы применяем гравитацию в каждом обновлении, ядро просто будет падать прямо и вниз.

Когда получено событие завершения касания (touch up), мы снова задаем положение ядра в точку (0; 0), а его скорость — в значение (`Math.cos(cannonAngle)`; `Math.sin(cannonAngle)`). Это гарантирует, что ядро будет лететь в сторону, в которую направлена пушка. Мы также задаем скорость, просто умножим скорость на расстояние между точкой касания и пушкой. Чем ближе точка касания к пушке, тем медленнее будет лететь ядро.

Звучит достаточно просто, реализуем это. Я скопировал код из `CannonTest` в новый файл `CannonGravityTest.java`. Переименовал классы, которые находятся в этом файле в `CannonGravityTest` и `CannonGravityScreen`.

В листинге 8.3 показан `CannonGravityScreen`.

Листинг 8.3. Фрагмент из CannonGravityTest

```
class CannonGravityScreen extends Screen {
    float FRUSTUM_WIDTH = 9.6f;
    float FRUSTUM_HEIGHT = 6.4f;
    GLGraphics glGraphics;
    Vertices cannonVertices;
    Vertices ballVertices;
    Vector2 cannonPos = new Vector2();
    float cannonAngle = 0;
    Vector2 touchPos = new Vector2();
    Vector2 ballPos = new Vector2(0,0);
    Vector2 ballVelocity = new Vector2(0,0);
    Vector2 gravity = new Vector2(0,-10);
```

Не так уж много изменилось. Мы просто вдвое увеличили размер конуса отображения и отразили это, установив `FRUSTUM_WIDTH` и `FRUSTUM_HEIGHT` на 9,6 и 6,2 соответственно. Это значит, что мы можем видеть в нашем игровом мире прямоугольник размером 9,2 × 6,2. Поскольку мы также хотим нарисовать ядро, я добавил еще один экземпляр класса `Vertices`, называющийся `ballVertices`, который будет содержать 4 вершины и 6 индексов прямоугольника ядра. Новые элементы `ballPos` и `ballVelocity` содержат положение и скорость ядра, а элемент `gravity` — это гравитационное ускорение, которое является константой (0; –10) м/с^2.

```
public CannonGravityScreen(Game game) {
    super(game);
    glGraphics = ((GLGame) game).getGLGraphics();
    cannonVertices = new Vertices(glGraphics, 3, 0, false, false);
    cannonVertices.setVertices(new float[] { -0.5f, -0.5f,
                                              0.5f,  0.0f,
                                             -0.5f, 0.5f }, 0, 6);
    ballVertices = new Vertices(glGraphics, 4, 6, false, false);
    ballVertices.setVertices(new float[] { -0.1f, -0.1f,
                                            0.1f, -0.1f,
                                            0.1f,  0.1f,
                                           -0.1f,  0.1f }, 0, 8);
    ballVertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);
}
```

В конструкторе просто создаем дополнительный экземпляр класса `Vertices` для прямоугольника пушки. Снова определяем его в пространстве модели с вершинами (–0,1; –0,1), (0,1; –0,1), (0,1; 0,1) и (–0,1; 0,1). Рисуем с применением индексов, так что в этом случае также определяем шесть вершин.

```
@Override
public void update(float deltaTime) {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    game.getInput().getKeyEvents();

    int len = touchEvents.size();
    for (int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);

        touchPos.x = (event.x / (float) glGraphics.getWidth())
                  * FRUSTUM_WIDTH;
        touchPos.y = (1 – event.y / (float) glGraphics.getHeight())
                  * FRUSTUM_HEIGHT;
        cannonAngle = touchPos.sub(cannonPos).angle();

        if(event.type == TouchEvent.TOUCH_UP) {
            float radians = cannonAngle * Vector2.TO_RADIANS;
            float ballSpeed = touchPos.len();
            ballPos.set(cannonPos);
            ballVelocity.x = FloatMath.cos(radians) * ballSpeed;
            ballVelocity.y = FloatMath.sin(radians) * ballSpeed;
        }
    }

    ballVelocity.add(gravity.x * deltaTime, gravity.y * deltaTime);
    ballPos.add(ballVelocity.x * deltaTime, ballVelocity.y * deltaTime);
}
```

Метод `update()` также изменился незначительно. Подсчеты точки касания в координатах мира и угла пушки остаются прежними. Первое дополнение заключается в конструкции `if` внутри цикла обработки событий. Еесли мы получаем событие

завершения касания, то готовим наше ядро к выстрелу. Сначала преобразуем угол пушки в радианы, для этого используем `FloatMath.cos()` и `FloatMath.sin()`. Затем подсчитываем расстояние между пушкой и точкой касания. Это будет скорость ядра. После этого устанавливаем местоположение ядра в положение пушки. Наконец, подсчитываем начальную скорость ядра. Применяем синусы и косинусы, как мы обсудили это выше, чтобы создать направление вектора из угла пушки. Умножаем этот вектор на скорость ядра, чтобы получить конечную скорость ядра. Это интересно, поскольку ядро с самого первого момента будет иметь скорость. В реальном мире ядро, конечно, ускорялось бы от 0 м/с до той скорости, которую могло бы достичь с учетом сопротивления воздуха и силы, примененной к нему со стороны пушки. Мы можем здесь схитрить, поскольку это ускорение будет происходить за очень краткий промежуток времени (пара тысячных секунды). Последняя деталь, которая выполняется в методе `update()`, — обновление скорости ядра и в зависимости от этого изменение местоположения.

```
@Override
    public void present(float deltaTime) {

        GL10 gl = glGraphics.getGL();
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glOrthof(0, FRUSTUM_WIDTH, 0, FRUSTUM_HEIGHT, 1, -1);
        gl.glMatrixMode(GL10.GL_MODELVIEW);

        gl.glLoadIdentity();
        gl.glTranslatef(cannonPos.x, cannonPos.y, 0);
        gl.glRotatef(cannonAngle, 0, 0, 1);
        gl.glColor4f(1,1,1,1);
        cannonVertices.bind();
        cannonVertices.draw(GL10.GL_TRIANGLES, 0, 3);
        cannonVertices.unbind();

        gl.glLoadIdentity();
        gl.glTranslatef(ballPos.x, ballPos.y, 0);
        gl.glColor4f(1,0,0,1);
        ballVertices.bind();
        ballVertices.draw(GL10.GL_TRIANGLES, 0, 6);
        ballVertices.unbind();
    }
```

В методе `present()` мы просто добавляем визуализацию прямоугольника ядра. Мы делаем это после визуализации треугольника пушки, что означает, что нам нужно очистить модельно-видовую матрицу до того, как мы визуализируем прямоугольник. Делаем это с помощью `glLoadIdentity()`, а затем используем `glTranslatef()`, чтобы преобразовать прямоугольник ядра из пространства модели в пространство мира, чтобы ядро попало в ту точку, в которой должно находиться.

```
    @Override
    public void pause() {

    }

    @Override
    public void resume() {

    }

    @Override
    public void dispose() {

    }
}
```

Если вы запустите пример и коснетесь экрана несколько раз, вы хорошо поймете, как летает ядро. На рис. 8.7 показан результат (который, конечно, не особо впечатляет, поскольку это просто схематичная картинка).

Рис. 8.7. Треугольная пушка, которая стреляет красными прямоугольниками. Впечатляюще!

Этой физики вполне хватит для наших целей. С данной простой моделью мы можем создать гораздо больше ядер. Например, «Супер Марио» был в основном создан так же. Если вы когда-то играли в «Супер Марио», вы заметили, что Марио требуется немного времени, чтобы достичь максимальной скорости при беге. Это реализуется с помощью очень большого ускорения и лимитирования скорости, как и в предыдущем псевдокоде. Прыжки можно реализовать таким же способом, каким мы запускаем ядра. Текущая скорость Марио будет скорректирована начальным

увеличением скорости по оси y (не забывайте, что мы можем складывать скорости так же, как и любые другие векторы). Если бы под его ногами не было земли, мы бы применили гравитационное ускорение, чтобы он снова упал на землю. Скорость по оси x не зависит от скорости по оси y. Мы также можем нажимать кнопки «Влево» и «Вправо», чтобы изменить скорость по оси x. Красота этой простой модели состоит в том, что она позволяет нам реализовать сложное поведение с помощью очень малого количества кода. Мы будем использовать тот же тип физики при написании нашей следующей игры.

Просто так стрелять ядрами не очень весело, гораздо интереснее стрелять ядрами по объектам. Для этого нам нужно уметь определять столкновение, что мы и изучим в следующем разделе.

Определение столкновений и представление объектов в 2D

Поскольку в нашем мире есть движущиеся объекты, существуют и взаимодействия между ними. Одним из видов таких взаимодействий является столкновение. Два объекта считаются столкнувшимся, когда они каким-либо образом пересекаются. Мы уже встречались со столкновениями, когда проверяли, что поглощает мистер Ном — себя или чернильное пятно. Определение столкновений обычно сопровождается ответом на столкновение: после того как мы определили, что два объекта столкнулись, мы должны отреагировать на столкновение корректировкой положения и/или движения наших объектов должным образом. Например, когда Марио прыгает на гриб Гумба, Гумба отправляется в свой грибной рай, а Марио выполняет еще один маленький прыжок. Более точный пример — столкновение и реакция на столкновение двух и более бильярдных шаров. Мы не будем углубляться в разбор этого вида реакции на столкновение, поскольку это не нужно для наших целей. Наша реакция на столкновение обычно будет состоять в изменении состояния объекта (например, объект может взорваться, умереть, забрать монетку и т. д.). Тип реакции зависит от игры, поэтому мы не будем говорить о ней в этом разделе. Итак, как же мы определяем, что два объекта столкнулись? В первую очередь надо подумать, когда проверять, есть ли столкновения. Если наши объекты соответствуют каким-либо простым физическим моделям, о чем говорилось в предыдущем разделе, мы можем производить проверку на столкновения после того, как переместили все наши объекты на текущий кадр и шаг по времени вперед.

Ограничивающие фигуры

Когда у нас есть конечные координаты наших объектов, мы можем проводить проверку на столкновения, которая сводится к проверке на пересечение. Но что именно пересекается? Каждый наш объект имеет некоторую математически определен-

ную фигуру, которая его ограничивает. В этом случае правильным будет употребить термин «ограничивающая фигура». На рис. 8.8 показано несколько разновидностей ограничивающих фигур.

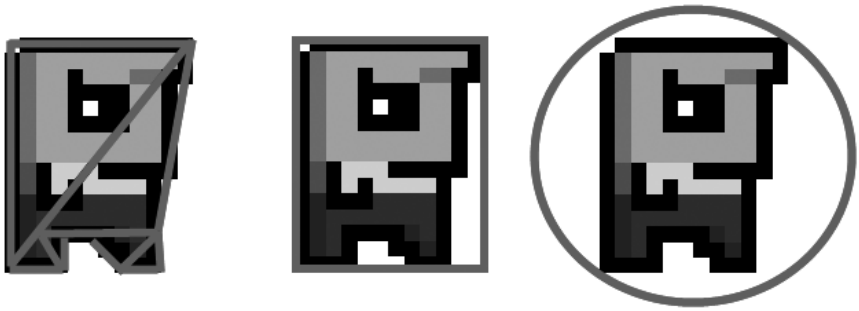

Рис. 8.8. Различные ограничивающие фигуры вокруг Боба

Это три типа ограничивающих фигур. Каждая, соответственно, имеет следующие свойства.

- *Треугольная сеть*. Больше всего прилегает к объекту, приближая его силуэт к нескольким треугольникам. Требует больше всего места для хранения, сложна в построении и требует больших затрат при тестировании, однако дает наиболее точные результаты. Необязательно использовать те же треугольники при рендеринге, но стоит хранить их для определения столкновений. Сеть может быть сохранена как список вершин, где каждые три последовательные вершины буду принадлежать одному треугольнику. Для экономии памяти можно применять индексированные списки вершин.
- *Ограничивающий прямоугольник с привязкой к осям координат*. Ограничивает объект посредством прямоугольника, который привязан к осям координат, то есть его верхняя и нижняя стороны всегда совпадают с осью x, а правая и левая стороны — с осью y. Такая фигура обеспечивает легкость проверок, но точность будет ниже, чем у треугольной сети. Ограничивающая рамка обычно хранится в виде координаты ее нижнего левого угла и ее ширины и длины (при работе с 2D фигуры также могут называться ограничивающими прямоугольниками).
- *Ограничивающий круг*. Ограничивает объект наименьшей окружностью, которая может вместить объект. Она очень быстро тестируется, но результаты наименее точные из всех ограничивающих фигур. Круг обычно хранится в виде координаты центра и диаметра.

Каждый объект в игре имеет ограничивающую фигуру, которая заключает его в себе и определяет его, наряду с его координатами, масштабом и ориентацией.

Конечно, мы должны установить координаты, масштаб и положение ограничивающей фигуры в соответствии с координатами, масштабом и положением объекта, когда перемещаем объект, скажем, на этапе интегрирования физики.

Корректировка при изменении положения выполняется просто: мы всего лишь перемещаем ограничивающую фигуру в соответствии с этими изменениями. Если используется треугольная сеть, мы перемещаем каждую вершину, если ограничивающий прямоугольник — нижний левый угол, если ограничивающий круг — то перемещается его центр.

Масштабирование ограничивающей фигуры немного сложнее. Сначала нужно задать точку, вокруг которой мы будем масштабировать. Обычно это позиция объекта, которая определяется по центру объекта. С учетом этого масштабирование тоже будет несложным. Для треугольной сети масштабируем координаты каждой вершины, для ограничивающего прямоугольника — ширину, длину и нижний левый угол, а для ограничивающего круга — радиус (центр круга равен центру объекта).

Вращение ограничивающей фигуры также зависит от определения точки вращения. При упомянутом выше условии (вращение происходит вокруг центра объекта), вращение не составит труда. В случае с треугольной сетью мы просто вращаем все вершины вокруг центра объекта. С ограничивающим кругом вообще ничего не придется делать, так как его радиус не изменится, как бы мы ни вращали объект. С ограничивающим прямоугольником придется повозиться дольше. Нам надо будет построить все 4 вершины, вращать их, а потом найти привязанный к осям координат ограничивающий прямоугольник, который будет включать все 4 точки. На рис. 8.9 показаны все три ограничивающие фигуры после вращения.

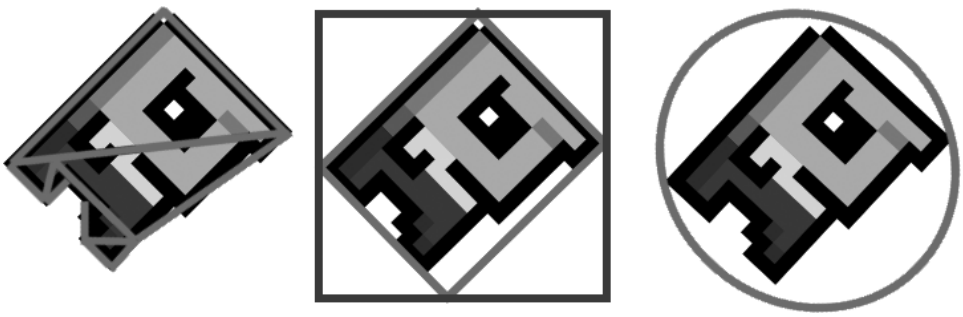

Рис. 8.9. Ограничивающие фигуры после вращения при центре объекта как точке вращения

В то время, как вращение треугольной сети или ограничивающего круга реализуется просто, вращение ограничивающей рамки, привязанной к осям координат, не всегда приводит к желаемым результатам. Стоит заметить, что ограничивающая рамка прилегает к изначальному объекту плотнее, чем к объекту после вращения. В результате возникает вопрос, как нам приспособить ограничивающие фигуры к работе с Бобом.

Создание ограничивающих фигур

В этом примере я просто построил ограничивающие фигуры от руки, взяв за основу изображение Боба. Но изображение Боба сделано в пикселах, а расстояния в нашем мире могут измеряться метрами. Для решения этой проблемы потребуется нормализация и применение пространства модели. Представьте себе два треугольника, которые мы можем использовать для Боба в пространстве модели, когда будем визуализировать его при помощи OpenGL. Прямоугольник находится по центру в начале координат пространства модели и имеет такое же соотношение ширины и высоты, как и текстурное изображение Боба (например, 32 × 32 пиксела на текстурной карте и 2 × 2 м в пространстве модели). Теперь мы можем применить растровое изображение Боба и выяснить, где в пространстве модели будут находиться точки ограничивающих фигур. На рис. 8.10 показано, как мы создаем ограничивающие фигуры вокруг Боба в пространстве модели.

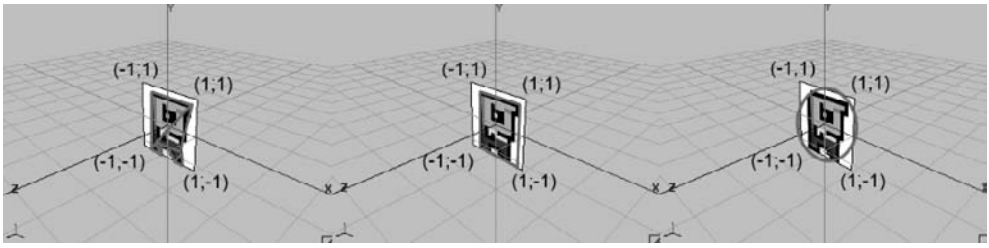

Рис. 8.10. Ограничивающие фигуры вокруг Боба в пространстве модели

Этот процесс может показаться несколько трудоемким, хотя на самом деле предпринимаемые шаги не так и сложны. В первую очередь мы должны помнить, как работает обработка текстур. Мы задаем текстурные координаты для каждой вершины прямоугольника Боба (который состоит из двух треугольников) в текстурном пространстве. Верхний левый угол растрового изображения в текстурном пространстве находится в (0; 0), а нижний левый угол — в (1; 1) независимо от ширины и высоты изображения в пикселах. Чтобы перейти из пиксельного пространства в текстурное, мы можем использовать такое несложное преобразование:

```
u = x / imageWidth
v = y / imageHeight,
```

где u и v — текстурные координаты пиксела, находящегося под координатами x и y в пространстве изображения. `ImageWidth` и `imageHeight` устанавливаются по размерам изображения в пикселах (в случае Боба — 32 × 32). На рис. 8.11 изображено, как центр изображения Боба переносится в текстурное пространство.

Текстура применяется к прямоугольнику, который мы задаем в пространстве модели. На рис. 8.10 был показан пример, где верхний левый угол находился в точке (–1; 1), а нижний правый — в (1; –1). Для измерений в нашем мире используем метры, поэтому ширина и длина прямоугольника равны 2 м каждая. Кроме того,

мы знаем, что верхний правый угол имеет текстурные координаты (0; 0), а нижний правый угол — (1; 1), и таким образом мы наносим полную текстуру Боба. Однако в одном из следующих разделов вы увидите, что это не всегда работает подобным образом.

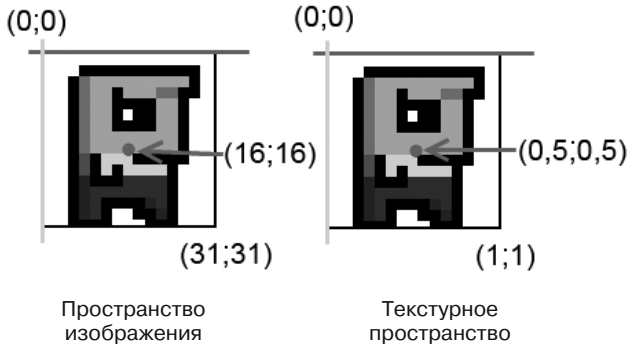

Рис. 8.11. Перенесение пиксела из пространства изображения в текстурное пространство

Рассмотрим универсальный способ ассоциирования текстурного пространства с пространством модели. Мы можем упростить себе задачу, ограничивая обработку только привязанными к осям координат прямоугольниками в текстурном пространстве и пространстве модели. Это значит, что мы допускаем, что привязанная к осям координат прямоугольная область в текстурном пространстве переносится в привязанный к осям координат прямоугольник в пространстве модели. Для преобразования нам нужно знать ширину и длину прямоугольника в пространстве модели и ширину и длину прямоугольника в текстурном пространстве. В нашем примере с Бобом у нас есть прямоугольник 2×2 в пространстве модели и прямоугольник 1×1 в текстурном пространстве (поскольку мы переносим полную текстуру в прямоугольник). Нам также необходимо знать координаты верхнего левого угла каждого прямоугольника в соответствующем пространстве. Для прямоугольника в пространстве модели это (–1; 1), а для прямоугольника в текстурном пространстве — (0; 0) (опять же, поскольку мы переносим всю текстуру, а не ее часть). С этими данными и координатами u и v пиксела, который мы хотим перенести в пространство модели, мы можем произвести преобразование при помощи следующих двух уравнений:

```
mx = (u – minU) / (tWidth) * mWidth + minX
my = (1 – ((v – minV) / (tHeight)) * mHeight – minY
```

Переменные `u` и `v` — это координаты, которые мы вычислили в последнем преобразовании из пиксельного в текстурное пространство. Переменные `minU` и `minV` — это координаты верхнего левого угла области, которую мы переносим из текстурного пространства. Переменные `tWidth` и `tHeight` — ширина и длина нашей области текстурного пространства. Переменные `mWidth` и `mHeight` — ширина и длина прямоугольника нашего пространства модели. Переменные `minx` и `minY`, как вы догадались, — координаты верхнего левого угла прямоугольника в пространственной

модели. И наконец, mx и my являются преобразованными координатами в пространстве модели.

Эти уравнения принимают координаты *u* и *v*, ассоциируют их с диапазоном от 0 до 1, затем масштабируют и помещают в пространство модели. На рис. 8.12 изображен тексел в текстурном пространстве и то, как он перенесен в прямоугольник в пространстве модели. По бокам вы видите tWidth, tHeight, mWidth и mHeight соответственно. Верхний левый угол каждого прямоугольника соответствует координатам (minU; minV) в текстурном пространстве и (minX; minY) в пространстве модели.

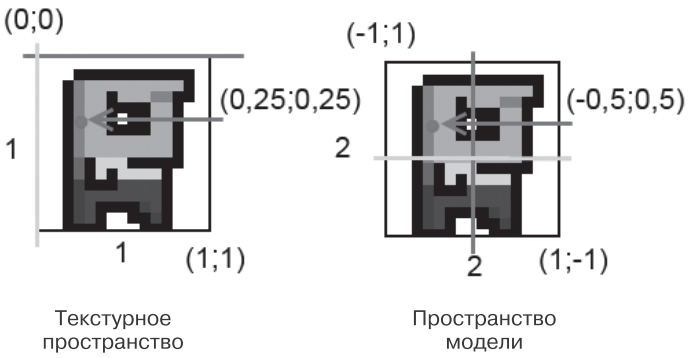

Рис. 8.12. Перенос из текстурного пространства в пространство модели

Заменив первые два уравнения, мы можем прямо перейти от пиксельного пространства к пространству модели:

```
mx = ((x/imageWidth) – minU) / (tWidth) * mWidth + minX
my = (1 – (((y/imageHeight) – minV) / (tHeight)) * mHeight – minY
```

Мы можем использовать эти два уравнения для вычисления ограничивающих фигур наших объектов, построенных на основе изображения, которое мы переместили в их прямоугольники с помощью обработки текстуры. В случае треугольной сети это может быть несколько сложным; ограничивающий прямоугольник и ограничивающий круг намного проще. Обычно мы не идем этим сложным путем, а стараемся создать текстуры таким образом, чтобы хотя бы ограничивающие прямоугольники имели такое же соотношение сторон, как и прямоугольник, который мы визуализируем для объекта при помощи OpenGL ES. Таким образом мы можем строить ограничивающий прямоугольник прямо по размерам изображения объекта. Это же будет верным и для ограничивающего круга. Я просто хотел показать, как вы можете создать какую-либо ограничивающую фигуру вокруг изображения, которое переносится в пространство модели.

Вам следует знать, как строить подходящую ограничивающую фигуру для своих 2D-объектов. Но помните, что мы определяем размеры этих ограничивающих фигур вручную, когда создаем наши графические ресурсы и задаем компоненты и размеры объектов в игровом мире. Затем мы используем эти размеры в нашем коде для столкновения объектов друг с другом.

Атрибуты игровых объектов

Теперь наш Боб стал толще. Кроме сети, которую мы используем для визуализации (отображение прямоугольника в текстуре изображения Боба), у нас также есть вторичная структура данных, которая сохраняет его границы в какой-то форме. Важно понимать, что пока мы моделируем границы по перенесенной версии Боба в пространстве, сами эти границы независимы от того участка текстуры, куда мы переносим прямоугольник Боба. Конечно, создавая ограничивающую фигуру, мы стараемся придать ей максимальное соответствие контуру Боба в текстуре. Впрочем, неважно, имеет ли текстурное изображение размеры 32×32 или 128×128 пикселов. У объекта в нашем игровом мире, таким образом, есть три группы атрибутов.

- Позиция, ориентация, масштаб, скорость и ускорение. Используя их, мы можем применить нашу физическую модель из прошлого раздела. Конечно, некоторые объекты могут быть статичными и иметь только позицию, ориентацию и масштаб. Иногда можно даже опустить и ориентацию с масштабом. Позиция объекта обычно совпадает с его началом координат в пространстве модели, как на рис. 8.10. Это делает некоторые вычисления проще.
- Ограничивающая фигура (обычно строится в пространстве модели вокруг центра объекта), которая совпадает с его позицией и согласована с объектом в ориентации и масштабе, как показано на рис. 8.10. Она придает объекту очертания и определяет его размер в мире. Мы можем сделать фигуру настолько сложной, насколько захотим, например создать ее из нескольких других ограничивающих фигур.
- Графическое представление. Как показано на рис. 8.12, мы все так же используем два треугольника, чтобы создать один прямоугольник для Боба, и переносим его изображение на прямоугольник. Прямоугольник определяется в пространстве модели, однако он необязательно равен ограничивающей фигуре, как показано на рис. 8.10. Графический прямоугольник Боба, который мы посылаем в OpenGL ES, немного больше, чем его ограничивающий прямоугольник.

Такое разделение атрибутов позволяет нам снова применить паттерн «Модель — вид — контроллер» (MVC).

- Со стороны модели у нас есть физические атрибуты Боба, включающие его позицию, масштаб, вращение, скорость, ускорение и ограничивающую фигуру. Позиция, масштаб и ориентация Боба определяют, где его ограничивающая фигура располагается в мире.
- Вид принимает графическое представление Боба (например, два текстурных треугольника, определенных в пространстве модели) и отображает их на соответствующих позициях в игровом мире, в зависимости от позиции, вращения и масштаба Боба. Здесь мы можем использовать матричные операции OpenGL ES, как делали ранее.

- Контроллер отвечает за обновление физических атрибутов Боба в соответствии с вводимыми пользователем данными (например, нажатие клавиши «Влево» подвинет его влево), а также в соответствии с такими физическими силами, как ускорение свободного падения (как те, которые мы применяли к пушечному ядру в предыдущем разделе).

Конечно, есть некоторое соответствие между ограничивающей фигурой Боба и его графическим представлением в текстуре, поскольку мы основываем ограничивающую фигуру на графическом представлении. Поэтому вариант MVC будет не совсем точным, но для наших целей подойдет.

Широкая и узкая фазы определения столкновений

Мы все еще не знаем, как проверять наличие столкновений между нашими объектами и их ограничивающими фигурами. Две фазы определения столкновения таковы.

- *Широкая фаза*. На этом этапе мы пытаемся определить, какие объекты в принципе могут столкнуться. Представьте, что есть 100 объектов, каждый из которых может столкнуться с остальными. Если мы решим упрощенно проверить каждую возможную пару объектов, нам придется провести $100 \cdot 100 / 2$ проверок на пересечение. Этот метод тестирования на пересечение имеет асимптотическую сложность $O(n2)$; это означает, что потребуется $n2$ шагов для завершения теста (на самом деле мы закончим, выполнив лишь половину операций, но асимптотическая сложность не учитывает константы). При использовании более умного алгоритма для широкой фазы мы пытаемся определить, какие пары объектов имеют вероятные шансы на столкновение. Другие пары (например, два объекта, которые расположены слишком далеко друг от друга, чтобы столкнуться), не будут проверяться. Таким образом мы сможем уменьшить вычислительную нагрузку на этом этапе, так как в узкой фазе тестирование требует больше затрат.
- *Узкая фаза*. После того как мы узнали, какие пары объектов могут столкнуться, мы проверяем, действительно ли они сталкиваются, проводя проверку на пересечение для их ограничивающих форм. Сосредоточимся на узкой фазе, а широкую оставим на потом.

Узкая фаза

После того как мы разобрались с дальним этапом, нам нужно проверить, пересекаются ли ограничивающие фигуры объектов, которые потенциально могут столкнуться. Я уже упоминал ранее, что у нас есть несколько видов ограничивающих фигур. Треугольные сети самые трудоемкие в плане вычисления и создания. Обычно в большинстве 2D-игр мы можем обойтись ограничивающими прямоугольниками и кругами, поэтому сконцентрируемся на них.

Столкновение кругов. Ограничивающие круги — наименее сложный инструмент для проверки столкновений. Определим простой класс `Circle` (листинг 8.4).

Листинг 8.4. `Circle.java`, простой класс `Circle`

```
package com.badlogic.androidgames.framework.math;

public class Circle {
    public final Vector2 center = new Vector2();
    public float radius;

    public Circle(float x, float y, float radius) {
        this.center.set(x,y);
        this.radius = radius;
    }
}
```

Мы сохраняем центр как `Vector2` и радиус как обычное число с плавающей точкой. Как проверить, перекрывают ли два круга друг друга? Смотрим на рис. 8.13.

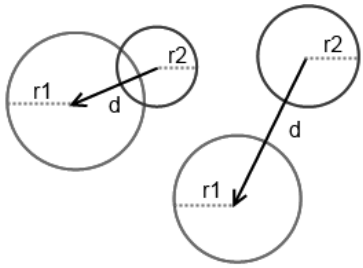

Рис. 8.13. Два круга, перекрывающие друг друга, и два круга, не перекрывающие друг друга

Это действительно просто и удобно в плане вычислений. Нам нужно выяснить расстояние между двумя центрами. Если расстояние больше, чем сумма двух радиусов, круги не пересекаются. В коде это будет выглядеть так:

```
public boolean overlapCircles(Circle c1, Circle c2) {
    float distance = c1.center.dist(c2.center);
    return distance <= c1.radius + c2.radius;
}
```

Сначала измеряем расстояние между центрами, а потом узнаем, превышает это расстояние сумму радиусов или равно ему.

Нам придется извлечь квадратный корень в методе `Vector2.dist()`. Это не очень хорошо, поскольку извлечение квадратного корня — затратная операция. Можем ли мы сделать это быстрее? Можем — достаточно только переформулировать наши условия:

```
sqrt(dist.x . dist.x + dist.y . dist.y) <= radius1 + radius2
```

Мы можем избавиться от квадратного корня, возведя в степень обе части неравенства, из чего получится:

```
dist.x . dist.x + dist.y . dist.y <= (radius1 + radius2) . (radius1 + radius2)
```

Мы заменяем квадратный корень дополнительными сложением и умножением в правой части.

Так уже лучше. Создадим функцию `Vector2.distSquared()`, которая вернет нам возведенное в квадрат расстояние между двумя векторами:

```
public float distSquared(Vector2 other) {
    float distX = this.x – other.x;
    float distY = this.y – other.y;
    return distX*distX + distY*distY;
}
```

Тогда метод `overlapCircles()` будет таким:

```
public boolean overlapCircles(Circle c1, Circle c2) {
    float distance = c1.center.distSquared(c2.center);
    float radiusSum = c1.radius + c2.radius;
    return distance <= radiusSum * radiusSum;
}
```

Столкновение прямоугольников. Займемся прямоугольниками. Для начала нам понадобится класс, который будет представлять прямоугольник. Ранее мы уже упоминали, что собираемся определять прямоугольник его нижним левым углом, шириной и длиной. Сделаем это, как в листинге 8.5.

Листинг 8.5. `Rectangle.java, класс Rectangle`

```
package com.badlogic.androidgames.framework.math;

public class Rectangle {
    public final Vector2 lowerLeft;
    public float width, height;

    public Rectangle(float x, float y, float width, float height) {
        this.lowerLeft = new Vector2(x,y);
        this.width = width;
        this.height = height;
    }
}
```

Мы сохраняем координаты нижнего левого угла как `Vector2`, а ширину и длину как два числа с плавающей точкой. Как проверить, перекрываются ли два прямоугольника? Представление об этом можно получить на рис. 8.14.

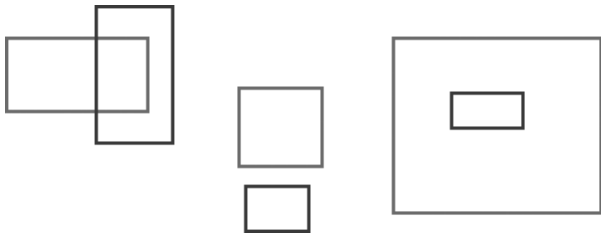

Рис. 8.14. Перекрывающиеся и неперекрывающиеся прямоугольники

Первые два случая частичного перекрывания и отсутствия перекрывания достаточно простые. Последний уже интереснее. Один прямоугольник может быть полностью заключен в другом прямоугольнике. Это может произойти и с кругами. Однако наша проверка на пересечение вернет нам правильный результат, если один круг будет содержаться в другом.

Проверка на пересечение для прямоугольников на первый взгляд кажется сложной. Однако немного логики — и мы можем получить довольно простой тест. Вот самый простой метод проверки на пересечение между двумя прямоугольниками:

```
public boolean overlapRectangles(Rectangle r1, Rectangle r2) {
    if(r1.lowerLeft.x < r2.lowerLeft.x + r2.width &&
        r1.lowerLeft.x + r1.width > r2.lowerLeft.x &&
        r1.lowerLeft.y < r2.lowerLeft.y + r2.height &&
        r1.lowerLeft.y + r1.height > r2.lowerLeft.y)
            return true;
    else
            return false;
}
```

С первого взгляда он выглядит немного запутанным, поэтому рассмотрим каждое условие. Согласно первому условию, левая сторона первого прямоугольника должна быть слева от правой стороны второго прямоугольника. Второе условие заключается в том, что правая сторона первого прямоугольника должна быть справа от левой стороны второго прямоугольника. Два оставшихся условия указывают то же самое про верхние и нижние стороны прямоугольников. Если все эти условия выполняются, прямоугольники пересекаются (см. рис. 8.14). Это также касается случая содержания одного прямоугольника в другом.

Столкновение круга и прямоугольника. Можем ли мы проверить наличие столкновения между кругом и прямоугольником? Да, можем, однако это будет сложнее. Рассмотрим рис. 8.15.

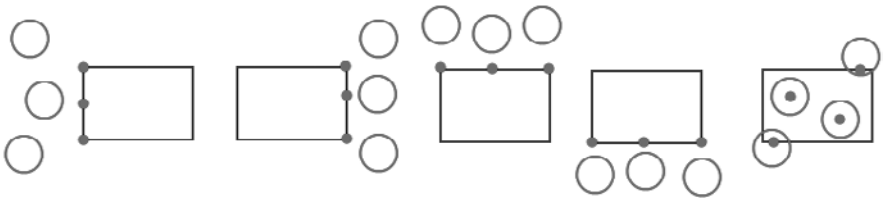

Рис. 8.15. Проверка на пересечение круга и прямоугольника путем нахождения самой близкой к кругу точки на/в прямоугольнике

Общая концепция проверки на пересечение круга и прямоугольника выглядит примерно так.

- Сначала надо найти ближайшую к центру круга координату x на периметре прямоугольника или в самом прямоугольнике. Эта координата может быть точкой на левой или правой стороне прямоугольника, если только круг не находится внутри прямоугольника, потому что в этом случае ближайшей x-координатой будет x-координата центра круга.

- Определить ближайшую к центру круга y-координату на периметре прямоугольника или в самом прямоугольнике. Данная координата может быть точкой верхней или нижней стороны прямоугольника, если только круг не содержится в прямоугольнике, потому что в этом случае ближайшей y-координатой будет y-координата центра круга.
- Если точка, определяемая ближайшими x- и y-координатами, находится в пределах круга, круг и прямоугольник пересекаются. Хотя этот метод не изображен на рис. 8.15, он также работает для кругов, полностью содержащих прямоугольник. Отобразим метод в коде:

```
public boolean overlapCircleRectangle(Circle c, Rectangle r) {
    float closestX = c.center.x;
    float closestY = c.center.y;
    if(c.center.x < r.lowerLeft.x) {
        closestX = r.lowerLeft.x;
    }
    else if(c.center.x > r.lowerLeft.x + r.width) {
        closestX = r.lowerLeft.x + r.width;
    }

    if(c.center.y < r.lowerLeft.y) {
        closestY = r.lowerLeft.y;
    }
    else if(c.center.y > r.lowerLeft.y + r.height) {
        closestY = r.lowerLeft.y + r.height;
    }

    return c.center.distSquared(closestX, closestY) < c.radius * c.radius;
}
```

Описание выглядело страшнее, чем исполнение. Мы определяем ближайшую точку прямоугольника относительно круга, а потом просто проверяем, находится ли точка внутри круга. Если это так, то круг и прямоугольник пересекаются. Обратите внимание, что я добавил перегруженный метод `distSquared()` к `Vector2`, который берет две переменные `float` вместо еще одного `Vector2`. Я сделал то же самое для функции `dist()`.

Все вместе

Проверка, лежит ли точка внутри круга или прямоугольника, тоже может быть полезной. Закодируем еще два метода и отправим в класс под названием `OverlapTester` вместе с тремя другими методами, которые мы уже определили (листинг 8.6).

Листинг 8.6. `OverlapTester.java`; проверка на пересечение кругов, прямоугольников и точек

```
package com.badlogic.androidgames.framework.math;

public class OverlapTester {
    public static boolean overlapCircles(Circle c1, Circle c2) {
        float distance = c1.center.distSquared(c2.center);
        float radiusSum = c1.radius + c2.radius;
```

```
        return distance <= radiusSum * radiusSum;
    }

    public static boolean overlapRectangles(Rectangle r1, Rectangle r2) {
        if(r1.lowerLeft.x < r2.lowerLeft.x + r2.width &&
            r1.lowerLeft.x + r1.width > r2.lowerLeft.x &&
            r1.lowerLeft.y < r2.lowerLeft.y + r2.height &&
            r1.lowerLeft.y + r1.height > r2.lowerLeft.y)
                return true;
        else
                return false;
    }

    public static boolean overlapCircleRectangle(Circle c, Rectangle r) {
        float closestX = c.center.x;
        float closestY = c.center.y;
        if(c.center.x < r.lowerLeft.x) {
            closestX = r.lowerLeft.x;
        }
        else if(c.center.x > r.lowerLeft.x + r.width) {
            closestX = r.lowerLeft.x + r.width;
        }

        if(c.center.y < r.lowerLeft.y) {
            closestY = r.lowerLeft.y;
        }
        else if(c.center.y > r.lowerLeft.y + r.height) {
            closestY = r.lowerLeft.y + r.height;
        }

        return c.center.distSquared(closestX, closestY) < c.radius *
                                                          c.radius;
    }

    public static boolean pointInCircle(Circle c, Vector2 p) {
        return c.center.distSquared(p) < c.radius * c.radius;
    }

    public static boolean pointInCircle(Circle c, float x, float y) {
        return c.center.distSquared(x, y) < c.radius * c.radius;
    }

    public static boolean pointInRectangle(Rectangle r, Vector2 p) {
        return r.lowerLeft.x <= p.x && r.lowerLeft.x + r.width >= p.x &&
               r.lowerLeft.y <= p.y && r.lowerLeft.y + r.height >= p.y;
    }

    public static boolean pointInRectangle(Rectangle r, float x, float y) {
        return r.lowerLeft.x <= x && r.lowerLeft.x + r.width >= x &&
               r.lowerLeft.y <= y && r.lowerLeft.y + r.height >= y;
    }
}
```

Прекрасно, теперь у нас есть полностью функциональная математическая 2D-библиотека, которую мы можем использовать для всех наших маленьких физических моделей и определения столкновений. Теперь более подробно обсудим широкую фазу.

Широкая фаза

Как же нам получить от широкой фазы максимум пользы? Посмотрите на рис. 8.16, на котором показана типичная сцена из «Супер Марио».

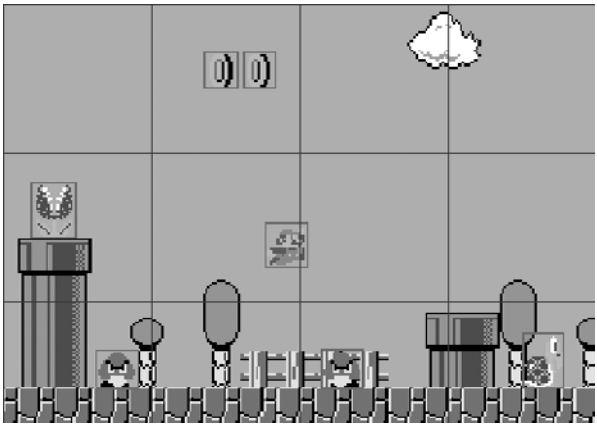

Рис. 8.16. Супер Марио и его враги; рамки вокруг объектов — ограничивающие прямоугольники; большие прямоугольники создают сетку, наложенную на картинку

Догадываетесь, что мы можем сделать, чтобы избежать некоторых проверок? Сетка на рис. 8.16 представляет собой ячейки, на которые мы делим наш игровой мир. Каждая ячейка имеет один и тот же размер; ячейки покрывают весь мир. Марио в данный момент в двух из этих ячеек. Другие объекты, с которыми Марио может столкнуться, находятся в других ячейках. Таким образом, нам не нужно проводить никаких проверок столкновения, поскольку Марио не может находиться в тех же ячейках, что и другие объекты в кадре. Необходимо сделать следующее:

- Обновить все объекты мира на основе применяемой физики и управления.
- Обновить позицию каждой ограничивающей фигуры каждого объекта согласно с позицией объекта. Конечно, мы также можем включить сюда и обновление ориентации и масштаба.
- Определить, в какой ячейке или ячейках находится каждый объект на основании его ограничивающей фигуры, и добавить его в список объектов, содержащихся в этих ячейках.
- Проверить на столкновения, но только те пары объектов, которые могут столкнуться (например, Гумбы не могут столкнуться с другими Гумбами) и которые находятся в одной ячейке. На этом этапе широкой фазы применяется пространственная сетка (spatial hash grid), реализация этого этапа не составляет труда. Первое, что нам необходимо определить, — размер каждой ячейки. Он во многом зависит от масштаба и компонентов, которые мы используем в игровом мире.

Усложненный пример

Продолжим рассматривать широкую фазу с пространственной сеткой на нашем последнем примере с пушечным ядром. Мы полностью переделаем этот пример, включив все, о чем шла речь в этом разделе. Кроме пушки и ядра у нас должны быть мишени для стрельбы. Не будем усложнять себе жизнь и используем в качестве мишеней квадраты размером 0,5 × 0,5 м. Эти квадраты статичны и не двигаются. Наша пушка также статична. Единственное, что двигается, — это, собственно, само ядро. В целом мы можем разделить объекты в нашей игре на статичные (неподвижные) и динамичные (подвижные). Создадим класс для представления таких объектов.

GameObject, DynamicGameObject и Cannon

Начнем со статичного, или базового, случая в листинге 8.7.

Листинг 8.7. GameObject.java, статичный игровой объект с позицией и границами

```
package com.badlogic.androidgames.gamedev2d;

import com.badlogic.androidgames.framework.math.Rectangle;
import com.badlogic.androidgames.framework.math.Vector2;

public class GameObject {
    public final Vector2 position;
    public final Rectangle bounds;

    public GameObject(float x, float y, float width, float height) {
        this.position = new Vector2(x,y);
        this.bounds = new Rectangle(x-width/2, y-height/2, width, height);
    }
}
```

Каждый объект в нашей игре имеет позицию, которая совпадает с его центром. Мы также разрешаем каждому объекту иметь одну ограничивающую фигуру — в данном случае прямоугольник. В конструкторе определим позицию и ограничивающую фигуру (которая сосредоточена вокруг центра объекта) в соответствии с его параметрами.

Для динамических объектов, то есть объектов, которые двигаются, нам также необходимо отслеживать скорость и ускорение (если они ускоряются сами по себе, например мотор или двигатель). В листинге 8.8 показан код для класса DynamicGameObject.

Листинг 8.8. DynamicGameObject.java: расширение GameObject при добавлении вектора скорости и ускорения

```
package com.badlogic.androidgames.gamedev2d;

import com.badlogic.androidgames.framework.math.Vector2;

public class DynamicGameObject extends GameObject {
    public final Vector2 velocity;
```

```
    public final Vector2 accel;

    public DynamicGameObject(float x, float y, float width, float height) {
        super(x, y, width, height);
        velocity = new Vector2();
        accel = new Vector2();
    }
}
```

Мы дополняем класс GameObject, чтобы он наследовал члены позиции и границ. Дополнительно создаем векторы скорости и ускорения. Новый динамический игровой объект имеет нулевые скорость и ускорение после инициализации.

В нашем примере с пушечным ядром есть пушка, ядро и мишени. Пушечное ядро является DynamicGameObject, поскольку оно движется согласно нашей простой физической модели. Мишени статичны и могут быть реализованы с использованием стандартного GameObject. Сама пушка может быть реализована с помощью класса GameObject. Из класса GameObject мы произведем класс Cannon и добавим поле, содержащее текущий угол наклона пушки. Код приведен в листинге 8.9.

Листинг 8.9. Cannon.java: расширение GameObject при введении Angle

```
package com.badlogic.androidgames.gamedev2d;

public class Cannon extends GameObject {
    public float angle;

    public Cannon(float x, float y, float width, float height) {
        super(x, y, width, height);
        angle = 0;
    }
}
```

Так мы красиво инкапсулируем все данные, необходимые для представления объекта в нашем мире. Каждый раз, когда нам нужен объект определенного вида, например пушка, мы можем просто извлекать его из GameObject, если это статический объект, или из DynamicGameObject, если у него есть скорость и ускорение.

ПРИМЕЧАНИЕ

Переизбыток расширений классов может привести к лишней головной боли и некрасивой архитектуре кода. Не используйте дополнения только ради того, чтобы использовать. Простая иерархия классов, которую мы применяли выше, не беда, но не стоит позволять ей становиться глубже (например, не стоит дополнять класс Cannon). Существуют альтернативные представления игровых объектов, которые позволяют избавиться от всего наследования благодаря правильной композиции. Однако для наших целей и простого дополнения более чем достаточно. Если вы хотите больше узнать о других способах представления, поищите в Интернете статьи о составных объектах, или примесях.

Пространственная сетка

Наша пушка будет ограничена прямоугольником размером 1×1 м; ограничивающий прямоугольник пушечного ядра будет размером $0{,}2 \times 0{,}2$ м, а у мишеней будут ограничивающие прямоугольники по $0{,}5 \times 0{,}5$ м каждый. Чтобы все стало

немного проще, ограничивающие прямоугольники центрированы относительно позиции каждого объекта.

Когда мы запускаем наш пример с пушкой, мы просто располагаем некоторое количество мишеней в произвольных позициях. Вот как мы можем разместить объекты в нашем мире:

```
Cannon cannon = new Cannon(0, 0, 1, 1);
DynamicGameObject ball = new DynamicGameObject(0, 0, 0.2f, 0.2f);
GameObject[] targets = new GameObject[NUM_TARGETS];
for(int i = 0; i < NUM_TARGETS; i++) {
    targets[i] = new GameObject((float)Math.random() * WORLD_WIDTH,
                                (float)Math.random() * WORLD_HEIGHT,
                                0.5f, 0.5f);
}
```

Константы `WORLD_WIDTH` и `WORLD_HEIGHT` определяют размеры игрового мира. Все действия должны происходить в прямоугольнике, описанном координатами (0; 0) и (`WORLD_WIDTH`, `WORLD_HEIGHT`). На рис. 8.17 показан небольшой макет нашего игрового мира.

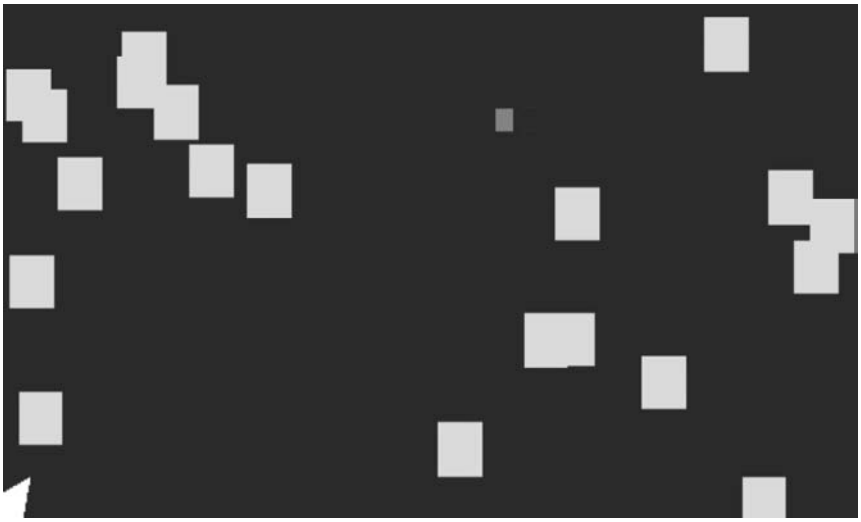

Рис. 8.17. Макет нашего игрового мира

Так наш мир будет выглядеть позже, но пока наложим на него пространственную сетку. Насколько большими должны быть ячейки пространственной сетки? Здесь нет однозначного ответа, но я обычно выбираю такой размер, чтобы ячейка была в пять раз больше самого большого объекта в кадре. В нашем примере самый большой объект — это пушка, но с пушкой мы ничего не сталкиваем. Поэтому лучше определить размер ячейки сетки в соответствии со следующими по величине объектами — мишенями. Мишени у нас размером 0,5 × 0,5 м. Тогда ячейка сетки должна иметь размер 2,5 × 2,5 м. На рис. 8.18 показана сетка, наложенная на наш мир.

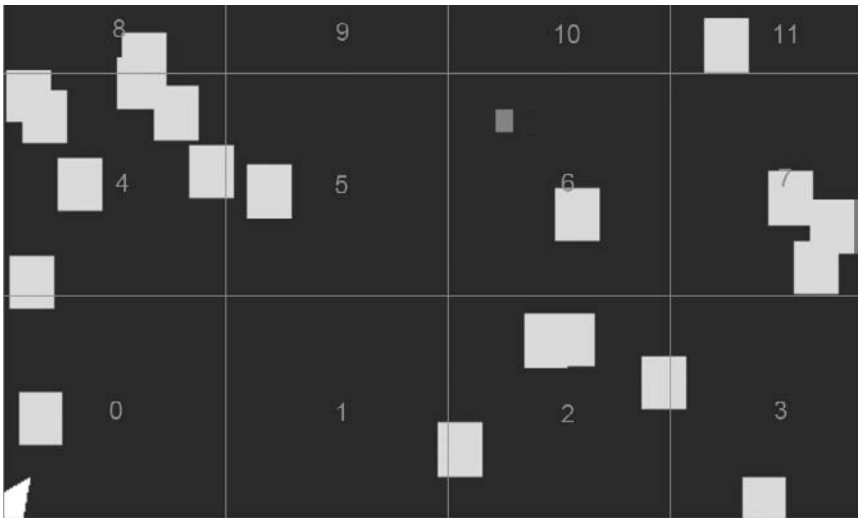

Рис. 8.18. Наш игровой мир, на который наложена пространственная сетка из 12 ячеек

У нас есть определенное количество ячеек — в случае с этим пушечным миром их 12. Мы присваиваем каждой ячейке уникальный номер, начиная с нижней левой, которая получает ID 0. Обратите внимание, что верхние ячейки выходят за пределы нашего мира. Это не проблема; стоит только убедиться, что все объекты нашего мира находятся в его пределах. Мы хотим определить, к какой ячейке или ячейкам принадлежит объект. В идеале мы хотим вычислить ID ячеек, в которых содержится объект. Это позволяет нам использовать простую структуру данных для хранения ячеек:

```
List<GameObject>[] cells;
```

Да, именно, мы описываем каждую ячейку как список классов `GameObject`. Сама пространственная сетка, таким образом, представляет собой массив списков классов `GameObject`.

Давайте подумаем, как мы можем определить ID ячеек, в которых находится объект. На рис. 8.18 показаны несколько мишеней, которые охватывают две ячейки. На самом деле маленький объект может располагаться одновременно в 4 ячейках, а объект, больший по размерам, чем ячейка сетки, может занимать даже больше четырех ячеек. Мы можем гарантировать, что это не может произойти, выбрав в качестве размера ячейки нашей сетки число, кратное размеру наибольшего объекта нашей игры. После этого останется только вероятность, что один объект может содержаться максимум в четырех ячейках.

Чтобы вычислить ID ячейки для объекта, мы можем взять точки в углах его ограничивающего прямоугольника и проверить, в какой ячейке находится каждая точка. Определить ячейку, в которой находится точка, просто: нужно разделить ее координаты на ширину ячейки в первую очередь. Например, у нас есть точка

с координатами (3; 4) и ячейка размером 2,5 × 2,5 м. Точка будет находиться в ячейке с ID 5 на рис. 8.18.

Мы можем разделить каждую координату точки на размер ячейки, чтобы получить целочисленные 2D-координаты, как здесь:

```
cellX = floor(point.x / cellSize) = floor(3 / 2.5) = 1
cellY = floor(point.y / cellSize) = floor(4 / 2.5) = 1
```

И из этих координат ячейки мы можем просто получить ID ячейки:

```
cellId = cellX + cellY * cellsPerRow = 1 + 1 * 4 = 5
```

Константа `cellsPerRow` представляет собой количество ячеек, которое нужно, чтобы покрыть наш мир ячейками по оси *x*:

```
cellsPerRow = ceil(worldWidth / cellSize) = ceil(9.6 / 2.5) = 4
```

Можно вычислить, сколько ячеек приходится на столбец, вот так:

```
cellsPerColumn = ceil(worldHeight / cellSize) = ceil(6.4 / 2.5) = 3
```

Основываясь на этом, мы можем реализовать пространственную сетку достаточно легко. Мы наносим ее, давая ей размеры нашего мира и определяя размер ячейки. Мы допускаем, что все действия происходят в положительном квадранте мира. Это означает, что все *x*- и *y*-координаты точек в мире должны быть положительными. Такое ограничение для нас вполне приемлемо.

Из параметров пространственная сетка может определить, сколько ее ячеек необходимо (`cellsPerRow` × `cellsPerColomn`). Мы также можем добавить простой метод для вставки объекта в сетку, который будет использовать границы объекта для определения ячеек, в которых он содержится. В этом случае объект будет добавляться в список объектов каждой ячейки, в которой он содержится. Если одна из угловых точек ограничивающей фигуры объекта находится за пределами сетки, мы просто игнорируем эту угловую точку.

Мы будем заново вставлять каждый объект в пространственную сетку в каждом кадре после того, как будем обновлять его позицию. Однако в нашем пушечном мире есть объекты, которые не двигаются, поэтому вставлять их заново в каждый кадр очень неэкономно. В таком случае можно сделать разделение объектов на динамические и статические, заведя два списка для каждой ячейки. Один будет обновляться каждый кадр и содержать только динамические объекты, а другой будет статичен и будет изменяться только в случае добавления нового статического объекта.

И наконец, нам нужен метод, который будет возвращать список объектов в ячейках для объекта, который мы хотим столкнуть с другими объектами. Этот метод будет проверять, в каких ячейках располагается интересующий нас объект, находить список динамических и статических объектов в этих ячейках и возвращать их вызывающей стороне. Конечно, нужно будет убедиться, что мы не будем возвращать дубликаты, которые могут появляться, если объект находится в нескольких ячейках. В листинге 8.10 показана большая часть кода. Поскольку метод `SpatialHarshGrid getCellds()` немного запутанный, мы обсудим его позже.

Листинг 8.10. Фрагмент класса SpatialHashGrid.java: реализация пространственной сетки

```
package com.badlogic.androidgames.framework.gl;

import java.util.ArrayList;
import java.util.List;

import com.badlogic.androidgames.gamedev2d.GameObject;

import android.util.FloatMath;

public class SpatialHashGrid {
    List<GameObject>[] dynamicCells;
    List<GameObject>[] staticCells;
    int cellsPerRow;
    int cellsPerCol;
    float cellSize;
    int[] cellIds = new int[4];
    List<GameObject> foundObjects;
```

Как уже обсуждалось, мы сохраняем два списка ячеек: один — для динамических объектов, второй — для статических. Мы также храним количество ячеек в ряду и столбце, чтобы позже иметь возможность проверить, находится ли точка, которую мы проверяем, внутри или за пределами нашего мира. Размер ячейки также следует сохранить. Массив `cellsds` — это рабочий массив, который мы используем для временного хранения ID четырех ячеек, в которых содержится `GameObject`. Если он находится только в одной ячейке, то только первому элементу массива будет присвоен ID ячейки, которая полностью содержит объект. Если объект располагается в двух ячейках, первые два элемента массива будут содержать ID ячеек и т. д. Чтобы обозначить количество ID ячеек, мы присвоим всем «пустым» элементам массива значение –1. Список `foundObjects` также рабочий, который мы будем возвращать по вызову `getPotentialColliders()`. Зачем держать эти два элемента, если можно создавать новый массив и список каждый раз, когда будет требоваться? Вспомните нашего старого знакомого — страшного сборщика мусора.

```
@SuppressWarnings("unchecked")
public SpatialHashGrid(float worldWidth, float worldHeight, float cellSize) {
    this.cellSize = cellSize;
    this.cellsPerRow = (int)FloatMath.ceil(worldWidth/cellSize);
    this.cellsPerCol = (int)FloatMath.ceil(worldHeight/cellSize);
    int numCells = cellsPerRow * cellsPerCol;
    dynamicCells = new List[numCells];
    staticCells = new List[numCells];
    for(int i = 0; i < numCells; i++) {
        dynamicCells[i] = new ArrayList<GameObject>(10);
        staticCells[i] = new ArrayList<GameObject>(10);
    }
    foundObjects = new ArrayList<GameObject>(10);
}
```

Конструктор этого класса берет размер мира и желаемый размер ячейки. По данным параметрам мы вычисляем, сколько ячеек нам надо, и заполняем массивы ячеек и списки, содержащие объекты, находящиеся в каждой ячейке. Мы также инициализируем список foundObjects. Все созданные нами списки ArrayList будут иметь исходный размер в 10 GameObject. Мы предполагаем, что вероятность нахождения 10 GameObject в одной ячейке очень мала. Пока это так, массивы не требуют изменения в размерах.

```
public void insertStaticObject(GameObject obj) {
    int[] cellIds = getCellIds(obj);
    int i = 0;
    int cellId = -1;
    while(i <= 3 && (cellId = cellIds[i++]) != -1) {
        staticCells[cellId].add(obj);
    }
}

public void insertDynamicObject(GameObject obj) {
    int[] cellIds = getCellIds(obj);
    int i = 0;
    int cellId = -1;
    while(i <= 3 && (cellId = cellIds[i++]) != -1) {
        dynamicCells[cellId].add(obj);
    }
}
```

Далее следуют методы insertStaticObject() и insertDynamicObject. Они вычисляют ID ячеек, в которых содержится объект, путем вызова getCellIds() и вставляют объект в подходящие списки. Метод getCellIds() будет заполнять массив переменных cellIds.

```
public void removeObject(GameObject obj) {
    int[] cellIds = getCellIds(obj);
    int i = 0;
    int cellId = -1;
    while(i <= 3 && (cellId = cellIds[i++]) != -1) {
        dynamicCells[cellId].remove(obj);
        staticCells[cellId].remove(obj);
    }
}
```

Нам также потребуется метод removeObject(), который мы используем для выяснения, в каких ячейках содержится объект, и затем для удаления его из списков динамических и статических объектов соответственно. Это понадобится, например, если игровой объект умирает.

```
public void clearDynamicCells(GameObject obj) {
    int len = dynamicCells.length;
    for(int i = 0; i < len; i++) {
        dynamicCells[i].clear();
    }
}
```

Метод clearDynamicCells() будет использоваться для очистки всех списков динамических объектов. Нам нужно будет вызывать этот метод перед каждым кадром, прежде чем мы будем заново вставлять динамические объекты, что уже обсуждалось ранее.

```
public List<GameObject> getPotentialColliders(GameObject obj) {
    foundObjects.clear();
    int[] cellIds = getCellIds(obj);
    int i = 0;
    int cellId = -1;
    while(i <= 3 && (cellId = cellIds[i++]) != -1) {
        int len = dynamicCells[cellId].size();
        for(int j = 0; j < len; j++) {
            GameObject collider = dynamicCells[cellId].get(j);
            if(!foundObjects.contains(collider))
                foundObjects.add(collider);
        }

        len = staticCells[cellId].size();
        for(int j = 0; j < len; j++) {
            GameObject collider = staticCells[cellId].get(j);
            if(!foundObjects.contains(collider))
                foundObjects.add(collider);
        }
    }
    return foundObjects;
}
```

И наконец, есть еще метод getPotentialColliders(). Он берет объект и возвращает список соседних от него объектов, которые содержатся в тех же ячейках, что и этот объект. Мы будем использовать рабочий список foundObjects для хранения списка найденных объектов. Опять же, мы не хотим создавать новый список каждый раз, когда вызываем этот метод. Все, что нам надо выяснить, — это в каких ячейках находится объект, переданный в метод. Затем мы просто добавляем все статические и динамические объекты, найденные в этих ячейках, в список foundObjects и убеждаемся, что в нем нет дубликатов. Применять foundObjects.contains() для проверки на дубликаты, конечно, не совсем оптимально. Но поскольку количество найденных объектов в данном случае никогда не будет большим, он вполне приемлем. Если у вас возникнут проблемы при реализации, это первое, что стоит оптимизировать. К сожалению, это не так и просто. Конечно, мы можем задействовать Set, но тогда новые объекты будут создаваться внутри каждый раз, когда мы будем добавлять объект в Set. Пока оставим все как есть, запомнив, к чему вернуться в случае чего.

Я не упомянул метод SpatialHashGrid.getCellIds(). В листинге 8.11 показан код. Не бойтесь, он не такой страшный, каким кажется на первый взгляд.

Листинг 8.11. Остаток файла SpatialHashGrid.java: реализация getCellIds()

```
    public int[] getCellIds(GameObject obj) {
        int x1 = (int)FloatMath.floor(obj.bounds.lowerLeft.x / cellSize);
        int y1 = (int)FloatMath.floor(obj.bounds.lowerLeft.y / cellSize);
```

```
        int x2 = (int)FloatMath.floor((obj.bounds.lowerLeft.x +
                                       obj.bounds.width) / cellSize);
        int y2 = (int)FloatMath.floor((obj.bounds.lowerLeft.y +
                                       obj.bounds.height) / cellSize);

        if(x1 == x2 && y1 == y2) {
            if(x1 >= 0 && x1 < cellsPerRow && y1 >= 0 && y1 < cellsPerCol)
                cellIds[0] = x1 + y1 * cellsPerRow;
            else
                cellIds[0] = -1;
            cellIds[1] = -1;
            cellIds[2] = -1;
            cellIds[3] = -1;
        }
        else if(x1 == x2) {
            int i = 0;
            if(x1 >= 0 && x1 < cellsPerRow) {
                if(y1 >= 0 && y1 < cellsPerCol)
                    cellIds[i++] = x1 + y1 * cellsPerRow;
                if(y2 >= 0 && y2 < cellsPerCol)
                    cellIds[i++] = x1 + y2 * cellsPerRow;
               }
            while(i <= 3) cellIds[i++] = -1;
        }
        else if(y1 == y2) {
            int i = 0;
            if(y1 >= 0 && y1 < cellsPerCol) {
                if(x1 >= 0 && x1 < cellsPerRow)
                    cellIds[i++] = x1 + y1 * cellsPerRow;
                if(x2 >= 0 && x2 < cellsPerRow)
                    cellIds[i++] = x2 + y1 * cellsPerRow;
            }
            while(i <= 3) cellIds[i++] = -1;
        }
        else {
            int i = 0;
            int y1CellsPerRow = y1 * cellsPerRow;
            int y2CellsPerRow = y2 * cellsPerRow;
            if(x1 >= 0 && x1 < cellsPerRow && y1 >= 0 && y1 < cellsPerCol)
                cellIds[i++] = x1 + y1CellsPerRow;
            if(x2 >= 0 && x2 < cellsPerRow && y1 >= 0 && y1 < cellsPerCol)
                cellIds[i++] = x2 + y1CellsPerRow;
            if(x2 >= 0 && x2 < cellsPerRow && y2 >= 0 && y2 < cellsPerCol)
                cellIds[i++] = x2 + y2CellsPerRow;
            if(x1 >= 0 && x1 < cellsPerRow && y2 >= 0 && y2 < cellsPerCol)
                cellIds[i++] = x1 + y2CellsPerRow;
            while(i <= 3) cellIds[i++] = -1;
        }
        return cellIds;
    }

}
```

Первые четыре строки этого метода вычисляют координаты ячейки, в которой находятся нижний левый и верхний правый углы ограничивающей фигуры объекта. Мы уже обсуждали эти подсчеты ранее. Чтобы понять остальную часть метода, нужно представить, каким образом объект может перекрывать ячейки сетки. Есть четыре варианта.

- Объект находится в одной клетке. Нижний левый и верхний правый углы ограничивающего прямоугольника, таким образом, имеют одинаковые координаты ячейки.
- Объект перекрывает две ячейки горизонтально. Нижний левый угол находится в одной ячейке, в то время как верхний правый угол располагается в ячейке справа.
- Объект занимает две ячейки по вертикали. Нижний левый угол находится в одной ячейке, в то время как верхний правый угол располагается в ячейке сверху.
- Объект находится в четырех ячейках. Нижний левый угол располагается в одной ячейке, нижний правый угол находится в ячейке справа, верхний правый угол размещается на ячейку выше, а верхний левый угол находится на ячейку выше первой ячейки.

Этот метод по отдельности проверяет каждый из частных случаев. Первый блок `if` проверяет случай с одной ячейкой, второй — случай для двух горизонтально расположенных ячеек, третий — для двух вертикально размещенных ячеек, и последний блок для случая, если объект перекрывает четыре ячейки сетки. В каждом из четырех блоков мы убеждаемся, что назначаем ID ячейки только в том случае, если соответствующие координаты ячейки находятся внутри игрового мира. Вот и все, из чего состоит этот метод .

Весь метод наводит на мысль, что для его выполнения потребуется большая вычислительная мощность. И на самом деле так и есть, но не настолько большая, как кажется на первый взгляд. Самым распространенным случаем будет первый, а его обработка не требует много ресурсов. Можете сами придумать, как еще оптимизировать этот метод?

Все вместе

Применим все знания, приобретенные в этом разделе, на маленьком хорошем примере. Разовьем пример с пушкой, который мы обсуждали несколько страниц назад. Используем объект `Cannon` для пушки, объект `DynamicGameObject` для пушечного ядра и несколько объектов `GameObject` для мишеней. Каждая мишень будет иметь размеры $0{,}5 \times 0{,}5$ м, и они будут расположены в мире в произвольном порядке.

Мы хотим стрелять по этим мишенями. Для этого нам понадобится определение столкновений. Мы можем просто перебрать все мишени и проверить их относительно пушечного ядра, но это будет слишком скучно. Мы используем наш замечательный новый класс `SpatialHashGrid`, чтобы ускорить нахождение потенциально сталкивающихся с ядром мишеней относительно текущей позиции ядра. Однако мы не будем вставлять в сетку ядро или пушку, поскольку это ничего нам не даст.

Поскольку этот пример уже достаточно большой, разделим его на несколько листингов. Назовем тест CollisionTest и соответствующий экран CollisionScreen. Как обычно, мы смотрим только на экран. Начнем с элементов теста и конструктора в листинге 8.12.

Листинг 8.12. Фрагмент CollisionTest.java: элементы и конструктор

```
class CollisionScreen extends Screen {
    final int NUM_TARGETS = 20;
    final float WORLD_WIDTH = 9.6f;
    final float WORLD_HEIGHT = 4.8f;
    GLGraphics glGraphics;
    Cannon cannon;
    DynamicGameObject ball;
    List<GameObject> targets;
    SpatialHashGrid grid;

    Vertices cannonVertices;
    Vertices ballVertices;
    Vertices targetVertices;

    Vector2 touchPos = new Vector2();
    Vector2 gravity = new Vector2(0,-10);

    public CollisionScreen(Game game) {
        super(game);
        glGraphics = ((GLGame)game).getGLGraphics();

        cannon = new Cannon(0, 0, 1, 1);
        ball = new DynamicGameObject(0, 0, 0.2f, 0.2f);
        targets = new ArrayList<GameObject>(NUM_TARGETS);
        grid = new SpatialHashGrid(WORLD_WIDTH, WORLD_HEIGHT, 2.5f);
        for(int i = 0; i < NUM_TARGETS; i++) {
            GameObject target = new GameObject((float)Math.random() *
                                                    WORLD_WIDTH,
                                               (float)Math.random() *
                                                    WORLD_HEIGHT,
                                                0.5f, 0.5f);
            grid.insertStaticObject(target);
            targets.add(target);
        }

        cannonVertices = new Vertices(glGraphics, 3, 0, false, false);
        cannonVertices.setVertices(new float[] { -0.5f, -0.5f,
                                                   0.5f,  0.0f,
                                                  -0.5f,  0.5f }, 0, 6);

        ballVertices = new Vertices(glGraphics, 4, 6, false, false);
        ballVertices.setVertices(new float[] { -0.1f, -0.1f,
                                                0.1f, -0.1f,
                                                0.1f,  0.1f,
                                               -0.1f,  0.1f }, 0, 8);
```

```
        ballVertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);

        targetVertices = new Vertices(glGraphics, 4, 6, false, false);
        targetVertices.setVertices(new float[] { -0.25f, -0.25f,
                                                  0.25f, -0.25f,
                                                  0.25f,  0.25f,
                                                 -0.25f,  0.25f }, 0, 8);
        targetVertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);
    }
```

Мы многое заимствовали из `CannonGravityScreen`. Начинаем с определения нескольких констант, указывая количество мишеней и размеры мира. Далее идет экземпляр класса `GLGraphics` и объекты для пушки, ядра и мишеней, которые мы помещаем в список. Есть также `SpatialHashGrid`, конечно. Для визуализации мира нам понадобится несколько сеток: одна для пушки, одна для ядра и одну мы будем использовать для отображения каждого объекта. Вспомните, что в `BobTest` мы применяли только одну треугольную сетку для визуализации на экране ста Бобов. Мы задействуем этот принцип и здесь вместо того, чтобы заводить по экземпляру `Vertices` для каждой мишени. Последние два члена класса те же, что и в `CannonGravityTest`. Мы используем их, чтобы стрелять ядром и применять законы гравитации, когда пользователь дотрагивается до экрана.

Конструктор просто выполняет все те операции, которые мы уже обсудили ранее. Мы инициализируем объекты и сети нашего мира. Единственное, что здесь интересно, — добавление мишеней в пространственную сетку как статических объектов.

Рассмотрим следующий метод в классе `CollisionTest` (листинг 8.13).

Листинг 8.13. Фрагмент `CollisionTest.java`: метод `update()`

```
@Override
public void update(float deltaTime) {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    game.getInput().getKeyEvents();

    int len = touchEvents.size();
    for (int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);

        touchPos.x = (event.x / (float) glGraphics.getWidth())* WORLD_WIDTH;
        touchPos.y = (1 – event.y / (float) glGraphics.getHeight()) *
                                                    WORLD_HEIGHT;

        cannon.angle = touchPos.sub(cannon.position).angle();

        if(event.type == TouchEvent.TOUCH_UP) {
            float radians = cannon.angle * Vector2.TO_RADIANS;
            float ballSpeed = touchPos.len() * 2;
            ball.position.set(cannon.position);
            ball.velocity.x = FloatMath.cos(radians) * ballSpeed;
            ball.velocity.y = FloatMath.sin(radians) * ballSpeed;
            ball.bounds.lowerLeft.set(ball.position.x – 0.1f,
```

```
                                        ball.position.y - 0.1f);
            }
        }

        ball.velocity.add(gravity.x * deltaTime, gravity.y * deltaTime);
        ball.position.add(ball.velocity.x * deltaTime, ball.velocity.y *
                                                        deltaTime);
        ball.bounds.lowerLeft.add(ball.velocity.x * deltaTime, ball.velocity.y *
                                                        deltaTime);

        List<GameObject> colliders = grid.getPotentialColliders(ball);
        len = colliders.size();
        for(int i = 0; i < len; i++) {
            GameObject collider = colliders.get(i);
            if(OverlapTester.overlapRectangles(ball.bounds, collider.bounds)) {
                grid.removeObject(collider);
                targets.remove(collider);
            }
        }
    }
```

Как обычно, мы сначала выбираем события касания и события нажатия клавиш и обрабатываем только события касания. Обработка случаев касания практически такая же, как в `CannonGravityTest`. Вся разница в том, что вместо векторов, которые мы использовали в старом примере, мы применяем объект `Cannon`, а также возвращаем в исходное положение ограничивающий прямоугольник ядра, когда пушка готова стрелять в случае касания.

Следующая перемена заключается в том, каким образом мы обновляем ядро. Вместо векторов используем члены класса `DynamicGameObject`, которые создали для ядра. Мы пренебрегаем ускорением из `DynamicGameObject` и вместо него добавляем наше ускорение силы тяжести к скорости ядра. Мы также умножаем скорость ядра на два, чтобы оно летело немного быстрее. Интересно то, что мы обновляем не только позицию ядра, но и координаты нижнего левого угла ограничивающего прямоугольника. Это очень важно, потому что иначе наше ядро будет двигаться, а его ограничивающий прямоугольник нет. Почему бы нам просто не использовать ограничивающий прямоугольник для хранения позиции ядра? Вспомните, что мы можем применять несколько ограничивающих фигур для одного объекта. Какая же тогда фигура будет определять позицию объекта? Поэтому разделение этих двух понятий выгодно, да и требует небольших вычислительных издержек. Конечно, мы можем провести еще некоторую оптимизацию, умножив скорость на дельту времени. Издержки тогда сократятся до еще двух операций сложения, но это слишком малая цена за гибкость, которую мы имеем.

Последняя часть метода — код определения столкновений. Нам нужно найти в пространственной сетке мишени, которые находятся в тех же ячейках, что и ядро. Для этого используем метод `SpatialHashGrid.getPotentialColliders()`. Поскольку ячейки, в которых находится ядро, рассчитываются прямо в методе, нет необходимости вставлять ядро в сетку. Далее проходим в цикле по всем объектам, с кото-

рыми возможно столкновение, и проверяем, есть ли действительно пересечение ограничивающего прямоугольника ядра с ограничивающим прямоугольником объекта, с которым возможно столкновение. Если есть, мы просто удаляем эту мишень из списка мишеней. Помните, что мы добавили мишени в сетку только как статические объекты.

И все это и есть законченный механизм нашей игры. Последний кусок головоломки, по сути, и есть сам этап визуализации, который вряд ли вас чем-то удивит (листинг 8.14).

Листинг 8.14. Фрагмент CollisionTest.java: метод present()

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    gl.glOrthof(0, WORLD_WIDTH, 0, WORLD_HEIGHT, 1, -1);
    gl.glMatrixMode(GL10.GL_MODELVIEW);

    gl.glColor4f(0, 1, 0, 1);
    targetVertices.bind();
    int len = targets.size();
    for(int i = 0; i < len; i++) {
        GameObject target = targets.get(i);
        gl.glLoadIdentity();
        gl.glTranslatef(target.position.x, target.position.y, 0);
        targetVertices.draw(GL10.GL_TRIANGLES, 0, 6);
    }
    targetVertices.unbind();

    gl.glLoadIdentity();
    gl.glTranslatef(ball.position.x, ball.position.y, 0);
    gl.glColor4f(1,0,0,1);
    ballVertices.bind();
    ballVertices.draw(GL10.GL_TRIANGLES, 0, 6);
    ballVertices.unbind();

    gl.glLoadIdentity();
    gl.glTranslatef(cannon.position.x, cannon.position.y, 0);
    gl.glRotatef(cannon.angle, 0, 0, 1);
    gl.glColor4f(1,1,1,1);
    cannonVertices.bind();
    cannonVertices.draw(GL10.GL_TRIANGLES, 0, 3);
    cannonVertices.unbind();
}
```

Здесь нет ничего нового. Как и всегда, мы определяем проекционную матрицу и область просмотра и сначала очищаем экран. Далее отображаем все мишени, вновь используя прямоугольную модель, сохраненную в `targetVertices`. Это, по существу,

то же, что мы делали в BobTest, но на этот раз мы отображаем мишени. Далее отображаем ядро и пушку, как мы делали в CollisionGravityTest.

Я изменил порядок расположения изображений, чтобы ядро всегда было над мишенями, а пушка всегда была над ядром. Я также сделал мишени зелеными, вызвав glColor4f().

Результат этого небольшого теста точно такой же, как показано на рис. 8.17. Когда вы стреляете из пушки ядром, оно будет прокладывать себе путь через поле с мишенями. Любая мишень, задетая ядром, будет удалена из мира.

Этот пример может стать хорошей игрой, если мы немного усовершенствуем его и добавим несколько мотивирующих игровых механизмов. Попробуйте сами придумать добавления. Я думаю, что для вас хорошо было бы поэкспериментировать с этим примером, чтобы опробовать все те новые средства, которые мы изучили на протяжении последних страниц.

Есть еще несколько вещей, которые я хотел бы обсудить в этой главе: камеры, атласы текстур и спрайты. Они используют некоторые трюки, имеющие отношение к графике, и не зависят от нашей модели игрового мира. Приступим!

Камера в 2D

До этого момента у нас не было понятия камеры в нашем коде; мы только определили область видимости камеры (конус отображения) при помощи glOrthof():

```
gl.glMatrixMode(GL10.GL_PROJECTION);
gl.glLoadIdentity();
gl.glOrthof(0, FRUSTUM_WIDTH, 0, FRUSTUM_HEIGHT, 1, -1);
```

Из главы 6 вы знаете, что первые два параметра определяют x-координаты левого и правого краев конуса отображения в нашем мире, следующие два параметра задают y-координаты нижнего и верхнего краев конуса отображения, а последние два параметра указывают ближнюю и дальнюю плоскости отсечения. На рис. 8.19 показана область видимости камеры.

Таким образом, мы видим только область от (0; 0; 1) до (FRUSTUM_WIDTH; FRUSTUM_HEIGHT; –1) нашего мира. Как думаете, хорошо было бы иметь возможность сдвинуть область видимости, скажем, влево? Конечно, это было бы неплохо, а это ведь очень просто:

```
gl.glOrthof(x, x + FRUSTUM_WIDTH, 0, FRUSTUM_HEIGHT, 1, -1);
```

В данном случае x — это просто какая-то начальная точка, которую мы можем определить. Мы также можем произвести сдвиги по x- и y-осям:

```
gl.glOrthof(x, x + FRUSTUM_WIDTH, y, y + FRUSTUM_HEIGHT, 1, -1);
```

На рис. 8.20 показано, что имеется в виду.

Так, мы просто обозначаем нижний левый угол конуса отображения нашей камеры в пространстве мира. Этого уже достаточно для реализации свободно движущейся 2D-камеры. Но мы можем сделать ее еще лучше. Как насчет определения

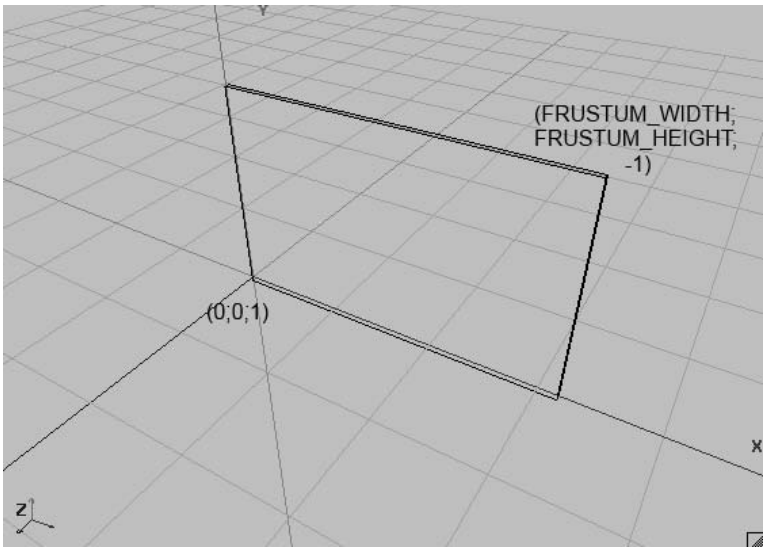

Рис. 8.19. Область видимости камеры для нашего 2D-мира

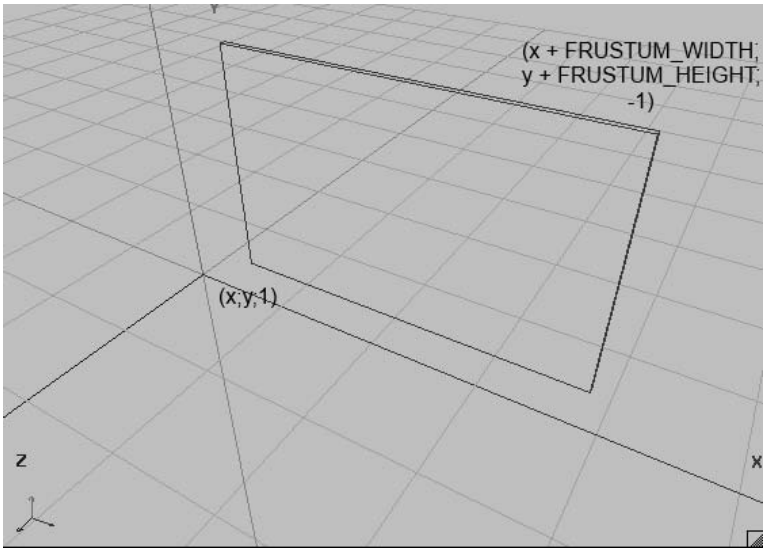

Рис. 8.20. Сдвиг области видимости камеры

не нижнего левого угла области видимости точками *x* и *y*, а центра области видимости? Таким образом мы могли бы просто размещать центр нашей области видимости на объекте в специфическом положении — скажем, пушечном ядре из предыдущего примера:

```
gl.glOrthof(x – FRUSTUM_WIDTH / 2, x + FRUSTUM_WIDTH / 2, y –
FRUSTUM_HEIGHT / 2, y + FRUSTUM_HEIGHT / 2, 1, -1);
```

На рис. 8.21 показано, как это выглядит.

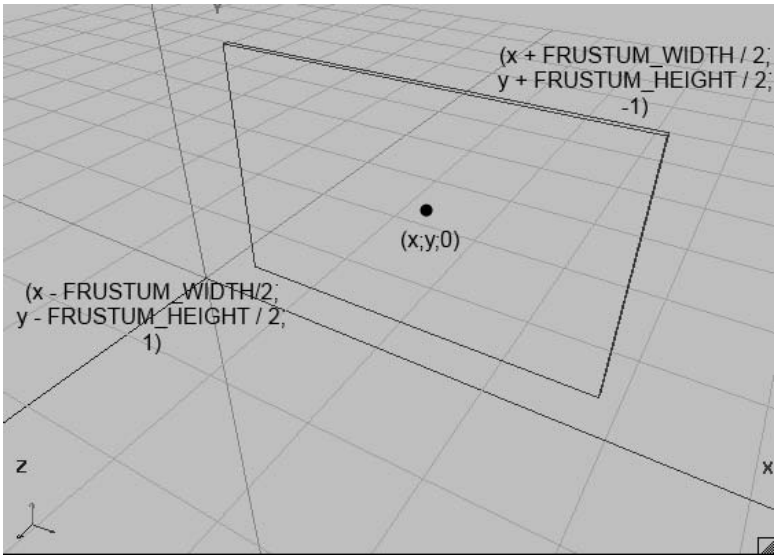

Рис. 8.21. Определение центра конуса отображения через его центр

И это еще не все, что можно сделать с `glOrthof()`. Как насчет изменения масштаба? Мы знаем, что благодаря `glViewportf()` мы можем сообщить OpenGL ES, на какую часть экрана демонстрировать содержимое нашего конуса отображения. OpenGL ES автоматически растянет и изменит масштаб итогового изображения, согласовывая его с областью просмотра (viewport). Если мы сделаем ширину и длину нашего конуса отображения меньше, мы просто покажем меньшую область мира на экране. Это увеличение масштаба. Если мы делаем область больше, мы показываем большую часть нашего мира — это уменьшение масштаба. Таким образом, мы можем ввести переменную масштаба и умножать ее на ширину и длину нашего конуса отображения для увеличения или уменьшения масштаба. При множителе, равном 1, мир будет показан, как на рис. 8.21, при использовании нормальных ширины и длины области видимости. При множителе меньше 1 будет происходить уменьшение масштаба к центру конуса отображения. А при множителе больше 1 масштаб будет уменьшаться, показывая большую часть мира (например, при множителе, равном 2, будет видно в 2 раза больше мира). Вот как мы можем использовать `glOrthof()`, чтобы сделать это:

```
gl.glOrthof(x – FRUSTUM_WIDTH / 2 * zoom, x + FRUSTUM_WIDTH / 2 * zoom, y -
FRUSTUM_HEIGHT / 2 * zoom, y + FRUSTUM_HEIGHT / 2 * zoom, 1, -1);
```

Проще простого! Теперь мы можем создать класс камеры с точкой, к которой камера обращена (центр конуса отображения), стандартными шириной и длиной области видимости и переменной масштаба, которая делает конус отображения

больше или меньше, таким образом показывая нам меньшую (увеличение масштаба) или большую (уменьшение масштаба) часть мира. На рис. 8.22 изображена область видимости при множителе масштаба, равном 0,5 (внутренний серый прямоугольник), и та же область при множителе, равном 1 (внешний прозрачный прямоугольник).

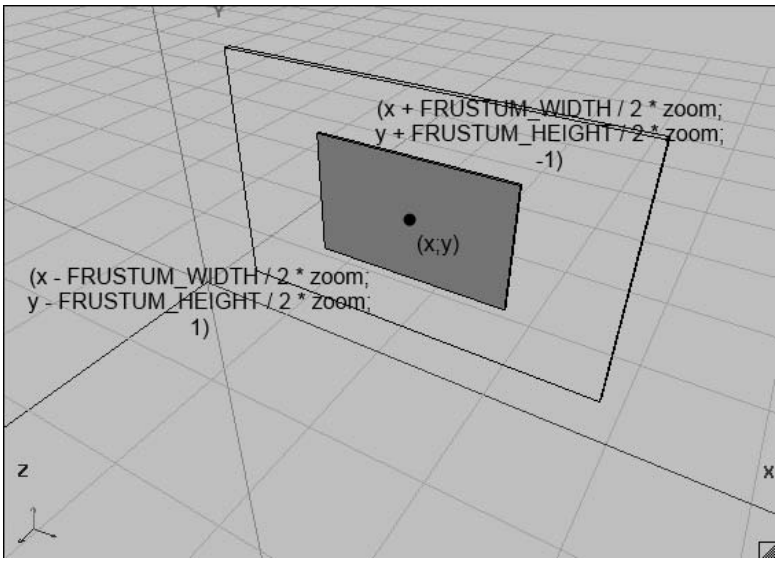

Рис. 8.22. Изменение масштаба с помощью изменения размеров конуса отображения

Для полноты картины нужно сделать еще одну вещь. Представьте, что мы касаемся экрана и хотим выяснить, какой точки в нашем 2D-мире мы коснулись. Мы уже делали такое несколько раз в наших постоянно совершенствующихся примерах с пушкой. При конфигурации конуса отображения, которая не учитывает положение камер и масштаб, как на рис. 8.19, мы имеет следующие уравнения (смотрите метод `update()` в наших примерах с пушкой):

```
worldX = (touchX / Graphics.getWidth()) × FRUSTUM_WIDTH;
worldY = (1 – touchY / Graphics.getHeight()) × FRUSTUM_HEIGHT;
```

Сначала нормализуем координаты x и y, полученные при касании, до промежутка от 0 до 1 путем деления их на ширину и длину экрана. Затем изменяем их масштаб так, чтобы они выражались через пространство нашего мира, умножая их на ширину и длину области видимости. Нам необходимо учесть позицию конуса отображения и переменную масштаба. Вот как мы это делаем:

```
worldX = (touchX / Graphics.getWidth()) . FRUSTUM_WIDTH + x – FRUSTUM_WIDTH / 2;
worldY = (1 – touchY / Graphics.getHeight()) . FRUSTUM_HEIGHT + y – FRUSTUM_HEIGHT / 2;
```

`x` и `y` здесь — положение камеры в пространстве мира.

Класс Camera2D

Объединим все рассмотренные детали работы с камерой в один класс. Мы хотим, чтобы в нем хранились позиция камеры, стандартные ширина и длина области видимости и переменная масштаба. Мы также хотим иметь удобный метод для правильного назначения порта просмотра (всегда использовать весь экран) и матрицы проецирования. Еще нам нужен метод для перевода координат при касании в координаты нашего мира. Наш новый класс Camera2D показан в листинге 8.15.

Листинг 8.15. Camera2D.java: наш новенький класс камеры для рендеринга 2D

```
package com.badlogic.androidgames.framework.gl;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.impl.GLGraphics;
import com.badlogic.androidgames.framework.math.Vector2;

public class Camera2D {
    public final Vector2 position;
    public float zoom;
    public final float frustumWidth;
    public final float frustumHeight;
    final GLGraphics glGraphics;
```

Как мы уже обсуждали, мы сохраняем позицию камеры, ширину и длину конуса отображения и переменную масштаба как члены класса. Члены позиции камеры и переменной масштаба общедоступные, поэтому мы легко можем управлять ими. Нам также нужна ссылка на GLGraphics, чтобы брать оттуда последние известные ширину и длину экрана в пикселах для перевода координат касания в координаты нашего мира.

```
public Camera2D(GLGraphics glGraphics, float frustumWidth,
                                       float frustumHeight) {
    this.glGraphics = glGraphics;
    this.frustumWidth = frustumWidth;
    this.frustumHeight = frustumHeight;
    this.position = new Vector2(frustumWidth / 2, frustumHeight / 2);
    this.zoom = 1.0f;
}
```

В конструкторе мы берем экземпляр класса GLGraphics, ширину и длину конуса отображения при переменной масштаба 1 как параметры. Сохраняем их и инициализируем позицию камеры, чтобы она смотрела в центр прямоугольника, ограниченного координатами (0; 0; 1) и (frustumWidth; frustumHeight; –1), как на рис. 8.19. Исходное значение переменной масштаба равно 1.

```
public void setViewportAndMatrices() {
    GL10 gl = glGraphics.getGL();
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glMatrixMode(GL10.GL_PROJECTION);
```

```
        gl.glLoadIdentity();
        gl.glOrthof(position.x – frustumWidth * zoom / 2,
                    position.x + frustumWidth * zoom/ 2,
                    position.y – frustumHeight * zoom / 2,
                    position.y + frustumHeight * zoom/ 2,
                    1, -1);
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();
    }
```

Метод `setViewportAndMatrices()` задает область просмотра, охватывающую весь экран, а также задает проекционную матрицу в соответствии с параметрами нашей камеры, как мы уже говорили раньше. В конце метода мы указываем OpenGL ES, что все дальнейшие матричные операции направлены на матрицу модели-вида, и загружаем единичную матрицу. Мы будем вызывать этот метод каждый кадр, чтобы можно было начинать с чистого листа. И больше никаких прямых вызовов OpenGL ES для установления области просмотра и проекционной матрицы.

```
    public void touchToWorld(Vector2 touch) {
        touch.x = (touch.x / (float) glGraphics.getWidth()) *
                                            frustumWidth * zoom;
        touch.y = (1 – touch.y / (float) glGraphics.getHeight()) *
                                            frustumHeight * zoom;
        touch.add(position).sub(frustumWidth * zoom / 2,
                                            frustumHeight * zoom / 2);
    }
}
```

Метод `touchToWorld()` принимает член класса `Vector2`, содержащий координаты, которые получены при касании, и переводит вектор в пространство мира. Это то же, что мы уже обсуждали; единственная разница в том, что мы используем наш замечательный класс `Vector2`.

Пример

Давайте используем класс `Camera2D` в нашем примере с пушкой. Я скопировал файл `CollisionTest` и переименовал его в `Camera2DTest`. Я также поменял имя класса `GLGame` внутри файла `Camera2DTest` и переименовал класс `CollisionScreen` в `Camera2DScreen`. Мы обсудим только небольшие изменения, которые надо внести, чтобы использовать наш новый класс `Camera2D`.

Первое, что мы делаем, — добавляем новый член в класс `Camera2DScreen`:

```
Camera2D camera;
```

Инициализируем этот член класса в конструкторе следующим образом:

```
camera = new Camera2D(glGraphics, WORLD_WIDTH, WORLD_HEIGHT);
```

Предъявляем наш экземпляр класса `GLGraphics`, ширину и длину нашего мира, которые мы до этого использовали как ширину и длину конуса отображения

в вызове glOrthof(). Сейчас нужно заменить наши прямые вызовы OpenGL ES в методе present(), который выглядит вот так:

```
gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
gl.glMatrixMode(GL10.GL_PROJECTION);
gl.glLoadIdentity();
gl.glOrthof(0, WORLD_WIDTH, 0, WORLD_HEIGHT, 1, -1);
gl.glMatrixMode(GL10.GL_MODELVIEW);
```

Мы заменяем их этим:

```
gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
camera.setViewportAndMatrices();
```

Нам, конечно, еще нужно очищать кадровый буфер, но все остальные прямые вызовы OpenGL ES замечательно спрятаны в методе Camera2D.setViewportAndMatrices(). Если вы выполните этот код, вы увидите, что ничего не изменилось. Все работает, как и раньше, — все это мы делали, чтобы код был красивее и удобнее.

Мы также можем немного упростить метод теста update(). Поскольку мы добавили в класс камеры метод Camera2D.touchToWorld(), мы можем использовать и его тоже. Мы можем заменить этот фрагмент метода update:

```
touchPos.x = (event.x / (float) glGraphics.getWidth())* WORLD_WIDTH;
touchPos.y = (1 – event.y / (float) glGraphics.getHeight()) * WORLD_HEIGHT;
```

таким:

```
camera.touchToWorld(touchPos.set(event.x, event.y));
```

Вот теперь все прекрасно оформлено. Но было бы скучно так и не воспользоваться всеми возможностями нашего класса камеры. План таков: мы хотим, чтобы камера смотрела на мир, так сказать, нормально, пока пушечное ядро не летит. Это просто: мы уже делаем это. Мы можем определить, летит ли ядро, проверив, действительно ли y-координата его позиции меньше или равна нулю. Поскольку мы всегда применяем к ядру силу тяжести, оно, конечно, будет падать, даже если мы не стреляем им, поэтому это простой способ проверки.

Наше нововведение станет заметным, когда ядро будет лететь (когда y-координата будет больше нуля). Мы хотим, чтобы камера следила за ядром. Этого можно достичь, установив позицию камеры на позицию ядра. Это позволит всегда держать ядро в центре экрана. Также хорошо бы опробовать функционал масштабирования. Для этого будем увеличивать переменную масштаба в зависимости от y-координаты ядра. Чем дальше от нуля координата, тем больше переменная масштаба. Таким образом, камера будет отъезжать, когда у ядра будет более высокая y-координата. Вот то, что нам надо добавить в конце метода update() на экране нашего теста:

```
if(ball.position.y > 0) {
    camera.position.set(ball.position);
    camera.zoom = 1 + ball.position.y / WORLD_HEIGHT;
} else {
    camera.position.set(WORLD_WIDTH / 2, WORLD_HEIGHT / 2);
    camera.zoom = 1;
}
```

Когда y-координата ядра больше нуля, камера будет следовать за ним и отдаляться. Мы просто добавим некоторую величину к стандартной переменной масштаба, равной единице. Эта величина — отношение между положением ядра на y-оси и высотой мира. Если y-координата ядра находится в `WORLD_HEIGHT`, переменная масштаба будет равна 2, чтобы мы видели большую часть нашего мира. Способ, которым я это сделал, на самом деле произвольный — вы можете применить любую формулу, которую хотите, для данного случая. Если позиция ядра меньше или равна нулю, показываем мир в нормальном размере, как мы делали в предыдущих примерах.

Атлас текстур: не ленись — поделись

До этого момента мы использовали в наших программах только по одной текстуре. Что если мы захотим отобразить не только Боба, но и других супергероев, врагов, взрывы или монетки? У нас может быть несколько текстур, каждая из которых содержит изображение одного типа объектов. Но OpenGL вряд ли это понравится, поскольку нам надо будет переключать текстуры для каждого объекта, который мы отображаем (например, привязываем текстуру Боба, отображаем Боба, привязываем текстуру монетки, отображаем монетку и т. д.). Процесс может стать проще, если поместить различные изображения в одну текстуру. Это и будет *атлас текстур*: одна текстура, содержащая разные изображения. Мы должны сделать эту текстуру текущей (привязать) только один раз и тогда сможем отображать объекты любых типов, изображение которых есть в атласе. Так мы избавляемся от переизбытка смен состояния и ускоряем реализацию. Такой атлас текстур показан на рис. 8.23.

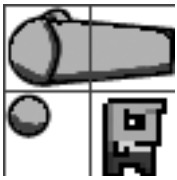

Рис. 8.23. Атлас текстур

На рис. 8.23 находятся три объекта: пушка, пушечное ядро и Боб. Сетка не является частью текстуры — она нужна только для того, чтобы показать, как я обычно создаю мои атласы текстур.

Атлас текстур имеет размер 64×64 пиксела, а каждая ячейка сетки имеет размеры 32×32 пиксела. Пушка занимает две ячейки, ядро немного меньше, чем четверть ячейки, а Боб помещается в одной ячейке. Теперь, если вы вспомните, как мы определяли границы (и графические прямоугольники) для пушки, ядра и мишеней, вы заметите, что отношение их размеров друг к другу очень напоминает то, которое мы имеем в этой сетке. Мишень в нашем мире имеет размер $0{,}5 \times 0{,}5$ м, пушка — $0{,}2 \times 0{,}2$ м. В нашем атласе текстур Боб имеет размеры 32×32 пиксела, а ядро немного меньше 16×16 пикселов. Соотношение между атласом текстур и размерами объектов в нашем мире должно быть понятно: 32 пиксела в атласе равны 0,5 м в мире.

В изначальном примере пушка имела размеры 1 × 1 метр, но мы, конечно, можем это изменить. В соответствии с нашим атласом текстур, где пушка имеет размеры примерно 64 × 32 пиксела, нам надо сделать пушку в нашем мире размерами 1 × 0,5 м. Совсем просто, правда?

Но почему же я выбрал именно 32 пиксела как эквивалент 1 м в нашем мире? Мы помним, что ширина и длина текстур должны быть степенями двух. Если количество пикселов равно 32, то есть степени 2, для соответствия 0,5 м в нашем мире позволяет художнику легко справиться с ограничениями размера текстур. Это также позволяет проще понять соотношение разных объектов из нашего мира относительно пиксельных изображений.

Вы можете приравнять и другое количество пикселов к единице мира. Например, 64 или 50 пикселов к 0,5 м в нашем мире. Какое же количество пикселов будет наилучшим? Это опять же зависит от разрешения экрана, на котором будет воспроизводиться игра. Проведем некоторые подсчеты.

Наш пушечный мир ограничен координатами (0; 0) в нижнем левом углу и (9,6; 4,8) в верхнем правом углу. Это отображено на экране. Выясним, сколько пикселов на единицу мира нам нужно для экрана Hero (480 × 320 пикселов в альбомном режиме):

```
pixelsPerUnitX = screenWidth / worldWidth = 480 / 9.6 = 50 pixels / meter
pixelsPerUnitY = screenHeight / worldHeight = 320 / 6.4 = 50 pixels / meter
```

Наша пушка, которая теперь будет занимать площадь 1 × 0,5 м в мире, будет иметь размеры 50 × 25 пикселов на экране. Мы используем область 64 × 32 пикселов нашей текстуры, так что немного снизим качество изображения пушки во время рендеринга. В зависимости от фильтра минификации, который мы применяем для текстуры, результат будет четким и мозаичным (`GL_NEAREST`) или немного размытым (`GL_LINEAR`). Если нам требуется идеальное отображение на Hero, надо немного масштабировать наши текстурные изображения. Мы можем установить размер ячейки сетки 25 × 25 пикселов вместо 32 × 32. Однако даже если мы просто изменим изображение атласа (или перерисуем его от руки), у нас будет изображение размером 50 × 50 пикселов, никак не подходящее для OpenGL ES. Нам надо будет добавить дополнительные участки слева и снизу, чтобы добиться изображения размером 64 × 64 пиксела (поскольку OpenGL ES требует, чтобы ширина и длина изображений были степенью 2). Можно сказать, что OpenGL ES прекрасно подходит для масштабирования нашего изображения текстуры для Hero.

Какова же ситуация с устройствами, имеющими более высокое разрешение, как Nexus One (800 × 480 в альбомном режиме)? Произведем подсчеты для такой конфигурации экрана при помощи следующих уравнений:

```
pixelsPerUnitX = screenWidth / worldWidth = 800 / 9.6 = 83 pixels / meter
pixelsPerUnitY = screenHeight / worldHeight = 480 / 6.4 = 75 pixels / meter
```

Количество пикселов на единицу мира на x- и y-осях будет разным, потому что отношение ширины и длины нашей области видимости (9,6 / 6,4 = 1,5) отличается от отношения стороны экрана (800 / 480 = 1,66). Мы уже говорили об этом в главе 4,

когда схематически описывали некоторые решения. Тогда мы упоминали об установленном размере в пикселах и отношении сторон. Теперь применим эту схему и выберем ширину и длину конуса отображения для нашего примера. В случае с Nexus One пушка, ядро и Боб будут немного увеличены и растянуты из-за высокого разрешения и иного отношения сторон. Мы примем этот факт, поскольку хотим, чтобы все игроки видели одну и ту же область нашего мира. Иначе игроки с более высоким показателем отношения сторон будут иметь возможность видеть большую часть нашего мира.

Итак, каким образом мы будем использовать атлас текстур? Мы просто перезададим наши прямоугольники. Вместо применения всей текстуры мы будем задействовать ее части. Чтобы рассчитать текстурные координаты углов изображений, содержащихся в атласе текстур, мы можем снова использовать уравнение из одного из наших последних примеров. Напомним:

```
u = x / imageWidth
v = y / imageHeight
```

Здесь u и v — текстурные координаты, а x и y — пиксельные координаты. Верхний левый угол Боба имеет пиксельные координаты (32; 32). Если мы вставим их в это уравнение, получим текстурные координаты (0,5; 0,5). Мы можем сделать то же самое для любых других углов, которые нам понадобятся, и, основываясь на этом, назначить правильные текстурные координаты для вершин наших прямоугольников.

Пример

Добавим этот атлас текстур к нашему предыдущему примеру и сделаем его красивее. Боб будет нашей мишенью.

Мы просто копируем `Camera2DTest` и немного его изменяем. Я поместил копию в файл под именем `TextureAtlasTest.java` и переименовал два класса, содержащихся в нем, соответственно `TextureAtlasTest` и `TextureAtlasScreen`.

Первое, что мы делаем, — добавляем новый член в класс `TextureAtlasScreen`:

```
Texture texture;
```

Вместо того чтобы создавать `Texture` в конструкторе, мы создаем текстуру в методе `resume()`. Помните, что текстуры будут потеряны, когда приложение вернется из состояния паузы, так что нам снова нужно будет воссоздавать их в методе `resume()`:

```
@Override
public void resume() {
    texture = new Texture(((GLGame)game), "atlas.png");
}
```

Я добавил изображение на рис. 8.23 в папку `assets` нашего проекта и назвал его `atlas.png`. (Конечно, оно не содержит сетки, изображенной на рисунке.)

Далее необходимо изменить определения вершин. У нас есть один экземпляр класса `Vertices` для каждого типа объектов (пушки, ядра и Боба), содержащий один

прямоугольник из четырех вершин и шести индексов. В результате получаются два треугольника. Нам нужно добавить текстурные координаты к каждой вершине в соответствии с атласом текстур. Мы также изменим представление пушки в качестве треугольника на представление как прямоугольник размером $1 \times 0{,}5$ м. Вот на что мы заменяем старый код создания вершин в конструкторе:

```
cannonVertices = new Vertices(glGraphics, 4, 6, false, true);
cannonVertices.setVertices(new float[] { -0.5f, -0.25f, 0.0f, 0.5f,
                                          0.5f, -0.25f, 1.0f, 0.5f,
                                          0.5f,  0.25f, 1.0f, 0.0f,
                                         -0.5f,  0.25f, 0.0f, 0.0f },
                                          0, 16);
cannonVertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);

ballVertices = new Vertices(glGraphics, 4, 6, false, true);
ballVertices.setVertices(new float[] { -0.1f, -0.1f, 0.0f,  0.75f,
                                        0.1f, -0.1f, 0.25f, 0.75f,
                                        0.1f,  0.1f, 0.25f, 0.5f,
                                       -0.1f,  0.1f, 0.0f,  0.5f },
                                        0, 16);
ballVertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);

targetVertices = new Vertices(glGraphics, 4, 6, false, true);
targetVertices.setVertices(new float[] { -0.25f, -0.25f, 0.5f, 1.0f,
                                          0.25f, -0.25f, 1.0f, 1.0f,
                                          0.25f,  0.25f, 1.0f, 0.5f,
                                         -0.25f,  0.25f, 0.5f, 0.5f },
                                          0, 16);
targetVertices.setIndices(new short[] {0, 1, 2, 2, 3, 0}, 0, 6);
```

Каждая из наших сетей теперь состоит из четырех вершин, все они, в свою очередь, имеют 2D-позицию и текстурные координаты. Добавляем в сеть 6 индексов, определяя два треугольника, которые хотим рендерить. Мы также немного уменьшаем пушку по оси y. Теперь она имеет размер $1 \times 0{,}5$ м вместо 1×1 м. Это также отражено ранее при создании объекта `Cannon` в конструкторе:

```
cannon = new Cannon(0, 0, 1, 0.5f);
```

Поскольку мы не производим проверку на столкновения с самой пушкой, нет особенной разницы, какой размер мы укажем в данном конструкторе. Мы делаем это скорее для единообразия.

Последнее, что нам надо изменить, — метод отображения:

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    camera.setViewportAndMatrices();

    gl.glEnable(GL10.GL_BLEND);
    gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
    gl.glEnable(GL10.GL_TEXTURE_2D);
```

```
    texture.bind();

    targetVertices.bind();
    int len = targets.size();
    for(int i = 0; i < len; i++) {
        GameObject target = targets.get(i);
        gl.glLoadIdentity();
        gl.glTranslatef(target.position.x, target.position.y, 0);
        targetVertices.draw(GL10.GL_TRIANGLES, 0, 6);
    }
    targetVertices.unbind();

    gl.glLoadIdentity();
    gl.glTranslatef(ball.position.x, ball.position.y, 0);
    ballVertices.bind();
    ballVertices.draw(GL10.GL_TRIANGLES, 0, 6);
    ballVertices.unbind();
    gl.glLoadIdentity();
    gl.glTranslatef(cannon.position.x, cannon.position.y, 0);
    gl.glRotatef(cannon.angle, 0, 0, 1);
    cannonVertices.bind();
    cannonVertices.draw(GL10.GL_TRIANGLES, 0, 6);
    cannonVertices.unbind();
}
```

Здесь мы включаем смешивание и устанавливаем нужную для этого функцию, а также включаем текстурирование и привязываем атлас текстур. Мы также немного изменяем вызов `cannonVertices.draw()`, который теперь отображает два треугольника вместо одного. Вот, пожалуй, все об этом. На рис. 8.24 показана наша косметическая операция.

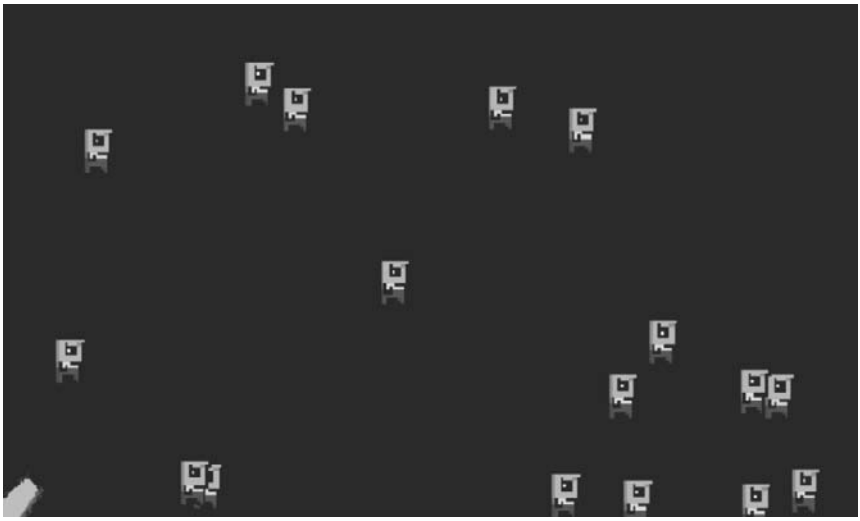

Рис. 8.24. Совершенствование примера с пушкой с помощью атласа текстур

Вот еще несколько вещей, которые стоит знать об атласах текстур.

- Когда мы используем GL_LINEAR как фильтр минификации и/или магнификации, могут возникнуть побочные явления, при которых два изображения внутри атласа касаются друг друга. Это происходит из-за того, что в процессе ассоциирования текстуры выбирается по четыре ближайших тексела из текстуры для одного пиксела на экране. Когда выбор выполняется на границе изображения, захватываются также текселы из соседнего изображения в атласе. Мы можем решить эту проблему, создав пустую границу в два пиксела между нашими изображениями. Или еще лучше, мы можем дублировать пограничный пиксел каждого изображения. Первое решение, разумеется, проще — надо просто убедиться, что ваша текстура остается степенью 2.
- Нет необходимости располагать все изображения в атласе в фиксированной сетке. Мы можем расположить изображения с произвольными размерами в атлас как можно ближе друг к другу. Нам нужно знать, где каждое изображение в атласе начинается и закачивается, чтобы мы могли подсчитать правильные текстурные координаты для него. Однако скомпоновать изображения с произвольным размером не так просто. В Интернете есть пара хороших программ, которые помогут вам создавать атласы текстур. Задайте запрос в поисковике, результатов будет море.
- Часто мы не можем сгруппировать все изображения игры в одну текстуру. Помните, что есть максимальный размер текстуры, который зависит от устройства. Мы точно можем сказать, что все устройства поддерживают текстуры размером 512×512 пикселов (или даже 1024×1024). Поэтому мы просто создаем несколько атласов текстур. Желательно группировать в один атлас объекты, которые будут на экране вместе, например все объекты первого уровня поместить в один атлас, объекты второго уровня — во второй, все элементы пользовательского интерфейса — в третий и т. д. Продумайте логическое размещение объектов, прежде чем окончательно дорабатывать ваши художественные ресурсы.
- Помните, как мы динамически рисовали цифры в игре с мистером Номом? Мы использовали для этого атлас текстур. Вообще, мы можем выполнять любое динамическое отображение текста при помощи атласа текстур. Просто добавьте все символы, которые нужны в игре, в атлас и отображайте их при необходимости при помощи нескольких треугольников, показывая нужные символы из атласа. В Интернете можно найти средства, которые помогут вам создать так называемый *растровый шрифт* (bitmap font). Однако для наших целей в следующих главах мы будем применять метод, который использовали в примере про мистера Нома: статический текст будет отображен заранее как единое целое, и только динамический текст (например, цифры при достижении рекорда) будет отображаться при помощи атласа.

Вы, вероятно, заметили, что Бобы исчезают немного раньше, чем по ним фактически попадает ядро. Это объясняется тем, что наши ограничивающие фигуры немного больше, чем нужно. Вокруг Боба и ядра по границам есть немного пусто-

го пространства. Что с этим делать? Мы просто сделаем ограничивающие фигуры чуть меньше. Это надо прочувствовать, поэтому поэкспериментируйте с ресурсами, пока столкновение не будет фиксироваться правильно. Во время разработки игр вы будете иметь дело со множеством таких тонких моментов. Умение тщательно отлаживать игру, наверное, один из наиболее важных навыков, кроме качественного проектирования различных уровней. Прочувствовать некоторые вещи трудно, но зато какое чувство удовлетворения вы испытаете, достигнув идеала. К сожалению, этому я не могу вас научить, потому что все зависит от того, как вы сами видите вашу игру. Считайте, что это та изюминка, которая отличает хорошие игры от плохих.

ПРИМЕЧАНИЕ

Чтобы справиться с преждевременными исчезновениями, которые мы только что упоминали, сделайте ограничивающие прямоугольники немного меньше, чем их графические представления, чтобы перед определением столкновения происходило пересечение объектов.

Фрагменты текстур, спрайты и пакеты: скрываем OpenGL ES

До настоящего времени наш код для примера с пушкой включал в себя шаблоны, без некоторых из них можно обойтись. В частности, это касается определения экземпляров класса `Vertices`. Утомительно постоянно писать семь строк кода только для того, чтобы определить один текстурированный прямоугольник. Еще одна область, в которой можно поработать, — ручное вычисление текстурных координат для изображений в атласах текстур. И наконец, очень много лишнего и повторяющегося кода употребляется для отображения наших 2D-прямоугольников. Я также упоминал об удобном способе отображения нескольких объектов вместо выполнения отрисовки для каждого объекта в отдельности. Мы можем решить все эти проблемы, для чего применим несколько новых концепций.

- *Фрагменты текстур*. Мы уже работали с фрагментами текстур в последнем примере. Фрагмент текстуры (texture region) — это прямоугольная область с одной текстурой (например, область в нашем атласе, содержащая пушку). Мы хотим, чтобы у нас был класс, инкапсулирующий все неудобные подсчеты преобразования пиксельных координат в текстурные.
- *Спрайты*. Спрайт очень похож на любой из наших игровых объектов. У него есть позиция (а также иногда ориентация и масштаб) и графические рамки. Мы отображаем спрайт с помощью прямоугольника, так же как и Боба или пушку. По сути, графические представления Боба и других объектов могут и должны считаться спрайтами. Спрайт также ассоциируется с определенным фрагментом в текстуре. Вот здесь и начинается работа с фрагментами текстур. Довольно соблазнительно скомбинировать спрайты непосредственно с игрой, но мы держим их отдельно в соответствии с паттерном «Модель — представление — контроллер».

Это четкое разделение между графикой и кодом модели нужно для красоты конструкции.

- *Сортировщики (бэтчеры) спрайтов.* Сортировщик спрайтов отвечает за одновременное отображение нескольких спрайтов. Для этого бэтчер спрайтов должен знать позицию, размер и фрагмент текстуры для каждого спрайта. Именно при помощи бэтчера спрайтов мы волшебным образом избавимся от многочисленных вызовов отрисовки и матричных операций для одного объекта.

  Эти концепции тесно взаимосвязаны; переходим к их обсуждению.

Класс TextureRegion

Поскольку мы уже работали с фрагментами текстур, будет несложно выяснить, что нам нужно. Мы знаем, как преобразовывать пиксельные координаты в текстурные. Мы хотим создать класс, где мы будем определять пиксельные координаты изображения в атласе текстур и который будет хранить текстурные координаты для фрагмента атласа для дальнейших операций (например, когда мы будем отображать спрайт). В листинге 8.16 показан класс `TextureRegion`.

Листинг 8.16. `TextureRegion.java`: преобразование пиксельных координат в текстурные координаты

```
package com.badlogic.androidgames.framework.gl;

public class TextureRegion {
    public final float u1, v1;
    public final float u2, v2;
    public final Texture texture;

    public TextureRegion(Texture texture, float x, float y, float width,
                                                    float height) {
        this.u1 = x / texture.width;
        this.v1 = y / texture.height;
        this.u2 = this.u1 + width / texture.width;
        this.v2 = this.v1 + height / texture.height;
        this.texture = texture;
    }
}
```

Класс `TextureRegion` хранит текстурные координаты верхнего левого угла (`u1`; `v1`) и нижнего правого угла (`u2`; `v2`) фрагмента в текстурных координатах. Конструктор принимает `Texture` и верхний левый угол, а также ширину и длину фрагмента в пиксельных координатах. Чтобы построить фрагмент текстуры для `Cannon`, нам надо сделать следующее:

```
TextureRegion cannonRegion = new TextureRegion(texture, 0, 0, 64, 32);
```

Таким же образом мы можем построить фрагмент для Боба:

```
TextureRegion bobRegion = new TextureRegion(texture, 32, 32, 32, 32);
```

Ну и т. д. Мы можем использовать это в коде примера, который мы создали, и применить члены класса u1, v1, u2, v2 для определения текстурных координат вершин наших прямоугольников. Но мы не сделаем этого, пока не избавимся от всех громоздких определений. Вот для чего мы будем использовать бэтчеры спрайтов.

Класс SpriteBatcher

Как мы уже говорили, спрайт легко определяется его позицией, размером и фрагментом текстуры (а также опциональными параметрами: ориентацией и масштабом). Это просто графический прямоугольник в пространстве нашего мира. Для простоты будем придерживаться обозначений позиции как центра спрайта и прямоугольника, построенного вокруг этого центра. Теперь мы можем получить класс Sprite и применить его таким образом:

```
Sprite bobSprite = new Sprite(20, 20, 0.5f, 0.5f, bobRegion);
```

Код будет строить новый спрайт с центром в точке (20; 20) в мире с расстоянием 0,25 м от центра в каждую сторону, используя bobRegion TextureRegion. Но вместо этого мы можем сделать так:

```
spriteBatcher.drawSprite(bob.x, bob.y, BOB_WIDTH, BOB_HEIGHT, bobRegion);
```

Так уже намного лучше. Нам уже не нужно строить другой объект для представления графической стороны нашего объекта. Вместо этого мы изображаем копию Боба по требованию. Мы можем также использовать такой перегруженный метод:

```
spriteBatcher.drawSprite(cannon.x, cannon.y, CANNON_WIDTH, CANNON_HEIGHT,
cannon.angle, cannonRegion);
```

Он будет рисовать пушку, вращающуюся под углом. Как же нам реализовать бэтчер спрайтов? И где представители класса Vertices? Давайте подумаем, как могут работать бэтчеры.

Что вообще означает понятие «пакетная обработка», или бэтчинг? В мире графики под ним обычно понимают соединение нескольких вызовов отрисовки в один. Как уже говорилось выше, наш графический процессор только «за». Вот как работает сортировщик спрайтов:

- Бэтчер имеет буфер, который изначально пуст (или становится пустым, когда мы сигнализируем о необходимости очистки). В буфере будут содержаться вершины. В нашем случае это будет просто массив переменных типа float — чисел с плавающей точкой.
- Каждый раз, когда мы вызываем метод SpriteBatcher.drawSprite(), мы добавляем четыре вершины в буфер, исходя из положения, размера, ориентации и фрагмента текстуры, которые задаются в качестве аргументов. Это также означает, что мы должны будем вручную менять и переводить координаты вершин без помощи OpenGL ES. Однако в этом нет ничего страшного, потому что тут нам

на помощь придет код из нашего класса Vector2. Это ключ к устранению всех вызовов отрисовки.

- Когда мы определили все спрайты, которые хотим отобразить, даем указание нашему бэтчеру спрайтов загрузить вершины всех прямоугольников спрайтов в графический процессор за один раз, а затем вызываем сам метод рисования OpenGL ES для изображения всех прямоугольников. Для этого мы преобразуем содержимое массива с переменными float в экземпляр класса Vertices и используем его для отображения прямоугольников.

ПРИМЕЧАНИЕ

Мы можем использовать только спрайты, находящиеся в одной и той же текстуре. Однако это не большая проблема, поскольку мы будем применять атласы текстур.

Обычная схема использования бэтчера спрайтов выглядит так:

```
batcher.beginBatch(texture);
// вызываем batcher.drawSprite() по мере необходимости,
// ссылаясь на фрагменты в текстуре
batcher.endBatch();
```

Вызов SpriteBatcher.beginBatch() позволяет сообщить сортировщику две вещи: он должен очистить буфер и использовать текстуру, которую мы ему передали. Для удобства свяжем текстуру с этим методом.

Далее отображаем столько спрайтов, относящихся к фрагментам в данной текстуре, сколько нам нужно. Это заполнит буфер, поскольку будут добавляться четыре вершины для каждого спрайта.

Вызов SpriteBatcher.endBatch() сообщит сортировщику спрайтов, что мы уже закончили отображение группы спрайтов (пакета) и что он должен загрузить вершины в графический процессор собственно для отображения. Мы собираемся использовать индексированное отображение с экземпляром класса Vertices, поэтому нам потребуется определить индексы в дополнение к вершинам в float-массиве буфера. Однако, поскольку мы всегда отображаем прямоугольники, мы можем один раз заранее сгенерировать индексы в конструкторе SpriteBatcher. Для этого нам нужно знать, какое максимальное количество спрайтов сортировщик может нарисовать за один раз. Наложив строгие ограничения на количество спрайтов, которое можно отображать для одной группы, мы сможем избежать наращивания массивов в других буферах; мы можем просто один раз распределить эти массивы и буферы в конструкторе.

Базовый механизм достаточно прост. Метод SpriteBatcher.drawSprite() может показаться странноватым, но на самом деле он довольно прост (пока мы не включаем вращение и масштабирование). Нам необходимо рассчитать координаты вершин и текстурные координаты в соответствии с параметрами. В предыдущих примерах мы уже делали это вручную, например, когда определяли прямоугольники для пушки, ядра и Боба. Мы выполним примерно то же самое в методе SpriteBatcher.drawSprite(), но только автоматически, исходя из параметров в методе. В листинге 8.17 показан код SpriteBatcher.

Листинг 8.17. Фрагмент из SpriteBatcher.java без вращения и масштабирования

```
package com.badlogic.androidgames.framework.gl;

import javax.microedition.khronos.opengles.GL10;

import android.util.FloatMath;

import com.badlogic.androidgames.framework.impl.GLGraphics;
import com.badlogic.androidgames.framework.math.Vector2;

public class SpriteBatcher {
    final float[] verticesBuffer;
    int bufferIndex;
    final Vertices vertices;
    int numSprites;
```

Сначала рассмотрим члены класса. Член класса `verticesBuffer` является временным `float`-массивом, в котором мы храним вершины спрайтов текущего пакета. Член `bufferIndex` указывает, с какого места `float`-массива мы должны начинать записывать следующие вершины. Член `vertices` — это экземпляр `Vertices`, который используется для отображения пакета. Он также хранит индексы, которые мы сейчас определим. Член `numSprites` содержит количество уже нарисованных спрайтов в текущем пакете.

```
public SpriteBatcher(GLGraphics glGraphics, int maxSprites) {
    this.verticesBuffer = new float[maxSprites*4*4];
    this.vertices = new Vertices(glGraphics, maxSprites*4, maxSprites*6,
                                 false, true);
    this.bufferIndex = 0;
    this.numSprites = 0;

    short[] indices = new short[maxSprites*6];
    int len = indices.length;
    short j = 0;
    for (int i = 0; i < len; i += 6, j += 4) {
            indices[i + 0] = (short)(j + 0);
            indices[i + 1] = (short)(j + 1);
            indices[i + 2] = (short)(j + 2);
            indices[i + 3] = (short)(j + 2);
            indices[i + 4] = (short)(j + 3);
            indices[i + 5] = (short)(j + 0);
    }
    vertices.setIndices(indices, 0, indices.length);
}
```

Переходя к конструктору, мы видим, что у нас есть две переменные: экземпляр класса `GLGraphics`, который нам нужен для создания экземпляра класса `Vertices`, и максимальное количество спрайтов, которые сортировщик может отображать в одном пакете. Первое, что мы делаем в конструкторе, — создаем массив переменных `float`. У нас есть четыре вершины для каждого спрайта, каждая вершина занимает

4 float (2 для x- и y-координаты и 2 для текстурных координат). Максимум у нас может быть количество спрайтов, равное maxSprites, поэтому для буфера нам понадобится $4 \cdot 4 \cdot$ maxSprites float. Далее создаем экземпляр класса Vertices. Он нужен нам для хранения не более maxSprites $\cdot$ 4 вершин и maxSprites $\cdot$ 6 индексов. Мы также сообщаем экземпляру класса Vertices, что у нас есть не только атрибуты положения, но и текстурные координаты для каждой вершины. Далее присваиваем членам bufferIndex и numSprites значение 0. Потом создаем индексы для нашего экземпляра класса Vertices. Нам нужно будет сделать это только один раз, потому что индексы не будут меняться. Первый спрайт в группе всегда будет иметь индексы 0, 1, 2, 2, 3, 0; следующий — 4, 5, 6, 6, 7, 4 и т. д. Мы можем предварительно вычислить их и сохранить в экземпляре класса Vertices. Таким образом мы сможем установить их только один раз, вместо того, чтобы делать это для каждого спрайта.

```
public void beginBatch(Texture texture) {
    texture.bind();
    numSprites = 0;
    bufferIndex = 0;
}
```

Далее следует метод beginBatch(). Он привязывает текстуру и возвращает в исходное состояние члены numSprites и bufferIndex, так что вершины первого спрайта будут вставлены в начало массива float verticesBuffer .

```
public void endBatch() {
    vertices.setVertices(verticesBuffer, 0, bufferIndex);
    vertices.bind();
    vertices.draw(GL10.GL_TRIANGLES, 0, numSprites * 6);
    vertices.unbind();
}
```

Следующий метод — endBatch(). Мы вызываем его, чтобы финализировать и отрисовать текущий пакет. Метод сначала переносит определенные для этого пакета вершины из массива float в экземпляр класса Vertices. Осталось привязать экземпляр класса Vertices, нарисовать numSprites $\cdot$ 2 треугольника и снова отвязать экземпляр класса Vertices. Поскольку мы используем индексированное отображение, мы определяем количество применяемых индексов, которое равно шести индексам для спрайта, умноженным на numSprites. Итак, остается само отображение.

```
public void drawSprite(float x, float y, float width, float height,
                       TextureRegion region) {
    float halfWidth = width / 2;
    float halfHeight = height / 2;
    float x1 = x - halfWidth;
    float y1 = y - halfHeight;
    float x2 = x + halfWidth;
```

```
        float y2 = y + halfHeight;

        verticesBuffer[bufferIndex++] = x1;
        verticesBuffer[bufferIndex++] = y1;
        verticesBuffer[bufferIndex++] = region.u1;
        verticesBuffer[bufferIndex++] = region.v2;

        verticesBuffer[bufferIndex++] = x2;
        verticesBuffer[bufferIndex++] = y1;
        verticesBuffer[bufferIndex++] = region.u2;
        verticesBuffer[bufferIndex++] = region.v2;

        verticesBuffer[bufferIndex++] = x2;
        verticesBuffer[bufferIndex++] = y2;
        verticesBuffer[bufferIndex++] = region.u2;
        verticesBuffer[bufferIndex++] = region.v1;

        verticesBuffer[bufferIndex++] = x1;
        verticesBuffer[bufferIndex++] = y2;
        verticesBuffer[bufferIndex++] = region.u1;
        verticesBuffer[bufferIndex++] = region.v1;

        numSprites++;
    }
```

Следующий метод можно назвать рабочей лошадкой класса `SpriteBatcher`. Он принимает *x*- и *y*-координаты центра спрайта, его ширину и длину и `TextureRegion`, к которому он относится. Метод добавляет четыре вершины в массив переменных `float`, начиная с текущего `bufferIndex`. Эти четыре вершины формируют текстурный прямоугольник. Мы подсчитываем позицию нижнего левого (`x1`; `y1`) и верхнего правого углов (`x2`; `y2`) и используем эти четыре переменные вместе с текстурными координатами из `TextureRegion` для построения вершин. Вершины добавляются против часовой стрелки, начиная с нижней левой вершины. Когда они добавлены в массив `float`, увеличиваем счетчик `mSprites` и ждем добавления следующего спрайта или завершения пакета.

Это, пожалуй, все, что нужно сделать. Мы избегаем вызовов множества методов отрисовки, просто помещая предварительно преобразованные вершины в массив `float` и отображая их за один раз. Так значительно ускорится реализация нашего отображения двухмерных спрайтов, по сравнению с методом, который мы использовали раньше. Меньшее количество изменений состояния OpenGL ES и меньше вызовов отрисовки упростят работу нашему графическому процессору.

Остается реализовать еще метод `SpriteBatcher.drawSprite()`, который может рисовать вращающийся спрайт. Нужно построить четыре угловые вершины без добавления позиции, повернуть их вокруг начала координат, добавить позицию спрайта, чтобы вершины были расположены в пространстве игрового мира, а затем продолжить так, как в предыдущем методе отрисовки. Мы можем использовать для

этого Vector2.rotate(), но тогда возникнут некоторые функциональные издержки. Мы, таким образом, воспроизводим код из Vector2.rotate() и оптимизируем его, где это возможно. Последний метод SpriteBatcher показан в листинге 8.18.

Листинг 8.18. Окончание метода SpriteBatcher.java: рисование вращающегося спрайта

```
    public void drawSprite(float x, float y, float width, float height,
                           float angle, TextureRegion region) {
        float halfWidth = width / 2;
        float halfHeight = height / 2;

        float rad = angle * Vector2.TO_RADIANS;
        float cos = FloatMath.cos(rad);
        float sin = FloatMath.sin(rad);

        float x1 = -halfWidth * cos – (-halfHeight) * sin;
        float y1 = -halfWidth * sin + (-halfHeight) * cos;
        float x2 = halfWidth * cos – (-halfHeight) * sin;
        float y2 = halfWidth * sin + (-halfHeight) * cos;
        float x3 = halfWidth * cos – halfHeight * sin;
        float y3 = halfWidth * sin + halfHeight * cos;
        float x4 = -halfWidth * cos – halfHeight * sin;
        float y4 = -halfWidth * sin + halfHeight * cos;

        x1 += x;
        y1 += y;
        x2 += x;
        y2 += y;
        x3 += x;
        y3 += y;
        x4 += x;
        y4 += y;

        verticesBuffer[bufferIndex++] = x1;
        verticesBuffer[bufferIndex++] = y1;
        verticesBuffer[bufferIndex++] = region.u1;
        verticesBuffer[bufferIndex++] = region.v2;

        verticesBuffer[bufferIndex++] = x2;
        verticesBuffer[bufferIndex++] = y2;
        verticesBuffer[bufferIndex++] = region.u2;
        verticesBuffer[bufferIndex++] = region.v2;

        verticesBuffer[bufferIndex++] = x3;
        verticesBuffer[bufferIndex++] = y3;
        verticesBuffer[bufferIndex++] = region.u2;
        verticesBuffer[bufferIndex++] = region.v1;

        verticesBuffer[bufferIndex++] = x4;
        verticesBuffer[bufferIndex++] = y4;
        verticesBuffer[bufferIndex++] = region.u1;
```

```
            verticesBuffer[bufferIndex++] = region.v1;

            numSprites++;
        }
    }
```

Мы делаем то же самое, что и в более простом методе отрисовки. Кроме того, строим все четыре угловые вершины вместо только двух противоположных. Это требуется для вращения. Все остальное такое же, как и раньше.

А что насчет масштабирования? Нам явно не нужен отдельный метод, поскольку для масштабирования спрайта нужно отмасштабировать только его ширину и длину. Мы можем сделать это внутри двух методов отрисовки, поэтому нет нужды создавать еще несколько методов для этого.

Итак, это и есть тот самый большой секрет о том, как происходит молниеносное отображение спрайтов при помощи OpenGL ES.

Используем класс SpriteBatcher

Добавим классы `TextureRegion` и `SpriteBatcher` в наш пример с пушкой. Я скопировал пример `TextureAtlas` и переименовал его в `SpriteBatcherTest`. Классы, содержащиеся в нем, теперь называются `SpriteBatcherTest` и `SpriteBatcherScreen`.

Первое, что я сделал, — избавился от членов класса `Vertices` в классе экрана. Нам они больше не нужны, ведь для исполнения всей черной работы у нас теперь есть `SpriteBatcher`. Вместо них я добавил следующие члены:

```
TextureRegion cannonRegion;
TextureRegion ballRegion;
TextureRegion bobRegion;
SpriteBatcher batcher;
```

Теперь у нас есть `TextureRegion` и `SpriteBatcher` для каждого из трех объектов в нашем атласе. Далее я видоизменил конструктор экрана. Избавился от кода реализации и инициализации класса `Vertices` и заменил его одной строкой кода:

```
batcher = new SpriteBatcher(glGraphics, 100);
```

Она поместит член `batcher` в новый экземпляр класса `SpriteBatcher`, который может рендерить 100 спрайтов в одном пакете.

Области `TextureRegion` будут инициализированы в методе `resume()`, так как они зависят от `Texture`:

```
@Override
public void resume() {
    texture = new Texture(((GLGame)game), "atlas.png");
    cannonRegion = new TextureRegion(texture, 0, 0, 64, 32);
    ballRegion = new TextureRegion(texture, 0, 32, 16, 16);
    bobRegion = new TextureRegion(texture, 32, 32, 32, 32);
}
```

Здесь нет ничего нового. Нам также нужно изменить метод present(). Вы удивитесь, насколько чистым и красивым он вам теперь покажется:

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    camera.setViewportAndMatrices();

    gl.glEnable(GL10.GL_BLEND);
    gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
    gl.glEnable(GL10.GL_TEXTURE_2D);

    batcher.beginBatch(texture);

    int len = targets.size();
    for(int i = 0; i < len; i++) {
        GameObject target = targets.get(i);
        batcher.drawSprite(target.position.x, target.position.y, 0.5f, 0.5f,
                           bobRegion);
    }

    batcher.drawSprite(ball.position.x, ball.position.y, 0.2f, 0.2f,
                       ballRegion);
    batcher.drawSprite(cannon.position.x, cannon.position.y, 1, 0.5f,
                       cannon.angle, cannonRegion);
    batcher.endBatch();
}
```

Совсем просто, правда? Единственные вызовы OpenGL ES, которые мы используем, предназначены для очистки экрана, активации смешивания и текстурирования и для задания функции смешивания. Все остальное — только SpriteBatcher и Camera2D. Поскольку все наши объекты находятся в одном атласе текстур, мы будем отображать их в одном пакете. Мы вызываем batcher.beginBatch() с атласом текстур, отображаем всех Бобов-мишеней, используя просто метод отрисовки, отображаем ядро (снова простым методом отрисовки) и наконец пушку, используя метод отрисовки, который может вращать спрайт. Мы заканчиваем метод вызовом batcher.endBatch(), который, собственно, переносит геометрию наших спрайтов в графический процессор для отображения.

Измерим производительность

Насколько же быстрее метод SpriteBatcher по сравнению с методом, который мы использовали в BobTest? Я добавил в код FPSCounter и засек время на Hero, Droid и Nexus One, как мы делали в случае с BobTest. Кроме того, увеличил количество мишеней до 100 и задал равным 102 максимальное количество спрайтов, которые SpriteBatcher может обрабатывать за раз. Ведь мы отображаем 100 мишеней, 1 ядро и 1 пушку. Вот результаты:

```
Hero (1.5):
12-27 23:51:09.400: DEBUG/FPSCounter(2169): fps: 31
12-27 23:51:10.440: DEBUG/FPSCounter(2169): fps: 31
12-27 23:51:11.470: DEBUG/FPSCounter(2169): fps: 32
12-27 23:51:12.500: DEBUG/FPSCounter(2169): fps: 32

Droid (2.1.1):
12-27 23:50:23.416: DEBUG/FPSCounter(8145): fps: 56
12-27 23:50:24.448: DEBUG/FPSCounter(8145): fps: 56
12-27 23:50:25.456: DEBUG/FPSCounter(8145): fps: 56
12-27 23:50:26.456: DEBUG/FPSCounter(8145): fps: 55

Nexus One (2.2.1):
12-27 23:46:57.162: DEBUG/FPSCounter(754): fps: 61
12-27 23:46:58.171: DEBUG/FPSCounter(754): fps: 61
12-27 23:46:59.181: DEBUG/FPSCounter(754): fps: 61
12-27 23:47:00.181: DEBUG/FPSCounter(754): fps: 60
```

Прежде чем переходить к выводам, протестируем и старый метод. Поскольку наш пример не эквивалентен старому `BobTest`, я также изменил `TextureAtlasTest`, который почти не отличается от нашего нынешнего примера — единственная разница в том, что он использует старый метод `BobTest` для отображения. Вот результаты:

```
Hero (1.5):
12-27 23:53:45.950: DEBUG/FPSCounter(2303): fps: 46
12-27 23:53:46.720: DEBUG/dalvikvm(2303): GC freed 21811 objects / 524280
bytes in 135ms
12-27 23:53:46.970: DEBUG/FPSCounter(2303): fps: 40
12-27 23:53:47.980: DEBUG/FPSCounter(2303): fps: 46
12-27 23:53:48.990: DEBUG/FPSCounter(2303): fps: 46

Droid (2.1.1):
12-28 00:03:13.004: DEBUG/FPSCounter(8277): fps: 52
12-28 00:03:14.004: DEBUG/FPSCounter(8277): fps: 52
12-28 00:03:15.027: DEBUG/FPSCounter(8277): fps: 53
12-28 00:03:16.027: DEBUG/FPSCounter(8277): fps: 53

Nexus One (2.2.1):
12-27 23:56:09.591: DEBUG/FPSCounter(873): fps: 61
12-27 23:56:10.591: DEBUG/FPSCounter(873): fps: 60
12-27 23:56:11.601: DEBUG/FPSCounter(873): fps: 61
12-27 23:56:12.601: DEBUG/FPSCounter(873): fps: 60
```

Hero работает с нашим новым методом `SpriteBatcher` гораздо хуже по сравнению со старым вариантом, использовавшим `glTranslate()` и другие похожие методы. Droid лучше работает с новым методом, а Nexus One особо не различает, что мы применяем. Если бы мы увеличили количество мишеней еще на 100, вы бы увидели, что новый метод `SpriteBatcher` также был бы быстрее и на Nexus One.

Так что же с Hero? Проблема BobTest была в слишком большом количестве вызовов OpenGL ES, так почему же новый метод работает хуже, хотя вызовов OpenGL ES в нем меньше?

Обходной маневр при помощи FloatBuffer

Причина этого совсем не очевидна. Наш SpriteBatcher помещает массив переменных float в прямой ByteBuffer каждый кадр, когда мы вызываем Vertices.setVertices(). Метод сводится к вызову FloatBuffer.put(float[]), и именно он виновен в ухудшении производительности. В то время как настольная версия Java выполняет метод FloatBuffer, задействовав память большого объема, версия Harmony вызывает FloatBuffer.put(float) для каждого элемента массива. И это очень плохо, потому что этот метод является JNI-методом, который приводит к большим издержкам (как и методы OpenGL ES, которые также представляют собой JNI-методы).

Есть несколько решений проблемы. Например, IntBuffer.put(int[]) этой проблеме не подвержен. Мы можем заменить FloatBuffer в нашем классе Vertices на IntBuffer и изменить Vertices.setVertices() так, чтобы он сначала переносил переменные float из float-массива во временный int-массив и затем копировал содержимое этого int-массива в IntBuffer. Это решение предложил разработчик игр Райан МакНелли, котрый также сообщил о баге в багтрекер Android. При устранении этого бага производительность на Hero увеличится в пять раз и немного меньше на других устройствах с Android.

Я включил в класс Vertices это изменение. Для этого я изменил тип поля verticies на IntBuffer. Я также добавил новый член tmpBuffer, который является массивом чисел int. Массив tmpBuffer инициализируется в конструкторе Vertices следующим образом:

```
this.tmpBuffer = new int[maxVertices * vertexSize / 4];
```

Мы также можем увидеть IntBuffer в конструкторе вместо FloatBuffer из ByteBuffer:

```
vertices = buffer.asIntBuffer();
```

А метод Vertices.setVertices() выглядит вот так:

```
public void setVertices(float[] vertices, int offset, int length) {
    this.vertices.clear();
    int len = offset + length;
    for(int i=offset, j=0; i < len; i++, j++)
        tmpBuffer[j] = Float.floatToRawIntBits(vertices[i]);
    this.vertices.put(tmpBuffer, 0, length);
    this.vertices.flip();
}
```

Таким образом, все, что мы делаем, — переносим содержимое параметра вершин в tmpBuffer. Статический метод Float.floatToRawIntBits() заново интерпретирует конфигурацию битов float как int. Нам нужно только скопировать содержимое массива int в IntBuffer, ранее известный нам как FloatBuffer. Увеличит ли это про-

изводительность? Теперь при использовании SpriteBatcherTest на Hero, Droid и Nexus One мы получаем такие результаты:

```
Hero (1.5):
12-28 00:24:54.770: DEBUG/FPSCounter(2538): fps: 61
12-28 00:24:54.770: DEBUG/FPSCounter(2538): fps: 61
12-28 00:24:55.790: DEBUG/FPSCounter(2538): fps: 62
12-28 00:24:55.790: DEBUG/FPSCounter(2538): fps: 62

Droid (2.1.1):
12-28 00:35:48.242: DEBUG/FPSCounter(1681): fps: 61
12-28 00:35:49.258: DEBUG/FPSCounter(1681): fps: 62
12-28 00:35:50.258: DEBUG/FPSCounter(1681): fps: 60
12-28 00:35:51.266: DEBUG/FPSCounter(1681): fps: 59

Nexus One (2.2.1):
12-28 00:27:39.642: DEBUG/FPSCounter(1006): fps: 61
12-28 00:27:40.652: DEBUG/FPSCounter(1006): fps: 61
12-28 00:27:41.662: DEBUG/FPSCounter(1006): fps: 61
12-28 00:27:42.662: DEBUG/FPSCounter(1006): fps: 61
```

Да, **я перепроверил, это не опечатка. Кадровая частота на Hero теперь действительно** 60 кадров в секунду. Обходной маневр размером всего в пять строк кода увеличивает быстродействие на 50 %. Быстродействие Droid тоже немного возрастает.

ПРИМЕЧАНИЕ

Есть еще и другой, более быстрый способ обойти проблему. Он касается JNI-метода, который производит изменения в памяти собственного кода. Вы можете найти его в Интернете в Android Game Development Wiki. Я по большей части использую этот обходной путь вместо чистого Java-способа. Однако работа с JNI-методами немного более сложная, поэтому я описал здесь способ решения с помощью Java.

Спрайт-анимация

Если вы когда-нибудь играли в 2D-видеоигру, то заметили, что мы до сих пор не занимались одним очень важным компонентом — спрайт-анимацией. Анимация состоит из так называемых ключевых кадров, которые создают иллюзию движения. На рис. 8.25 вы можете увидеть прекрасный анимированный спрайт, созданный Ари Фельдманном (взят из его безгонорарной библиотеки SpriteLib).

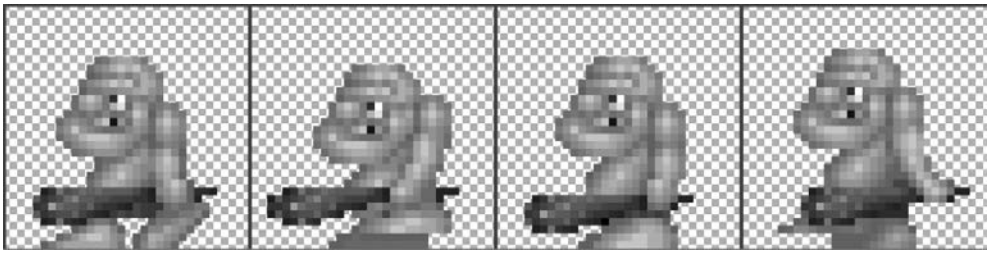

Рис. 8.25. Идущий пещерный человек, созданный Ари Фельдманном (в оригинале — без сетки)

Изображение имеет размеры 256 × 64 пиксела, каждый кадр — 64 × 64 пиксела. Для создания анимации мы просто рисуем спрайт, используя первый ключевой кадр, некоторый период времени — скажем, 0,25 секунды, затем переключаемся на следующий ключевой кадр и т. д. Когда мы достигаем последнего кадра, есть два варианта: мы можем остаться на последнем кадре или снова начать сначала (и реализовать так называемую циклическую анимацию).

Мы можем легко сделать это при помощи наших классов `TextureRegion` , а также `SpriteBatcher`. Обычно в одном атласе у нас будет не одна анимация, как на рис. 8.25, а намного больше. Кроме анимации ходьбы мы можем сделать анимацию прыжка, атаки и т. д. Для каждой анимации нам нужно знать длительность кадра, которая будет показывать, как долго мы используем в анимации один кадр перед переключением на другой.

Класс Animation

Исходя из этого, мы можем определить требования к классу `Animation`, который будет хранить данные об одной анимации, такой как анимация ходьбы на рис. 8.25.

- `Animation` будет содержать несколько `TextureRegion`, которые будут хранить информацию о позиции каждого кадра в атласе текстур. Порядок `TestureRegion` должен быть таким же, как при воспроизведении кадров.
- `Animation` также будет хранить длительность кадра, то есть время, которое должно пройти до переключения на следующий кадр.
- `Animation` должен содержать метод, куда мы будем передавать время, которое мы проводим в представленном `Animation` состоянии (например, шаг влево), и который будет возвращать нужный `TextureRegion`. Метод должен учитывать, хотим ли мы сделать анимацию циклической или остаться на последнем кадре при достижении конца.

Последнее требование особенно важно, потому что оно позволяет нам хранить только один экземпляр `Animation` для использования разными объектами нашего мира. Объект просто следит за своим текущим состоянием (например, стреляет ли объект, ходит или прыгает, а также как долго он находится в этом состоянии). Когда мы рендерим этот объект, мы используем его состояние для выбора анимации, которую мы хотим воспроизвести, и продолжительность состояния, чтобы выбрать нужный `TextureRegion` из `Animation`. В листинге 8.19 показан код нашего нового класса `Animation`.

Листинг 8.19. `Animation.java`, простой класс `Animation`

```
package com.badlogic.androidgames.framework.gl;

public class Animation {
    public static final int ANIMATION_LOOPING = 0;
    public static final int ANIMATION_NONLOOPING = 1;

    final TextureRegion[] keyFrames;
    final float frameDuration;

    public Animation(float frameDuration, TextureRegion ... keyFrames) {
```

```
        this.frameDuration = frameDuration;
        this.keyFrames = keyFrames;
    }

    public TextureRegion getKeyFrame(float stateTime, int mode) {
        int frameNumber = (int)(stateTime / frameDuration);

        if(mode == ANIMATION_NONLOOPING) {
            frameNumber = Math.min(keyFrames.length-1, frameNumber);
        } else {
            frameNumber = frameNumber % keyFrames.length;
        }
        return keyFrames[frameNumber];
    }
}
```

Сначала определяем две константы, которые будем использовать в методе `getKeyFrame()`. Первая указывает, что анимация должна быть циклической, вторая — что анимация должна остановиться на последнем кадре.

Далее задаем два члена: массив, содержащий `TextureRegion`, и переменную `float`, которая хранит длительность кадра.

Передаем длительность кадра и `TextureRegion`, которые содержат ключевые кадры, в конструктор, который просто их сохраняет. Мы могли бы создать внутреннюю копию массива `keyFrames`, но для этого нужно будет выделить еще один объект, который явно сведет с ума программу по очистке памяти.

Самое интересное тут заключается в методе `getKeyFrame()`. Мы передаем ему время, которое объект провел в состоянии, представленном анимацией, а также режим `Animation.ANIMATION_LOOPING` или `Animation.NON_LOOPING`. Сначала подсчитываем на основе `stateTime`, сколько кадров уже было отображено к данному моменту. Если анимация нециклическая, просто закрепляем `frameNumber` за последним элементом массива `TextureRegion`. В другом случае берем модуль, который автоматически создает эффект цикличности, который нам нужен (например, 4 % 3 = 1). Осталось только вернуть соответствующий `TextureRegion`.

Пример

Создадим пример под названием `AnimationTest` с соответствующим экраном `AnimationScreen`. Как и обычно, обсудим только экран.

Мы хотим отрендерить нескольких пещерных людей, идущих влево. Мир будет таких же размеров, как и конус отображения, то есть $4{,}8 \times 3{,}2$ м (этот размер произвольный, на самом деле мы можем использовать любой размер). Пещерный человек является `DynamicGameObject` размером 1×1 м. Выведем из `DynamicGameObject` новый класс `Caveman`, в котором будет храниться дополнительный член класса, следящий за тем, как долго пещерный человек уже находится в движении. Каждый пещерный человек будет двигаться со скоростью 0,5 м/с вправо или влево. Мы также добавим в класс `Caveman` метод `update()` для обновления позиции пещерного человека в соответствии с дельтой времени и его скоростью. Если пещерный человек достигает правого или левого края нашего мира, устанавливаем его с другой

стороны мира. Мы используем изображение с рис. 8.25 и создадим TextureRegion и экземпляр Animation соответственно. Для рендеринга применяем экземпляр класса Camera2D и SpriteBatcher. В листинге 8.20 показан код класса Caveman.

Листинг 8.20. Фрагмент из AnimationTest, внутренний класс Caveman

```
static final float WORLD_WIDTH = 4.8f;
static final float WORLD_HEIGHT = 3.2f;

static class Caveman extends DynamicGameObject {
    public float walkingTime = 0;

    public Caveman(float x, float y, float width, float height) {
        super(x, y, width, height);
        this.position.set((float)Math.random() * WORLD_WIDTH,
                          (float)Math.random() * WORLD_HEIGHT);
        this.velocity.set(Math.random() > 0.5f?-0.5f:0.5f, 0);
        this.walkingTime = (float)Math.random() * 10;
    }

    public void update(float deltaTime) {
        position.add(velocity.x * deltaTime, velocity.y * deltaTime);
        if(position.x < 0) position.x = WORLD_WIDTH;
        if(position.x > WORLD_WIDTH) position.x = 0;
        walkingTime += deltaTime;
    }
}
```

Две константы WORLD_WIDTH и WORLD_HEIGHT **являются частями внешнего класса** AnimationTest **и используются внутренним классом. Наш мир имеет размеры** 4,8 × 3,2 м.

Далее следует внутренний класс Caveman, который дополняет DynamicGameObject, поскольку мы будем перемещать пещерного человека с какой-то скоростью. Мы определяем дополнительный член класса для отслеживания того, как долго пещерный человек движется на данный момент. В конструкторе помещаем пещерного человека в произвольную позицию и позволяем идти направо или налево. Мы также присваиваем члену класса walkingTime номер от 1 до 10; таким образом наши пещерные люди не будут двигаться синхронно.

Метод update() перемещает пещерного человека в соответствии с его скоростью и дельтой времени. Если человек покидает мир, мы восстанавливаем его с правой или левой стороны. Мы также добавляем дельту времени к walkingTime, чтобы следить, как долго он уже движется. Листинг 8.21 показывает класс AnimationScreen.

Листинг 8.21. Фрагмент из Animation.java: класс AnimationScreen

```
class AnimationScreen extends Screen {
    static final int NUM_CAVEMEN = 10;
    GLGraphics glGraphics;
    Caveman[] cavemen;
    SpriteBatcher batcher;
    Camera2D camera;
    Texture texture;
    Animation walkAnim;
```

В классе экрана все члены будут нам уже привычны. Так, у нас есть экземпляр класса GLGraphics, массив Caveman, классы SpriteBatcher, Camera2D, класс Texture, содержащий ключевые кадры ходьбы, и экземпляр класса Animation.

```
public AnimationScreen(Game game) {
    super(game);
    glGraphics = ((GLGame)game).getGLGraphics();
    cavemen = new Caveman[NUM_CAVEMEN];
    for(int i = 0; i < NUM_CAVEMEN; i++) {
        cavemen[i] = new Caveman((float)Math.random(),
                                 (float)Math.random(), 1, 1);
    }
    batcher = new SpriteBatcher(glGraphics, NUM_CAVEMEN);
    camera = new Camera2D(glGraphics, WORLD_WIDTH, WORLD_HEIGHT);
}
```

В конструкторе создаем экземпляр класса Caveman, а также классов SpriteBatcher и Camera2D.

```
@Override
public void resume() {
    texture = new Texture(((GLGame)game), "walkanim.png");
    walkAnim = new Animation( 0.2f,
                              new TextureRegion(texture, 0, 0, 64, 64),
                              new TextureRegion(texture, 64, 0, 64, 64),
                              new TextureRegion(texture, 128, 0, 64, 64),
                              new TextureRegion(texture, 192, 0, 64, 64));
}
```

В методе resume() загружаем текстурный атлас, содержащий ключевые кадры анимации, из ресурсного файла под названием walkanim.png, который выглядит так же, как рис. 8.25. После этого создаем экземпляр класса Animation, установив длительность кадра в 0,2 секунды и передаем в него TextureRegion для каждого ключевого кадра в атласе текстур.

```
@Override
public void update(float deltaTime) {
    int len = cavemen.length;
    for(int i = 0; i < len; i++) {
        cavemen[i].update(deltaTime);
    }
}
```

Метод update() просто перебирает все экземпляры класса Caveman и вызывает их метод Caveman.update() с текущей дельтой времени. Это заставляет пещерных людей передвигаться и обновляет их время ходьбы.

```
    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        camera.setViewportAndMatrices();

        gl.glEnable(GL10.GL_BLEND);
```

```
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
        gl.glEnable(GL10.GL_TEXTURE_2D);

        batcher.beginBatch(texture);
        int len = cavemen.length;
        for(int i = 0; i < len; i++) {
            Caveman caveman = cavemen[i];
            TextureRegion keyFrame =
walkAnim.getKeyFrame(caveman.walkingTime, Animation.ANIMATION_LOOPING);
            batcher.drawSprite(caveman.position.x, caveman.position.y,
                               caveman.velocity.x < 0?1:-1, 1, keyFrame);
        }
        batcher.endBatch();
    }

    @Override
    public void pause() {
    }

    @Override
    public void dispose() {
    }
}
```

И наконец, у нас есть метод `present()`. Мы начинаем с чистки экрана и установления области просмотра и проекционной матрицы с помощью нашей камеры. Затем включаем смешивание и обработку текстур и устанавливаем функцию смешивания. Начинаем отображение, сообщая сортировщику спрайтов, что мы хотим завести новый пакет, используя анимационный атлас текстур. Затем проходим по всем пещерным человечкам и отображаем их. Для каждого пещерного человека находим сначала нужный ключевой кадр в экземпляре класса `Animation` в соответствии со временем ходьбы человека. Указываем, что анимация должна быть циклической. Затем рисуем пещерного человека с правильным текстурным фрагментом на его месте.

Но что делать с параметром ширины? Вы помните, что наша анимационная текстура содержит ключевые кадры для ходьбы влево. Мы можем отразить текстуру по горизонтали (на случай, если пещерный человек захочет пойти направо), просто указав отрицательную ширину. Если вы не верите мне, можете вернуться к коду `SpriteBatcher` и проверить, сработает ли этот способ. Мы, по существу, отражаем прямоугольник спрайта, указывая отрицательную ширину. Мы можем сделать то же самое по вертикали, задав отрицательную высоту.

Наши гуляющие пещерные люди изображены на рис. 8.26.

Вот и все, что нужно знать, чтобы создать хорошую 2D-игру с помощью OpenGL. Снова обратите внимание на то, что мы разделяем логическую составляющую игры и графическое представление. Пещерному человеку совсем необязательно знать, что его отображают. Поэтому он не содержит никаких членов, относящихся к рендерингу, как, например, экземпляр класса `Animation` или `Texture`. Все, что нам нужно делать, — следить за состоянием пещерного человека и тем, как долго он уже

находится в этом состоянии. Добавив к этому его позицию и размер, мы можем легко осуществить отображение с помощью наших маленьких вспомогательных классов.

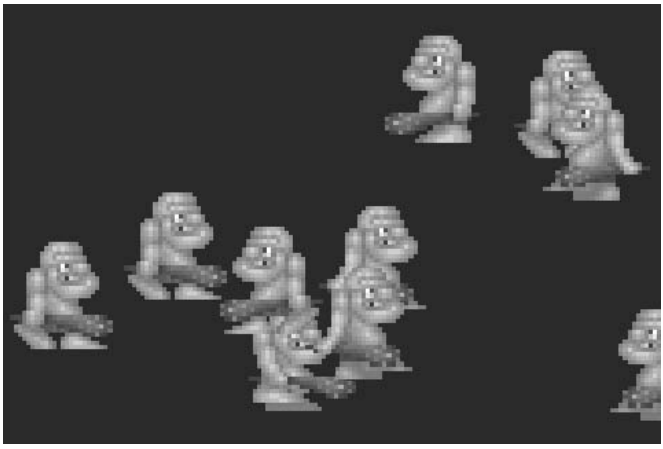

Рис. 8.26. Гуляющие пещерные люди

Подводя итог

Теперь вы достаточно подкованы, чтобы создать практически любую 2D-игру, какую хотите. Мы обсудили векторы и работу с ними, в итоге получив красивый класс `Vector2`, который можем использовать многократно. Мы также вспомнили основы физики для создания таких объектов, как баллистические пушечные ядра. Определение столкновений — также важнейшая часть большинства игр, и вы должны знать, как правильно и эффективно определять столкновения между объектами при помощи `SpatialHashGrid`. Мы рассмотрели, как отделять логическую составляющую игры и объекты от процесса отображения, создав классы `GameObject` и `DynamicGameObject`, которые отслеживают состояние и форму объектов. Мы узнали, как просто при помощи OpenGL ES и одного только метода `glOrthof()` можно реализовать идею 2D-камеры. Мы обсудили атласы текстур, выяснили, почему и зачем мы их применяем. Развив идею, мы поговорили о фрагментах текстуры, спрайтах и том, как можно эффективно отображать их при помощи `SpriteBatcher`. И наконец, изучили спрайт-анимацию, которая оказалась очень проста.

В следующей главе создадим новую игру, используя все новые средства, которые теперь у нас есть. Вы удивитесь, когда увидите, как просто будет это сделать!

9 «Большой прыгун»: двухмерная игра, написанная с помощью OpenGL ES

Пришло время использовать все, что вы узнали, для того, чтобы написать игру. В главе 3 вы узнали, что существует несколько очень популярных жанров игр для мобильных телефонов. Однако для нашей следующей игры я решил выбрать что-нибудь более простое. Мы реализуем игру в жанре «попрыгунчик» (jump-'em-up), подобную играм Abduction или Doodle Jump. Как и в случае с игрой «Мистер Ном», мы начнем с определения игровой механики.

Основная игровая механика

Я предлагаю вам установить игру Abduction на ваш телефон с ОС Android или просмотреть видеоролики об этой игре в Интернете. Из этого примера мы можем почерпнуть основную игровую механику нашей игры, которая будет называться «Большой прыгун». Вот некоторые детали.

- Персонаж постоянно прыгает вверх, передвигаясь с платформы на платформу. Игровой мир простирается по вертикали на множество экранов.
- Горизонтальным передвижением персонажа можно управлять, наклоняя телефон влево или вправо.
- Когда персонаж пересекает одну из боковых границ экрана, он появляется с противоположной стороны экрана.
- Платформы могут быть как неподвижные, так и перемещающиеся по горизонтали.
- Некоторые платформы могут рассыпаться (они выбираются случайным образом), когда персонаж наступает на них.

- На своем пути персонаж может собирать предметы для того, чтобы зарабатывать очки.
- Кроме монет персонажу могут встретиться пружины или платформы, позволяющие ему прыгать выше.
- Игровой мир также населяют и отрицательные персонажи, которые передвигаются по горизонтали. Если персонаж заденет одного из них, то он умрет и игра закончится.
- Игра также заканчивается, когда персонаж падает за нижнюю границу экрана.
- На вершине уровня находится какая-либо цель. Когда персонаж достигает ее, начинается новый уровень.

Хотя этот список и длиннее, чем тот, который мы создали для игры «Мистер Ном», он не кажется более сложным. На рис. 9.1 показан исходный макет игровых принципов. В этот раз для создания макета я использовал программу Paint.NET. Давайте теперь придумаем сюжет для игры.

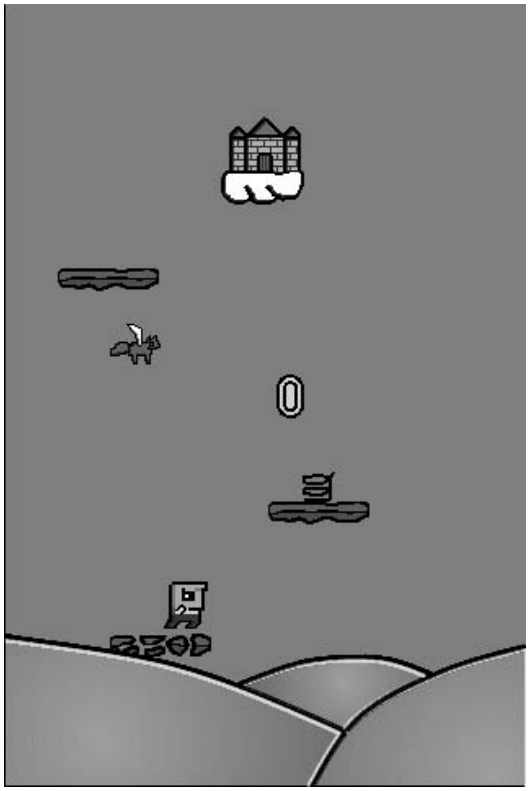

Рис. 9.1. Исходный макет, содержащий игровые механики;
на нем показаны персонаж, платформы, монеты, отрицательные персонажи и цель,
расположенная в верхней части уровня

Предыстория и стиль

Мы собираемся использовать всю нашу креативность и создать уникальный сюжет для игры.

Боб, наш игровой персонаж, страдает от хронического прыгунита. Он обречен прыгать каждый раз, когда касается земли. Что еще хуже, его возлюбленная принцесса, которая останется безымянной, была похищена армией злых летающий белок-убийц и помещена в небесный замок.

Недуг Боба в конечном итоге начинает приносить ему пользу, и он отправляется в путь за своей возлюбленной, сражаясь со злобной армией белок.

Такая история является классической для видеоигр, и для ее оформления отлично подойдет восьмибитная графика, образцы которой можно встретить в оригинальной игре «Супер Марио». Макет, изображенный на рис. 9.1, содержит итоговый вариант графики всех элементов игры. Боб, монеты, белки и рассыпающиеся платформы, конечно же, будут анимированными. Мы также используем музыку и звуковые эффекты, подходящие нашему графическому стилю.

Экраны и переходы

Теперь мы можем определить экраны игры и переходы между ними. Опишем их тем же способом, который использовался для экранов игры «Мистер Ном».

- У нас будет основной экран с логотипом; пункты меню PLAY (Играть), HIGHSCORES (Рекорды), и HELP (Помощь); а также кнопка, позволяющая включить или выключить звук.
- Кроме того, у нас будет основной экран игры, в котором пользователю будет задаваться вопрос о том, готов ли он играть, а также, соответственно, обрабатывать состояния выполнения игры, паузы, окончания игры и перехода на новый уровень. Отличие от игры «Мистер Ном» только одно — состояние перехода на новый уровень, в которое переключается игра в момент, когда Боб попадает в замок. В этом случае генерируется новый уровень, и Боб снова начинает свой путь с самого низа мира, при этом сохраняя набранные очки.
- В игре также будет экран, на котором будет отображаться таблица рекордов — пять лучших результатов, которых сумел достичь игрок.
- Наконец, в игре будет также экран помощи, на котором будет размещена информация об игровых механиках и цели игры. Кроме того, мы разместим там описание способа управления персонажем. Современные дети не сталкиваются со сложностями, с которыми сталкивались в 1980-х и ранних 1990-х годах мы, когда игры не подсказывали игрокам о том, как в них играть.

Этот список более-менее похож на тот, который мы создали для игры «Мистер Ном». На рис. 9.2 показаны все экраны игры, а также переходы между ними. Обратите внимание, на основном экране игры и его подэкранах нет никаких кнопок, кроме кнопки «Пауза». Пользователи будут интуитивно трогать экран, когда им зададут вопрос об их готовности начать игру.

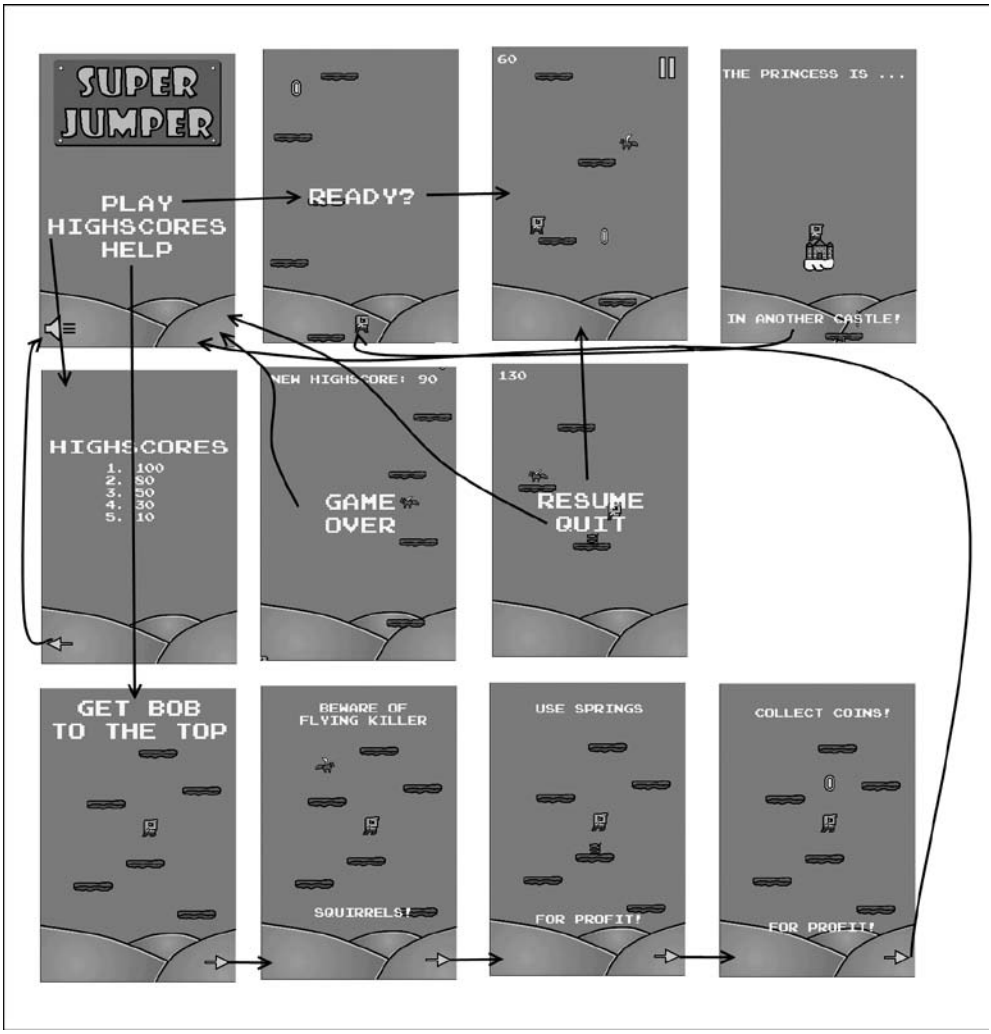

Рис. 9.2. Все экраны игры «Большой прыгун» и переходы между ними

Теперь, когда мы разобрались с этими важными аспектами, можно подумать о размерах игрового мира и его единицах измерения, а также о том, как наложить их на графические ресурсы.

Определение игрового мира

Мы снова столкнулись с классической проблемой «что было раньше, курица или яйцо?». Из предыдущей главы вы узнали, что существует соответствие между единицами измерения игрового мира (например, метрами) и пикселами. Объекты физически определены в пространстве мира. Ограничивающие их фигуры и позиции используют в качестве единицы измерения метры, скорости измеряются в метрах в секунду. Графическое представление объектов определяется в пикселах, поэтому нам придется установить некоторое соответствие между ними и метрами. Эту проблему возможно обойти, если сначала определить разрешение используемых графических ресурсов. Как и в случае с игрой «Мистер Ном», мы используем разрешение 320 × 480 пикселов (соотношение сторон равно 1,5). Далее необходимо установить соответствие между пикселями и метрами нашего мира. Макет, изображенный на рис. 9.1, позволяет получить представление о том, сколько места занимают различные объекты, а также их пропорции относительно друг друга. Обычно для двухмерных игр я использую следующий масштаб — 32 пиксела на один метр. Давайте теперь наложим на экран, чьи размеры составляют 320 × 480 пикселов, сетку, каждая клетка которой имеет размеры 32 × 32 пиксела. На рис. 9.3 показан макет с наложенной сеткой.

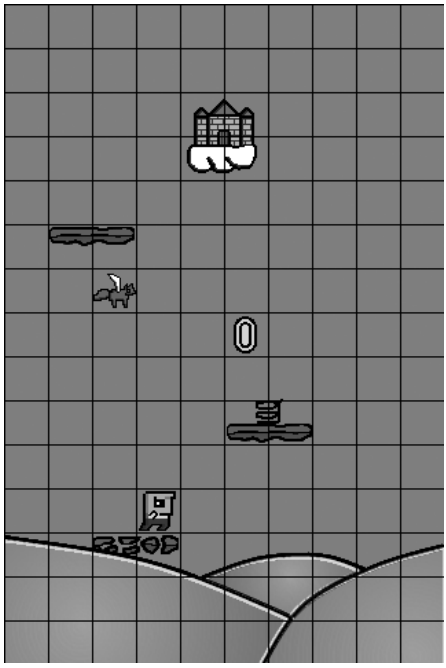

Рис. 9.3. Макет, на который наложена сетка; каждая клетка имеет размер 32 × 32 пиксела и соответствует площади игрового мира размером 1 × 1 м

Рисунок 9.3, конечно, является несколько ненатуральным. Я разместил объекты на экране таким образом, что они замечательно помещаются в клетки. В реальной игре мы расположим объекты по координатам, содержащим и дробные части.

Итак, что можно узнать из рис. 9.3? Прежде всего стало возможно напрямую определить размеры каждого объекта игрового мира в метрах. Следующие значения можно использовать для создания ограничивающих прямоугольников игровых объектов.

- Боб имеет размеры $0{,}8 \times 0{,}8$ м; он не занимает целую клетку.
- Платформа имеет размеры $2 \times 0{,}5$ м, занимая две клетки по горизонтали и половину клетки по вертикали.
- Монета имеет размеры $0{,}8 \times 0{,}5$ м. Она занимает практически целую клетку по вертикали и половину клетки по горизонтали.
- Пружина имеет размеры $0{,}5 \times 0{,}5$ метра, занимая половину клетки в каждом направлении. На самом деле высота пружины несколько больше ее ширины. Ее форма ограничена квадратом для того, чтобы было проще провести тестирование столкновений.
- Белка имеет размеры $1 \times 0{,}8$ м.
- Замок — $0{,}8 \times 0{,}8$ м.

Благодаря этим измерениям возможно получить размеры ограничивающих прямоугольников игровых объектов, что поможет определить столкновения. Мы можем подогнать их размеры, если они покажутся чересчур большими или маленькими в зависимости от того, как это отразится на игре.

Из рис. 9.3 можно также определить размеры окна просмотра. Игрок сможет увидеть область игрового мира площадью 10×15 м.

Единственное, что осталось определить, — это скорости и ускорения, которые будут использованы в игре. Это окажет значительное влияние на игровой процесс. Обычно следует провести несколько экспериментов для того, чтобы определить их верные значения. Рассмотрим те значения, к которым я пришел после нескольких этапов тестирования.

- Вектор ускорения, создаваемый гравитацией, равен $(0, -13)$ м/с^2, что несколько больше, чем гравитация на планете Земля, а также значения, использованного нами в примере с пушкой.
- Исходный вектор скорости прыжка Боба равен (0,11) м/с. Обратите внимание на то, что этот вектор влияет лишь на передвижение по вертикали. Горизонтальное передвижение будет определяться согласно данным, полученным от акселерометра.
- Вектор скорости прыжка Боба будет увеличиваться в 1,5 раза, когда он заденет пружину. Он будет равен (0; 16,5) м/с. Опять же, это значение получено лишь экспериментальным путем.

- Скорость передвижения Боба по горизонтали равна 20 м/с. Обратите внимание на то, что она не имеет направления, то есть не является вектором. Далее я объясню, как именно использовать данные, полученные от акселерометра.
- Белки будут патрулировать пространство слева направо и наоборот. У них будет постоянная скорость передвижения, равная 3 м/с. Если выразить ее вектором, то она будет равна (–3,0) м/с, когда белка двигается влево, и (3,0) м/с, когда белка двигается вправо.

Так как же Боб будет перемещаться по горизонтали? Скорость передвижения по горизонтали, которую мы определили ранее, на самом деле является максимальной скоростью передвижения по горизонтали. В зависимости от того, насколько игрок наклонит свой телефон, скорость перемещения Боба по горизонтали будет изменяться от 0 (нет наклона) до 20 м/с (максимальный наклон телефона в одну сторону).

Мы будем использовать наклон акселерометра по оси x, поскольку игра будет работать в портретной ориентации. В то время, когда телефон не наклонен, ось сообщит об ускорении, равном 0 м/с^2. Когда же телефон максимально наклонен влево, при этом оказавшись в альбомном режиме, ось сообщит об ускорении, равном приблизительно –10 м/с^2. В случае когда телефон максимально наклонен вправо, ось сообщит об ускорении, равном приблизительно 10 м/с^2. Нам нужно нормализовать показания акселерометра, разделив их на максимальное абсолютное значение (10), а затем умножив их на максимальную скорость перемещения Боба по горизонтали. Поэтому Боб будет передвигаться влево или вправо со скоростью 20 м/с, когда телефон максимально наклонен в одну из сторон. Скорость будет уменьшаться с уменьшением угла наклона телефона. Боб может дважды за секунду пересечь экран, когда телефон максимально наклонен.

Скорость перемещения по горизонтали будет обновляться каждый кадр с учетом показаний оси x акселерометра. Следует также объединять ее со скоростью перемещения Боба по вертикали, которую можно определить, используя гравитационное ускорение и его текущую вертикальную скорость, как мы делали это с пушечным ядром в предыдущих примерах.

Большое значение играет также то, какая часть мира видна в данный момент. Поскольку Боб будет погибать всякий раз, когда пересечет нижнюю часть экрана, камера будет иметь важную роль в игровой механике. Поскольку камера уже используется для отрисовки и перемещается вверх, когда Боб прыгает, мы не будем применять ее в наших классах эмуляции игрового мира. Вместо этого мы будем фиксировать самое большое значение по оси y, которого сумел достичь Боб. Если в какой-то момент он окажется ниже координаты, соответствующей разности этого значения и половины высоты окна просмотра, мы будем знать, что он покинул экран. Поскольку нам необходимо знать высоту окна просмотра, чтобы определять, погиб Боб или нет, у нас не будет четкого разделения между моделью (классами

эмуляции мира) и графической составляющей. Я бы сказал, что с этим можно смириться.

Теперь рассмотрим ресурсы, которые нам понадобятся для создания игры.

Создание ресурсов

Наша новая игра будет иметь два типа графических ресурсов: элементы пользовательского интерфейса и непосредственно игровые элементы (элементы игрового мира). Начнем с описания элементов пользовательского интерфейса.

Элементы пользовательского интерфейса

Первое, на что следует обратить внимание, — элементы пользовательского интерфейса (кнопки, логотипы и т. д.) не зависят от преобразования единиц измерения из пикселов в метры, которое было определено ранее. Как и в случае с игрой «Мистер Ном», следует разработать их так, чтобы они соответствовали конкретному разрешению — в нашем случае 320 × 480 пикселов. Взглянув на рис. 9.2 можно определить, какие именно элементы пользовательского интерфейса будет иметь игра.

В первую очередь следует создать кнопки, которые могут понадобиться для различных экранов. На рис. 9.4 показаны все кнопки, задействованные в игре.

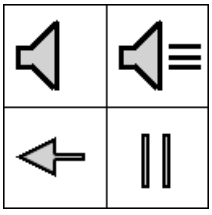

Рис. 9.4. Различные кнопки, каждая из которых имеет размер 64 × 64 пиксела

Я всегда создаю все графические ресурсы на сетке, ячейки которой имеют размер 32 × 32 или 64 × 64 пиксела. Кнопки, показанные на рис. 9.4, находятся на сетке с ячейками 64 × 64 пиксела. Кнопки, которые располагаются в верхнем ряду сетки, использованы на экране главного меню. Они сигнализируют о том, включен ли звук. Стрелка, расположенная в нижней левой ячейке сетки, используется на нескольких экранах для перехода на следующий экран. Кнопка, находящаяся в правой нижней ячейке сетки, применяется на главном экране игры и позволяет поставить игру на паузу.

Вы можете задаться вопросом, почему нет стрелки, показывающей вправо. Помните, что с помощью нашего замечательного класса `Spritebatcher` возможно легко переворачивать картинки, задавая им отрицательные значения ширины и/или высоты.

Мы используем этот трюк на нескольких графических ресурсах для того, чтобы сэкономить немного памяти.

Далее рассмотрим элементы, которые понадобятся на экране главного меню. На нем разместятся логотип, пункты меню и фоновая картинка. Все эти элементы показаны на рис. 9.5.

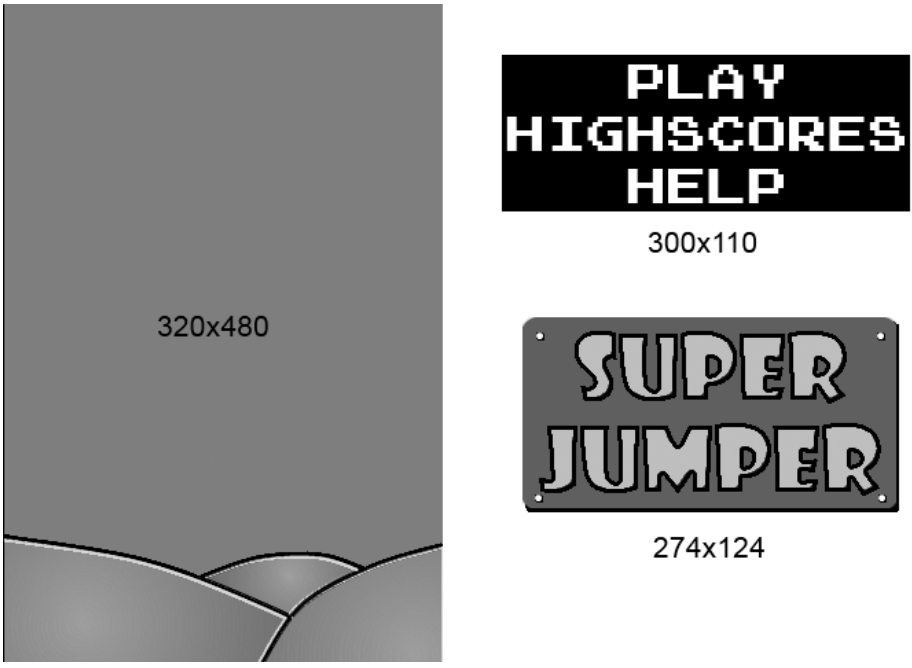

Рис. 9.5. Фоновое изображение, пункты главного меню и логотип

Фоновое изображение используется на всех экранах, а не только на экране главного меню. Его размеры в точности совпадают с размерами целевого разрешения 320 × 480 пикселов. Пункты главного меню имеют размер 300 × 110 пикселов. Для них был выбран черный фон, поскольку белое на белом выглядит совсем не так хорошо. В реальном изображении фон, конечно же, создан из прозрачных пикселов. Размеры логотипа составляют 274 × 142 пиксела, он имеет несколько прозрачных пикселов по углам.

Далее необходимо создать изображения для экранов помощи. Я поленился и вместо того, чтобы составлять их из нескольких элементов, нарисовал их как полноэкранные изображения размером 320 × 480. Это немного сократит размер кода, отрисовывающего их, и не добавит слишком много объема нашей программе. Вы можете увидеть все экраны помощи на рис. 9.2. Единственный элемент, который мы добавим на эти изображения, — кнопка со стрелкой.

Для экрана рекордов мы повторно используем часть изображения экрана главного меню, в которой есть пункт меню `HIGHSCORES` (Рекорды). Сами результаты

будут отображаться при помощи специального приема, который мы рассмотрим далее в этой главе. Остальная часть экрана будет составлена из фонового изображения и кнопки.

Экран игры имеет еще несколько текстовых элементов пользовательского интерфейса, например метку `READY?` (Готовы?), пункты меню, появляющегося, когда игра приостановлена (`RESUME` (Продолжить) и `QUIT` (Выйти)), а также метку `GAME OVER` (Игра окончена). На рис. 9.6 они показаны во всей красе.

Рис. 9.6. Метки READY? (Готовы?), RESUME (Продолжить), QUIT (Выйти) и GAME OVER (Игра окончена)

Обработка текста с помощью растровых шрифтов

Итак, как же следует отрисовывать прочие текстовые элементы игрового экрана? Мы будем использовать прием, задействованный при создании игры «Мистер Ном» для отрисовки результатов. В данном случае нам нужно применить набор не только цифр, но и символов. Используем атлас изображений, в котором каждое подизображение будет представлять символ (например, «0» или «a»). Такой атлас изображений называется *растровым шрифтом*. На рис. 9.7 показан растровый шрифт, который мы будем применять.

Рис. 9.7. Растровый шрифт

Черный фон и сетка, показанные на рис. 9.7, конечно, не являются частью изображений, входящих в состав растрового шрифта. Использование таких шрифтов — очень старый прием отрисовки текста на экране в игре. Обычно они состоят из изображений, созданных для набора символов ASCII. Один такой символ называется глифом. ASCII — это один из Юникода. Всего в наборе ASCII 128 символов (табл. 9.1).

Таблица 9.1. Символы ASCII и их десятичное, шестнадцатиричное и восьмиричное представления

Dex	Hex	Oct	Char	Dex	Hex	Oct	Char	Dex	Hex	Oct	Char	Dex	Hex	Oct	Char
0	0	0		32	20	40	(пробел)	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Из этих 128 символов печатаемыми являются лишь 96 (они имеют номера с 32-го по 126-й). Наш растровый шрифт состоит исключительно из печатаемых символов. Его первый ряд содержит с 32-го по 47-й символы, следующий ряд — с 48-го по 63-й и т. д. Использование ASCII полезно лишь в том случае, когда необходимо отобразить на экране текст, состоящий из символов стандартного латинского алфавита. Существует расширенный формат ASCII, который имеет символы с номерами с 128-го по 255-й. Эти символы используются для кодирования наиболее часто встречающихся символов западных языков, например ö или é. Более

объемные наборы символов (например, для китайского или арабского языков) представляются с помощью Юникода и не могут быть закодированы при помощи ASCII. Для нашей игры будет достаточно набора символов ASCII.

Как же отобразить текст с использованием растрового шрифта? Оказывается, это очень просто. Сначала следует создать 96 текстурных регионов, каждый из которых будет указывать на глиф растрового шрифта. Эти регионы возможно хранить в массиве, например, следующим образом:

```
TextureRegion[] glyphs = newTextureRegion[96];
```

Строки в языке Java кодируются при помощи 16-битного Юникода. К счастью для нас, символы ASCII, использованные в нашем растровом шрифте, имеют одинаковые номера как в ASCII, так и в Юникод. Чтобы получить регион для символа строки на языке Java, необходимо сделать лишь следующее:

```
int index = string.charAt(i) – 32;
```

Так мы получаем прямой индекс массива текстурных регионов. Мы просто отнимаем номер символа пробела (32) от порядкового номера текущего символа строки. Если индекс меньше нуля или больше 95, то можно определить, что это символ Юникод, не входящий в наш растровый шрифт. Обычно следует просто игнорировать такой символ.

Чтобы отрисовать несколько символов в строке, необходимо знать, как много пространства должно разделять эти символы. Растровый шрифт, показанный на рис. 9.7, является так называемым моноширинным шрифтом. Это значит, что каждый глиф имеет одинаковую ширину. Глифы нашего растрового шрифта имеют размер 16×20 пикселов каждый. Для перемещения позиции отрисовки от символа к символу нам следует всего лишь добавить к ней 20 пикселов. Количество пикселов, на которое перемещается позиция отрисовки от символа к символу, называется смещением. В нашем растровом шрифте оно фиксировано, но в общем случае оно является переменной величиной, которая изменяется в зависимости от отрисовываемого символа. Более сложная форма расчета смещения принимает во внимание как отрисовываемый в данный момент символ, так и следующий за ним. Такой прием называется *кернингом*, вы можете узнать о нем больше в Интернете. Мы будем использовать лишь моноширинные растровые шрифты, поскольку с ними наша задача значительно упрощается.

Итак, как я сгенерировал этот растровый шрифт ASCII? Я использовал один из многих инструментов для генерации атласов текстур, доступных в Сети. Тот, что применил я, называется Bitmap Font Generator. Он выпущен компанией Codehead и распространяется бесплатно. Вы можете выбрать файл шрифта, хранящийся на вашем компьютере, определить высоту шрифта, и генератор создаст изображение, содержащее символы набора ASCII. Этот инструмент имеет также и другие возможности, которые я не могу обсудить в рамках этой книги. Советую вам ознакомиться с ними самостоятельно.

Все остальные строки игры мы нарисуем именно таким способом. Далее вы увидите конкретную реализацию класса растрового шрифта. Рассмотрим и остальные ресурсы.

Теперь, когда мы создали растровый шрифт, у нас есть ресурсы для всех графических элементов пользовательского интерфейса игры. Мы будем отрисовывать их с помощью класса `SpriteBatcher`, используя камеру, которая задает конус отображения, точно соответствующий нашему целевому разрешению. Таким способом мы можем измерять все координаты в пикселах.

Элементы игры

Теперь рассмотрим непосредственно игровые элементы. Все они, как говорилось ранее, соответствуют нашей пиксельной единице измерения. Чтобы максимально упростить их создание, я использовал простой прием — начинал рисование каждого из них с сетки, чьи клетки были размером 32 × 32 пиксела. Все объекты располагались в центре одной или более клеток, поэтому они с легкостью могли соответствовать своим физическим размерам, которые они имеют в нашем мире. Начнем с Боба (рис. 9.8).

Рис. 9.8. Пять анимационных кадров, изображающих Боба

На рис. 9.8 изображены два кадра, на которых Боб подпрыгивает, два кадра, на которых он падает, и один кадр, где он мертв. Полный рисунок имеет размер 160 × 32 пиксела, а каждая анимация — 32 × 32 пиксела. Пикселы фонового изображения прозрачны.

Боб может находиться в трех состояниях: прыжок, падение и смерть. У нас есть анимационные кадры для каждого из этих состояний. Два кадра прыжка различаются лишь тем, что на одном у Боба торчит чуб. Мы создадим экземпляр класса `Animation` для каждой из этих трех анимаций Боба и будем использовать их для отрисовки персонажа соответственно его текущему состоянию. У нас не будет дублирующих кадров на тот случай, когда Боб будет двигаться влево. Как и в случае с кнопкой, изображающей стрелку, мы просто зададим отрицательную ширину кадра при вызове метода `SpriteBatcher.drawSprite()`, что перевернет изображение Боба по горизонтали.

На рис. 9.9 изображена злобная белка. В этот раз у нас два анимационных кадра — белка будет махать крыльями.

Рис. 9.9. Анимационные кадры, изображающие злобную летающую белку

Изображение на рис. 9.9 имеет размер 64 × 32 пиксела, а каждый кадр — 32 × 32 пиксела.

Анимация монеты, показанная на рис. 9.10, будет особенной. Вместо последовательности кадров 1, 2, 3, 1 мы используем 1, 2, 3, 2, 1. В противном случае монета из полностью повернутого состояния, изображенного на кадре 3, сразу перейдет в полностью развернутое состояние, показанное на кадре 1. Мы можем сохранить немного памяти, повторно использовав второй кадр.

Рис. 9.10. Анимационные кадры, изображающие монету

Изображение на рис. 9.10 имеет размер 96×32 пиксела, каждый его кадр — 32×32 пиксела.

О пружине, изображенной на рис. 9.11, можно сказать не так уж много. Она просто спокойно располагается в центре изображения.

Рис. 9.11. Пружина; изображение имеет размеры 32×32 пиксела

Замок, показанный на рис. 9.12, также не анимирован. По размерам он больше, чем все остальные объекты (64×64 пиксела).

Рис. 9.12. Замок

Платформа, приведенная на рис. 9.13 (64×64 пиксела), имеет четыре анимационных кадра. В соответствии с нашей игровой механикой некоторые платформы будут рассыпаться, когда Боб заденет их. В таком случае мы воспроизведем полную анимацию платформы. Для неподвижных платформ мы будем использовать только первый кадр.

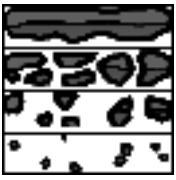

Рис. 9.13. Анимационные кадры, изображающие платформу

Атлас текстур спешит на помощь

Выше были перечислены все графические активы, которые будут присутствовать в нашей игре. Мы уже говорили о том, что текстурам необходимы определенные значения ширины и высоты. Фоновое изображение и все экраны помощи имеют размер 320 × 480 пикселов. Мы будем хранить их как изображения размером 512 × 512 пикселов, чтобы загружать их как текстуры. Всего получается 6 текстур.

Нужно ли создавать отдельные текстуры для каждого прочего изображения? Нет. Мы создадим единый атлас текстур. Все прочие элементы отлично помещаются в единый атлас размером 512 × 512 пикселов, который можно загрузить как единую текстуру — это должно действительно порадовать GPU, ведь в таком случае у нас будет гораздо меньше текстур. На рис. 9.14 показан наш атлас текстур.

Рис. 9.14. Мощный атлас текстур

Изображение на рис. 9.14 имеет размеры 512 × 512 пикселов. Сетка и границы не являются частью изображения, а фоновые пикселы прозрачны. Это также верно и для черных фоновых пикселов меток пользовательского интерфейса и растрового шрифта. Ячейки сетки имеют размер 32 × 32 пикселов каждая.

Я поместил все изображения атласа в точки, чьи координаты кратны 32. Это упростит создание текстурных регионов.

Музыка и звук

Нам также понадобятся звуковые эффекты и музыка. Поскольку у нашей игры 8-битный ретростиль, мы можем использовать так называемые чип-тюны. Чип-тюнами называются звуковые эффекты и музыка, сгенерированные синтезатором. Наиболее известные чип-тюны были сгенерированы приставками NES, SNES и GameBoy фирмы Nintendo. Для создания звуковых эффектов я использовал инструмент sfxr, созданный Томасом Петерссоном (есть также флэш-версия — as3sfxr). Вы можете найти его по адресу www.superflashbros.net/as3sfxr.

Я создал звуковые эффекты для прыжка, касания пружины, касания монеты и касания белки. Кроме того, я создал звуковой эффект для щелчка на элементах пользовательского интерфейса. Все, что я делал, — нажимал кнопки в левой части окна программы as3sfxr в каждой категории, пока не находил подходящий звуковой эффект.

С музыкой для игр обычно определиться немного сложнее. В Интернете существует всего несколько сайтов, предоставляющих 8-битные чип-тюны, подходящие для игр вроде «Большого прыгуна». Мы будем использовать одну песню, которая называется NewSong, ее написал Геир Тьелта (Geir Tjelta). Ее можно найти на сайте www.freemusicarchive.org. Она находится под лицензией Creative Commons Atribution-NonCommercial-NoDerivatives (также известной как Music Sharing). Это означает, что ее можно применять для некоммерческих проектов, например таких, как наш «Большой прыгун» с открытым исходным кодом, в том случае если мы укажем, что ее написал Геир, и не изменим оригинальный фрагмент. Когда вы будете искать музыку для игры в Интернете, всегда следите за тем, что вы твердо придерживаетесь этой лицензии. Люди вкладывают много сил в эти песни. Если лицензия вашему проекту не подходит (например, если он коммерческий), то вы не сможете использовать эти песни.

Реализация «Большого прыгуна»

Реализовать «Большого прыгуна» будет довольно легко. Мы можем повторно использовать весь фреймворк, который рассмотрели в предыдущей главе, а на высоком уровне будем следовать архитектуре, разработанной для игры «Мистер Ном». Это означает, что необходимо создать класс для каждого экрана и все эти классы будут реализовывать логику и представление, которые должен иметь данный экран. Кроме того, необходимо также создать стандартные действия перед началом работы над проектом — сделать подходящий файл манифеста, поместить все активы в каталог `assets/`, определить все необходимые значки приложения и т. д. Начнем с главного класса — `Assets`.

Класс Assets

В игре «Мистер Ном» у нас заранее был готов класс `Assets`, состоявший только из множества ссылок на изображения и звуки (`Pixmap` и `Sound`), которые хранились в статических переменных — членах класса. То же самое мы сделаем и для «Большого

прыгуна», но на этот раз мы добавим немного логики для их загрузки. В листинге 9.1 показан код класса.

Листинг 9.1. Класс Assets.java, в котором хранятся все активы, за исключением текстур экранов помощи

```
package com.badlogic.androidgames.jumper;

import com.badlogic.androidgames.framework.Music;
import com.badlogic.androidgames.framework.Sound;
import com.badlogic.androidgames.framework.gl.Animation;
import com.badlogic.androidgames.framework.gl.Font;
import com.badlogic.androidgames.framework.gl.Texture;
import com.badlogic.androidgames.framework.gl.TextureRegion;
import com.badlogic.androidgames.framework.impl.GLGame;

public class Assets {
    public static Texture background;
    public static TextureRegion backgroundRegion;

    public static Texture items;
    public static TextureRegion mainMenu;
    public static TextureRegion pauseMenu;
    public static TextureRegion ready;
    public static TextureRegion gameOver;
    public static TextureRegion highScoresRegion;
    public static TextureRegion logo;
    public static TextureRegion soundOn;
    public static TextureRegion soundOff;
    public static TextureRegion arrow;
    public static TextureRegion pause;
    public static TextureRegion spring;
    public static TextureRegion castle;
    public static Animation coinAnim;
    public static Animation bobJump;
    public static Animation bobFall;
    public static TextureRegion bobHit;
    public static Animation squirrelFly;
    public static TextureRegion platform;
    public static Animation brakingPlatform;
    public static Font font;
    public static Music music;
    public static Sound jumpSound;
    public static Sound highJumpSound;
    public static Sound hitSound;
    public static Sound coinSound;
    public static Sound clickSound;
```

В этом классе хранятся ссылки на все экземпляры классов `Texture`, `TextureRegion`, `Animation`, `Music` и `Sound` (текстуры, текстурные регионы, анимации, музыка и звуки),

которые понадобятся нам во время игры. Единственное, что мы не загружаем здесь, — изображения для экранов помощи.

```
public staticvoid load(GLGame game) {
    background = new Texture(game, "background.png");
    backgroundRegion = new TextureRegion(background, 0, 0, 320, 480);

    items = new Texture(game, "items.png");
    mainMenu = new TextureRegion(items, 0, 224, 300, 110);
    pauseMenu = new TextureRegion(items, 224, 128, 192, 96);
    ready = new TextureRegion(items, 320, 224, 192, 32);
    gameOver = new TextureRegion(items, 352, 256, 160, 96);
    highScoresRegion = new TextureRegion(Assets.items, 0, 257, 300, 110 / 3);
    logo = new TextureRegion(items, 0, 352, 274, 142);
    soundOff = new TextureRegion(items, 0, 0, 64, 64);
    soundOn = new TextureRegion(items, 64, 0, 64, 64);
    arrow = new TextureRegion(items, 0, 64, 64, 64);
    pause = new TextureRegion(items, 64, 64, 64, 64);

    spring = new TextureRegion(items, 128, 0, 32, 32);
    castle = new TextureRegion(items, 128, 64, 64, 64);
    coinAnim = new Animation(0.2f,
                             new TextureRegion(items, 128, 32, 32, 32),
                             new TextureRegion(items, 160, 32, 32, 32),
                             new TextureRegion(items, 192, 32, 32, 32),
                             new TextureRegion(items, 160, 32, 32, 32));
    bobJump = new Animation(0.2f,
                             new TextureRegion(items, 0, 128, 32, 32),
                             new TextureRegion(items, 32, 128, 32, 32));
    bobFall = new Animation(0.2f,
                             new TextureRegion(items, 64, 128, 32, 32),
                             new TextureRegion(items, 96, 128, 32, 32));
    bobHit = new TextureRegion(items, 128, 128, 32, 32);
    squirrelFly = new Animation(0.2f,
                             new TextureRegion(items, 0, 160, 32, 32),
                             new TextureRegion(items, 32, 160, 32, 32));
    platform = new TextureRegion(items, 64, 160, 64, 16);
    brakingPlatform = new Animation(0.2f,
                             new TextureRegion(items, 64, 160, 64, 16),
                             new TextureRegion(items, 64, 176, 64, 16),
                             new TextureRegion(items, 64, 192, 64, 16),
                             new TextureRegion(items, 64, 208, 64, 16));

    font = new Font(items, 224, 0, 16, 16, 20);
    music = game.getAudio().newMusic("music.mp3");
    music.setLooping(true);
    music.setVolume(0.5f);
    if(Settings.soundEnabled)
        music.play();
    jumpSound = game.getAudio().newSound("jump.ogg");
```

```
        highJumpSound = game.getAudio().newSound("highjump.ogg");
        hitSound = game.getAudio().newSound("hit.ogg");
        coinSound = game.getAudio().newSound("coin.ogg");
        clickSound = game.getAudio().newSound("click.ogg");
    }
```

Метод `load()`, который будет вызываться лишь один раз при запуске игры, отвечает за наполнение всех статических членов класса. Он загружает фоновое изображение и создает для него соответствующий текстурный регион (`TextureRegion`). Далее он загружает атлас текстур и создает все необходимые текстурные регионы и анимации (`Animation`). Сравните код с рис. 9.14 и прочими рисунками предыдущего раздела. Единственное, на что стоит обратить внимание при загрузке графических ресурсов, — создание анимации для монеты. Как мы говорили ранее, мы повторно используем второй кадр в конце анимационной последовательности кадров. Время, через которое кадры сменяют друг друга, — 0,2 секунды.

Мы также создаем экземпляр класса `Font`, который не обсуждался ранее. В нем будет реализована логика отрисовки текста с использованием растрового шрифта, встроенного в атлас текстур. Конструктор этого класса принимает текстуру (`Texture`), которая содержит глифы растрового шрифта, координаты в пикселах верхнего левого угла области, содержащей глифы, количество глифов в строке, а также размер каждого глифа в пикселах.

Кроме того, в этом методе загружаются музыка и звуки (экземпляры классов `Music` и `Sound`). Как вы можете видеть, мы вновь работаем с нашим старым добрым другом — классом `Settings`. Мы можем повторно использовать его реализацию, которую мы разработали для игры «Мистер Ном», внеся лишь одну небольшую модификацию, о которой вы узнаете через минуту. Обратите внимание, мы зацикливаем воспроизведение композиции и устанавливаем громкость ее звука равной 0,5, поэтому она будет звучать немного тише, чем звуковые эффекты. Музыка начнет проигрываться, если пользователь не отключил предварительно звук. Эта информация будет также храниться в классе `Settings`, как и в игре «Мистер Ном».

```
public static void reload() {
    background.reload();
    items.reload();
    if(Settings.soundEnabled)
        music.play();
    }
```

Далее рассмотрим таинственный метод, который называется `reload()`. Необходимо помнить, что контекст OpenGL ES будет теряться всякий раз, когда приложение будет ставиться на паузу. При возобновлении приложения следует каждый раз перезагружать текстуры — именно этим и занимается данный метод. Кроме того, возобновляется воспроизведение музыки, если, конечно, звук включен.

```
public static void playSound(Sound sound) {
    if(Settings.soundEnabled)
        sound.play(1);
    }
}
```

Последний метод этого класса — вспомогательный метод, который использован во всей остальной части кода, чтобы воспроизводить аудио. Вместо того чтобы каждый раз напрямую в коде проверять, включен ли звук, мы инкапсулируем эту проверку в отдельный метод.

Теперь взглянем на модифицированный класс Settings.

Класс Settings

В листинге 9.2 показан код несколько измененного класса Settings.

Листинг 9.2. Settings.java, несколько измененный класс Settings, который позаимствован у игры «Мистер Ном»

```
package com.badlogic.androidgames.jumper;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

import com.badlogic.androidgames.framework.FileIO;

public class Settings {
    public static boolean soundEnabled = true;
    public final static int[] highscores = new int[] { 100, 80, 50, 30, 10 };
    public final static String file = ".superjumper";

    public static void load(FileIO files) {
        BufferedReader in = null;
        try {
            in = new BufferedReader(new
                            InputStreamReader(files.readFile(file)));
            soundEnabled = Boolean.parseBoolean(in.readLine());
            for(int i = 0; i < 5; i++) {
                highscores[i] = Integer.parseInt(in.readLine());
            }
        } catch(IOExceptione) {
            // :( Не страшно, у нас есть значения по умолчанию
        } catch (NumberFormatExceptione) {
            // :/ Не страшно, опять же воспользуемся стандартными значениями
        } finally {
            try {
                if (in != null)
                    in.close();
            } catch (IOException e) {
            }
        }
    }

    public static void save(FileIO files) {
        BufferedWriter out = null;
```

```
        try {
            out = new BufferedWriter(new OutputStreamWriter(
                    files.writeFile(file)));
            out.write(Boolean.toString(soundEnabled));
            out.write("\n");
            for(int i = 0; i < 5; i++) {
                out.write(Integer.toString(highscores[i]));
                out.write("\n");
            }

        } catch (IOException e) {
        } finally {
            try {
                if (out != null)
                    out.close();
            } catch (IOException e) {
            }
        }
    }

    public static void addScore(int score) {
        for(int i=0; i < 5; i++) {
            if(highscores[i] < score) {
                for(int j= 4; j > i; j--)
                    highscores[j] = highscores[j-1];
                highscores[i] = score;
                break;
            }
        }
    }
}
```

Единственное отличие от класса `Settings` игры «Мистер Ном» заключается в том, что файл настроек, с которым осуществляется работа, имеет расширение не `.mrnom`, а `.superjumper`.

Основная активность

Как главную точку входа в нашу игру следует использовать `Activity`. Мы назовем ее `SuperJumper`. В листинге 9.3 показан ее исходный код.

Листинг 9.3. Класс SuperJumper.java, главная точка входа в игру

```
package com.badlogic.androidgames.jumper;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Screen;
import com.badlogic.androidgames.framework.impl.GLGame;

public class SuperJumper extends GLGame {
```

```
    boolean firstTimeCreate = true;

    @Override
    public Screen getStartScreen() {
        return new MainMenuScreen(this);
    }

    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        super.onSurfaceCreated(gl, config);
        if(firstTimeCreate) {
            Settings.load(getFileIO());
            Assets.load(this);
            firstTimeCreate = false;
        } else {
            Assets.reload();
        }
    }

    @Override
    public void onPause() {
        super.onPause();
        if(Settings.soundEnabled)
            Assets.music.pause();
    }
}
```

Из класса `GlGame` мы наследуем и реализуем метод `getStartScreen()`, который возвращает экземпляр `MainMenuScreen`. Два других метода чуть менее очевидны.

Переопределяем метод `onSurfaceCreate()`, который вызывается каждый раз при очередном создании контекста OpenGL ES (сравните новую реализацию с кодом в классе `GlGame` главы 6). Если этот метод вызывается в первый раз, мы используем метод `Assets.load()`для того, чтобы загрузить все активы в первый раз, а также загрузить настройки из файла, размещенного на карте памяти, если он доступен. В противном случае все, что нам нужно сделать, — перезагрузить текстуры и начать воспроизведение музыки с помощью метода `Assets.reload()`. Мы также переопределили метод `onPause()`, чтобы приостанавливать музыку, если она воспроизводится.

Мы выполняем оба эти действия, поэтому нет необходимости повторять их в методах `resume()` и `pause()`для экранов игры.

Перед тем как углубиться в реализацию классов экранов, взглянем на наш новый класс `Font`.

Класс Font

Для отрисовки случайного текста в формате ASCII мы будем использовать растровые шрифты. Принцип действия на высоком уровне уже обсуждался, поэтому просто взгляните на код листинга 9.4.

Листинг 9.4. Font.java, класс отрисовки растровых шрифтов

```
package com.badlogic.androidgames.framework.gl;

public class Font {
    public final Texture texture;
    public final int glyphWidth;
    public final int glyphHeight;
    public final TextureRegion[] glyphs = new TextureRegion[96];
```

Этот класс хранит текстуру, содержащую глифы шрифта, ширину и высоту одного глифа, а также массив TextureRegions — по одному региону на каждый глиф. Первый элемент массива хранит регион для глифа пробела, следующий хранит регион для глифа восклицательного знака и т. д. Другими словами, первый элемент соответствует ASCII-символу с кодом 32, а последний — символу с кодом 127.

```
    public Font(Texture texture,
                int offsetX, int offsetY,
                int glyphsPerRow, int glyphWidth, int glyphHeight) {
        this.texture = texture;
        this.glyphWidth = glyphWidth;
        this.glyphHeight = glyphHeight;
        int x = offsetX;
        int y = offsetY;
        for(int i = 0; i < 96; i++) {
            glyphs[i] = new TextureRegion(texture, x, y, glyphWidth,
                                          glyphHeight);
            x += glyphWidth;
            if(x == offsetX + glyphsPerRow * glyphWidth) {
                x = offsetX;
                y += glyphHeight;
            }
        }
    }
```

В конструкторе сохраняем конфигурацию растрового шрифта и генерируем регионы для глифов. Параметры offset и offsetY определяют верхний левый угол области текстуры, содержащей растровый шрифт. В созданном нами атласе текстур этот пиксел имеет координаты (224; 0). Параметр glyphsPerRow говорит о том, сколько глифов будет в строке, а параметры glyphWidth и glyphHeight определяют размер одного глифа. Поскольку мы используем моноширинный растровый шрифт, этот размер будет одинаковым для всех глифов. Параметр glyphWidth также является значением, на которое мы будем сдвигаться при отрисовке нескольких глифов.

```
    public void drawText(SpriteBatcher batcher, String text, float x,
                         float y) {
        int len = text.length();
        for(int i = 0; i < len; i++) {
            int c = text.charAt(i) - ' ';
            if(c < 0 || c > glyphs.length - 1)
                continue;

            TextureRegion glyph = glyphs[c];
            batcher.drawSprite(x, y, glyphWidth, glyphHeight, glyph);
```

```
                x += glyphWidth;
            }
        }
    }
```

Метод `drawText()` принимает экземпляр `SpriteBatcher`, строку текста и позиции x и y, откуда следует начинать рисовать текст. Координаты x и y определяют центр первого глифа. Мы получаем индекс каждого символа строки, проверяем, имеется ли для него глиф, и, если ответ положительный, отрисовываем его с помощью экземпляра класса `SpriteBatcher`. Далее увеличиваем координату x на значение `glyphWidth`. После этого мы готовы отрисовывать следующий символ строки.

Вы, возможно, задаетесь вопросом, почему нам не нужно привязывать текстуру, содержащую глифы. Предполагается, что это уже сделано перед вызовом метода `drawText()`. Причина заключается в том, что отрисовка текста может быть частью пакета, в этом случае текстура должна быть привязана заранее.

Почему необязательно привязывать ее снова в методе `drawText()`? Помните, для OpenGLES гораздо лучше, если состояния меняются лишь незначительно.

Конечно, с помощью класса `Font` можно работать лишь с моноширинными шрифтами. Если бы была необходимость в использовании стандартных шрифтов, нам понадобились бы данные о смещении каждого символа. Одним из решений проблемы является использование кернинга, как это было описано в подразделе «Обработка текста с помощью растровых шрифтов». Однако нашим требованиям удовлетворяет и подобное простое решение.

Экран GL

В примерах, приведенных в двух предыдущих главах, мы всегда получали ссылку на объект класса `GLGraphics` при помощи преобразования типов. Исправим это с помощью небольшого вспомогательного класса по имени `GLScreen`, который будет делать за нас всю грязную работу и хранить ссылку на объект `GLGraphics`. В листинге 9.5 показан код этого класса.

Листинг 9.5. Небольшой вспомогательный класс GLScreen.java

```
package com.badlogic.androidgames.framework.impl;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Screen;

public abstract class GLScreen extends Screen {
    protected final GLGraphics glGraphics;
    protected final GLGame glGame;

    public GLScreen(Game game) {
        super(game);
        glGame = (GLGame)game;
        glGraphics = ((GLGame)game).getGLGraphics();
    }

}
```

Мы храним экземпляры классов GLGraphics и GLGame. Конечно же, программа выдаст ошибку, если экземпляр класса Game, передаваемый конструктору, не будет иметь тип GLGame. Но теперь можно быть уверенным, что этого не случится.

Экран главного меню

Этот экран возвращается методом SuperJumper.getStartScreen(), пользователь увидит его в первую очередь. Он отрисовывает фоновое изображение и элементы интерфейса, а затем ожидает нажатия одного из элементов. Основываясь на нажатом элементе мы либо изменяем конфигурацию (включение/выключение звука), либо переходим на новый экран. В листинге 9.6 содержится код этого класса.

Листинг 9.6. Класс MainMenuScreen.java: экран главного меню

```
package com.badlogic.androidgames.jumper;

import java.util.List;
import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.gl.Camera2D;
import com.badlogic.androidgames.framework.gl.SpriteBatcher;
import com.badlogic.androidgames.framework.impl.GLScreen;
import com.badlogic.androidgames.framework.math.OverlapTester;
import com.badlogic.androidgames.framework.math.Rectangle;
import com.badlogic.androidgames.framework.math.Vector2;

public class MainMenuScreen extends GLScreen {
    Camera2D guiCam;
    SpriteBatcher batcher;
    Rectangle soundBounds;
    Rectangle playBounds;
    Rectangle highscoresBounds;
    Rectangle helpBounds;
    Vector2 touchPoint;
```

Этот класс наследует от класса GLScreen, поэтому мы можем получить доступ к экземпляру класса GLGraphics проще.

В этом классе есть несколько членов. Первый — это экземпляр класса Camera2D, который называется guiCam. Нам также понадобится экземпляр класса SpriteBatcher для отрисовки фонового изображения и элементов пользовательского интерфейса. Чтобы определить нажатие пользователем одного из элементов интерфейса, мы будем применять прямоугольники (Rectangles). Поскольку мы задействуем класс Camera2D, нам также понадобится экземпляр класса Vector2, чтобы преобразовать координаты нажатия в координаты игрового мира.

```
    public MainMenuScreen(Game game) {
        super(game);
        guiCam = new Camera2D(glGraphics, 320, 480);
        batcher = new SpriteBatcher(glGraphics, 100);
```

```
        soundBounds = new Rectangle(0, 0, 64, 64);
        playBounds = new Rectangle(160 - 150, 200 + 18, 300, 36);
        highscoresBounds = new Rectangle(160 - 150, 200 - 18, 300, 36);
        helpBounds = new Rectangle(160 - 150, 200 - 18 - 36, 300, 36);
        touchPoint = new Vector2();
    }
```

В конструкторе инициализируются все члены класса. Здесь есть небольшая особенность. Экземпляр класса `Camera2D` позволяет работать с нашим целевым разрешением 320×480 пикселов. Нам нужно установить подходящие значения для ширины и высоты области видимости. Остальное OpenGL ES сделает на ходу. Однако обратите внимание на то, что начало координат по-прежнему находится в левом нижнем углу и ось устремится вверх. Мы будем использовать подобную GUI-камеру на всех экранах, имеющих элементы пользовательского интерфейса, поэтому становится возможно измерять их в пикселах, а не в координатах игрового мира. Конечно, мы немного жульничаем в случае, если разрешение экрана не соответствует целевому, но мы уже применяли этот трюк в игре «Мистер Ном», и ничего плохого не произошло. Поэтому координаты `Rectangle`, которые мы используем для каждого элемента, даны в пикселах.

```
@Override
public void update(float deltaTime) {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    game.getInput().getKeyEvents();

    int len = touchEvents.size();
    for(int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);
        if(event.type == TouchEvent.TOUCH_UP) {
            touchPoint.set(event.x, event.y);
            guiCam.touchToWorld(touchPoint);

            if(OverlapTester.pointInRectangle(playBounds, touchPoint)) {
                Assets.playSound(Assets.clickSound);
                game.setScreen(new GameScreen(game));
                return;
            }
            if(OverlapTester.pointInRectangle(highscoresBounds,
                                              touchPoint)) {
                Assets.playSound(Assets.clickSound);
                game.setScreen(new HighscoresScreen(game));
                return;
            }
            if(OverlapTester.pointInRectangle(helpBounds, touchPoint)) {
                Assets.playSound(Assets.clickSound);
                game.setScreen(new HelpScreen(game));
                return;
            }
            if(OverlapTester.pointInRectangle(soundBounds, touchPoint)) {
                Assets.playSound(Assets.clickSound);
                Settings.soundEnabled = !Settings.soundEnabled;
                if(Settings.soundEnabled)
```

```
                    Assets.music.play();
                else
                    Assets.music.pause();
            }
        }
    }
}
```

Далее рассмотрим метод `update()`. В цикле мы проходим по всем событиям `TouchEvents`, которые возвращает экземпляр класса `Input`, и проверяем их тип. Нам нужно обнаружить прикосновения к экрану. Если такое событие произошло, прежде всего необходимо преобразовать координаты прикосновения в координаты игрового мира. Поскольку камера установлена для работы с целевым разрешением, суть трансформации заключается лишь в простом преобразовании координаты по оси y для экрана размером 320×480 пикселов. Для меньших или больших по размеру экранов следует преобразовывать координаты прикосновения для целевого разрешения. Как только координаты прикосновения получены, можно сверить их с координатами расположения прямоугольников, содержащих элементы пользовательского интерфейса. Если таким образом были выбраны пункты меню `PLAY` (Играть), `HIGHSCORES` (Рекорды) или `HELP` (Помощь), осуществляется переход на соответствующий экран. Если была нажата кнопка звука, изменяются настройки и, соответственно, воспроизведение музыки либо начинается, либо приостанавливается. Обратите также внимание на то, что если нажатие было произведено с помощью метода `Assets.playSound()`, проигрывается звук щелчка.

```
    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        guiCam.setViewportAndMatrices();

        gl.glEnable(GL10.GL_TEXTURE_2D);

        batcher.beginBatch(Assets.background);
        batcher.drawSprite(160, 240, 320, 480,Assets.backgroundRegion);
        batcher.endBatch();

        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);

        batcher.beginBatch(Assets.items);

        batcher.drawSprite(160, 480 - 10 - 71, 274, 142, Assets.logo);
        batcher.drawSprite(160, 200, 300, 110, Assets.mainMenu);
        batcher.drawSprite(32, 32, 64, 64,
Settings.soundEnabled?Assets.soundOn:Assets.soundOff);

        batcher.endBatch();

        gl.glDisable(GL10.GL_BLEND);
    }
```

Метод present()не нуждается в подробном объяснении. Он очищает экран, устанавливает проекционные матрицы с помощью камеры и отрисовывает фоновое изображение и элементы пользовательского интерфейса. Поскольку у пользовательских элементов прозрачный фон, мы временно делаем доступным смешивание для того, чтобы отрисовать их. Фоновое изображение не нуждается в смешивании, поэтому мы отключаем его, чтобы сэкономить несколько циклов процессора. Опять же, помните о том, что пользовательские элементы отрисовываются в системе координат, которая начинается в левом нижнем углу экрана, а ось *y* стремится вверх.

```
    @Override
    public void pause() {
        Settings.save(game.getFileIO());
    }

    @Override
    public void resume() {
    }

    @Override
    publicvoiddispose() {
    }
}
```

Последний метод, делающий что-либо, — pause(). Здесь мы убеждаемся в том, что все настройки после манипуляций пользователя со звуком на этом экране сохранены на SD-карту.

Экраны помощи

Всего в игре пять экранов помощи, все они работают по одному принципу: загружают изображение экрана помощи, отрисовывают его и кнопку со стрелкой и ожидают нажатия этой кнопки, чтобы перейти на следующий экран. Единственное, что меняется, — загружаемое ими изображение, а также экран, на который осуществляется переход. По этой причине я приведу код только первого экрана помощи, с которого можно перейти на второй. Файлы, содержащие изображения экранов помощи, называются help1.png и т. д. вплоть до help5.png. Соответствующие классы называются HelpScreen, Help2Screen и т. д. Последний экран, Help5Screen, переходит обратно к экрану главного меню.

```
package com.badlogic.androidgames.jumper;

import java.util.List;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.gl.Camera2D;
import com.badlogic.androidgames.framework.gl.SpriteBatcher;
import com.badlogic.androidgames.framework.gl.Texture;
```

```
import com.badlogic.androidgames.framework.gl.TextureRegion;
import com.badlogic.androidgames.framework.impl.GLScreen;
import com.badlogic.androidgames.framework.math.OverlapTester;
import com.badlogic.androidgames.framework.math.Rectangle;
import com.badlogic.androidgames.framework.math.Vector2;

public class HelpScreen extends GLScreen {
    Camera2D guiCam;
    SpriteBatcher batcher;
    Rectangle nextBounds;
    Vector2 touchPoint;
    Texture helpImage;
    TextureRegion helpRegion;
```

В этом классе также есть несколько членов для хранения экземпляров класса камеры, `SpriteBatcher`, прямоугольника кнопки со стрелкой, вектора точки прикосновения, `Texture` и `TextureRegion` для изображения экрана помощи.

```
public HelpScreen(Game game) {
    super(game);

    guiCam = new Camera2D(glGraphics, 320, 480);
    nextBounds = new Rectangle(320 - 64, 0, 64, 64);
    touchPoint = new Vector2();
    batcher = new SpriteBatcher(glGraphics, 1);
}
```

В конструкторе инициализируются все члены класса. Это происходит примерно так же, как и в классе `MainMenuScreen`.

```
@Override
public void resume() {
        helpImage = new Texture(glGame, "help1.png" );
        helpRegion = new TextureRegion(helpImage, 0, 0, 320, 480);
    }

@Override
public void pause() {
    helpImage.dispose();
}
```

В методе `resume()` загружается соответствующая текстура экрана помощи и создается соответствующий `TextureRegion` для его отрисовки с помощью `SpriteBatcher`. Загрузка вынесена в этот метод, поскольку контекст OpenGL ES может быть утерян. Текстуры фонового изображения и элементов пользовательского интерфейса, как говорилось ранее, обрабатываются классами `Assets` и `SuperJumper`. В работе с ними на других экранах нет необходимости. Вдобавок мы удаляем текстуру изображения экрана помощи в методе `pause()`, чтобы очистить память.

```
@Override
public void update(float deltaTime) {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    game.getInput().getKeyEvents();
```

```
        int len = touchEvents.size();
        for(int i = 0; i < len; i++) {
            TouchEvent event = touchEvents.get(i);
            touchPoint.set(event.x, event.y);
            guiCam.touchToWorld(touchPoint);

            if(event.type == TouchEvent.TOUCH_UP) {
                if(OverlapTester.pointInRectangle(nextBounds, touchPoint)) {
                    Assets.playSound(Assets.clickSound);
                    game.setScreen(new HelpScreen2(game));
                    return;
                }
            }
        }
    }
```

Далее следует метод `update()`, в котором выполняется простая проверка нажатия кнопки со стрелкой, при котором происходит переход на следующий экран. При нажатии также воспроизводится звук щелчка.

```
    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        guiCam.setViewportAndMatrices();

        gl.glEnable(GL10.GL_TEXTURE_2D);

        batcher.beginBatch(helpImage);
        batcher.drawSprite(160, 240, 320, 480, helpRegion);
        batcher.endBatch();

        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);

        batcher.beginBatch(Assets.items);
        batcher.drawSprite(320 – 32, 32, -64, 64, Assets.arrow);
        batcher.endBatch();

        gl.glDisable(GL10.GL_BLEND);
    }

    @Override
    public void dispose() {
    }
}
```

В методе `present()` очищается экран, устанавливаются значения матриц, отрисовывается изображение экрана за один раз, а затем отрисовывается кнопка со стрелкой. Конечно, здесь нет необходимости отрисовывать фоновое изображение, поскольку изображение экрана помощи уже содержит его.

Как было отмечено ранее, другие экраны помощи имеют аналогичную реализацию.

Экран лучших результатов

Далее в нашем списке следует экран лучших результатов. Для его создания будут использованы несколько меток пользовательского интерфейса главного меню (участок HIGHSCORES (Рекорды)). Лучшие результаты, хранящиеся в экземпляре класса Settings, будут отрисовываться с помощью объекта класса Font, который хранится в экземпляре класса Assets. Конечно, на этом экране также будет кнопка со стрелкой, с помощью которой игрок сможет вернуться в главное меню. В листинге 9.7 показан код класса этого экрана.

Листинг 9.7. Класс HighscoresScreen.java: экран лучших результатов

```
package com.badlogic.androidgames.jumper;

import java.util.List;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.gl.Camera2D;
import com.badlogic.androidgames.framework.gl.SpriteBatcher;
import com.badlogic.androidgames.framework.impl.GLScreen;
import com.badlogic.androidgames.framework.math.OverlapTester;
import com.badlogic.androidgames.framework.math.Rectangle;
import com.badlogic.androidgames.framework.math.Vector2;

public class HighscoreScreen extends GLScreen {
    Camera2D guiCam;
    SpriteBatcher batcher;
    Rectangle backBounds;
    Vector2 touchPoint;
    String[] highScores;
    float xOffset = 0;
```

Как обычно, в этом классе есть несколько членов для камеры, SpriteBatcher, границы кнопки со стрелкой и т. д. В массиве лучших результатов хранятся отформатированные строки, представляющие каждый лучший результат, которые демонстрируется игроку. Член класса xOffset — это значение, которое вычисляется для определения смещения каждой строки, поэтому строки находятся в середине экрана.

```
    public HighscoreScreen(Game game) {
        super(game);

        guiCam = new Camera2D(glGraphics, 320, 480);
        backBounds = new Rectangle(0, 0, 64, 64);
        touchPoint = new Vector2();
        batcher = new SpriteBatcher(glGraphics, 100);
        highScores = new String[5];
        for(int i = 0; i < 5; i++) {
```

```
            highScores[i] = (i + 1) + ". " + Settings.highscores[i];
            xOffset = Math.max(highScores[i].length() *
                      Assets.font.glyphWidth, xOffset);
        }
        xOffset = 160 - xOffset / 2;
    }
```

В конструкторе мы как обычно инициализируем все члены класса и вычисляем значение параметра `xOffset`. Это можно осуществить, оценив размер самой длинной из пяти строк, созданных нами для пяти лучших результатов. Поскольку мы используем моноширинный растровый шрифт, можно с легкостью посчитать количество пикселов, необходимых для одной строки текста, умножив количество символов на ширину глифа. Такой способ, конечно, не годится для непечатаемых символов или для символов, находящихся за пределами таблицы ASCII. Поскольку мы уверены, что не будем использовать такие символы, можно обойтись подобными простыми вычислениями. Потом в последней строке конструктора из 160 (горизонтальный центр экрана с целевым разрешением 320×480 пикселов) вычитается половина значения ширины самой длинной строки и подстраивается это значение далее путем вычитания половины ширины глифа. Это необходимо, поскольку метод `Font.drawText()` работает с центром глифа вместо одной из крайних точек.

```
    @Override
    public void update(float deltaTime) {
        List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
        game.getInput().getKeyEvents();
        int len = touchEvents.size();
        for(int i = 0; i < len; i++) {
            TouchEvent event = touchEvents.get(i);
            touchPoint.set(event.x, event.y);
            guiCam.touchToWorld(touchPoint);

            if(event.type == TouchEvent.TOUCH_UP) {
                if(OverlapTester.pointInRectangle(backBounds, touchPoint)) {
                    game.setScreen(new MainMenu(game));
                    return;
                }
            }
        }
    }
```

Метод `update()` выполняет простую проверку — если была нажата кнопка со стрелкой, воспроизводится звук щелчка и осуществляется обратный переход на экран основного меню.

```
    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        guiCam.setViewportAndMatrices();
```

```
        gl.glEnable(GL10.GL_TEXTURE_2D);

        batcher.beginBatch(Assets.background);
        batcher.drawSprite(160, 240, 320, 480, Assets.backgroundRegion);
        batcher.endBatch();

        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);

        batcher.beginBatch(Assets.items);
        batcher.drawSprite(160, 360, 300, 33, Assets.highScoresRegion);

        float y = 240;
        for(int i = 4; i >= 0; i--) {
            Assets.font.drawText(batcher, highScores[i], xOffset, y);
            y += Assets.font.glyphHeight;
        }

        batcher.drawSprite(32, 32, 64, 64, Assets.arrow);
        batcher.endBatch();

        gl.glDisable(GL10.GL_BLEND);
    }

    @Override
    public void resume() {
    }

    @Override
    public void pause() {
    }

    @Override
    public void dispose() {
    }
}
```

В этом классе метод `present()` также довольно прямолинеен. Он очищает экран, устанавливает значения матриц, отрисовывает фоновое изображение и все метки главного меню, относящиеся в лучшим результатам. Далее с использованием рассчитанного в конструкторе значения `xOffset` отрисовываются пять строк, содержащих лучшие результаты. Теперь мы можем заметить, почему класс `Font` не выполняет никаких привязок текстур: мы можем объединить пять вызовов метода `Font.drawText()`. Конечно, необходимо убедиться в том, что экземпляр класса `SpriteBatcher` может принять необходимое для отрисовки наших текстов количество спрайтов (или в нашем случае — глифов). Убедиться в этом можно при его создании в конструкторе, где в качестве максимального значения устанавливается 100 спрайтов (глифов).

Теперь взглянем на классы эмуляции игрового мира.

Классы эмуляции игрового мира

Перед тем как мы сможем погрузиться в игровые экраны, нам необходимо создать классы эмуляции игрового мира. Как и в случае с игрой «Мистер Ном», в игре будет класс для каждого игрового объекта и связывающий их все суперкласс по имени `World`, который объединяет все мелкие детали и оживляет игровой мир. Нам понадобятся классы для следующих объектов:

- Боб;
- белки;
- пружины;
- монеты;
- платформы.

Боб, белки и платформы могут передвигаться, поэтому их классы будут основаны на классе `DynamicGameObject`, который мы создали ранее. Пружины и монеты являются статическими, поэтому они будут наследовать от класса `GameObject`. Приведем основные задачи для классов эмуляции игрового мира:

- хранение позиции, скорости и фигуры, ограничивающей объект;
- хранение состояния и времени нахождения в этом состоянии (время состояния), если это необходимо;
- предоставление метода `update()`, который будет изменять объект в соответствии в его поведением;
- предоставление методов, изменяющих состояние объекта (например, сообщающих Бобу о том, что он погиб или коснулся пружины).

Класс `World` будет отслеживать многочисленные экземпляры этих игровых объектов, обновлять их каждый кадр, определять столкновения между игровыми объектами и Бобом, а также генерировать последствия таких столкновений (например, позволять Бобу погибнуть, подобрать монету и т. д.). Рассмотрим каждый класс: от самого простого до самого сложного.

Класс пружины

Начнем с рассмотрения кода класса пружины, приведенного в листинге 9.8.

Листинг 9.8. Spring.java, класс пружины

```
packagecom.badlogic.androidgames.jumper;

import com.badlogic.androidgames.framework.GameObject;

public class Spring extends GameObject {
    publicstatic float SPRING_WIDTH = 0.3f;
    publicstatic float SPRING_HEIGHT = 0.3f;

    public Spring(float x, float y) {
        super(x, y, SPRING_WIDTH, SPRING_HEIGHT);
    }
}
```

Класс Spring наследует от класса GameObject: нам необходимо знать только позицию и ограничивающую фигуру пружины, поскольку она неподвижна.

Далее мы определяем две константы, которые доступны всем: ширину и высоту пружины в метрах. Эти значения уже были определены заранее, поэтому нам остается просто повторно использовать их в этом классе.

Финальным аккордом является конструктор, принимающий координаты центра пружины по осям x и y. Зная эти значения, становится возможно вызвать конструктор суперкласса GameObject, принимающего позицию, а также ширину и высоту пружины, чтобы создать ограничивающую фигуру (прямоугольник с центром в заданной точке). Теперь пружина полностью определена, поскольку имеет позицию и ограничивающую фигуру, с которой может столкнуться Боб.

Класс монеты

Далее рассмотрим класс монеты (листинг 9.9).

Листинг 9.9. Coin.java, класс монеты

```
packagecom.badlogic.androidgames.jumper;

import com.badlogic.androidgames.framework.GameObject;

publicclass Coin extends GameObject {
    public static final float COIN_WIDTH = 0.5f;
    public static final float COIN_HEIGHT = 0.8f;
    public static final int COIN_SCORE = 10;

    float stateTime;
    public Coin(float x, float y) {
        super(x, y, COIN_WIDTH, COIN_HEIGHT);
        stateTime = 0;
    }

    public void update(float deltaTime) {
        stateTime += deltaTime;
    }
}
```

Класс Coin выглядит практически так же, как и класс Spring, он имеет только одно отличие: необходимо отслеживать продолжительность жизни монеты. Эта информация потребуется далее, когда нам понадобится отрисовать монету, используя экземпляр класса Animation. Мы делали то же самое для пещерного человека, героя последнего примера предыдущей главы. Этот прием мы используем для всех классов эмуляции игрового мира. Основываясь на состоянии и времени, на протяжении которого объект находится в этом состоянии, мы можем выбрать анимацию, а также текущий кадр этой анимации для отрисовки. У монеты есть всего одно состояние, поэтому необходимо отслеживать лишь время нахождения в этом состоянии. Для этого у класса есть метод update(), увеличивающий значение времени состояния каждый раз, когда ему передается параметр deltaTime.

Константы, заданные в верхней части класса, определяют рассчитанные нами заранее ширину и высоту монеты, а также количество очков, которые заработает Боб, прикоснувшись к монете.

Класс замка

Далее рассмотрим класс замка, находящегося на вершине нашего игрового мира (листинг 9.10).

Листинг 9.10. Castle.java, класс замка

```
package com.badlogic.androidgames.jumper;

import com.badlogic.androidgames.framework.GameObject;

public class Castle extends GameObject {
    public static float CASTLE_WIDTH = 1.7f;
    public static float CASTLE_HEIGHT = 1.7f;

    public Castle(float x, float y) {
        super(x, y, CASTLE_WIDTH, CASTLE_HEIGHT);
    }

}
```

Этот класс не очень сложен. Нам нужно сохранить позицию замка и его границы. Размер замка определяется константами CASTLE_WIDTH и CASTLE_HEIGHT, для которых использованы значения, рассмотренные нами ранее.

Класс белки

Теперь рассмотрим класс Squirrel (листинг 9.11).

Листинг 9.11. Squirrel.java, класс белки

```
package com.badlogic.androidgames.jumper;

import com.badlogic.androidgames.framework.DynamicGameObject;

public class Squirrel extends DynamicGameObject {
    public static final float SQUIRREL_WIDTH = 1;
    public static final float SQUIRREL_HEIGHT = 0.6f;
    public static final float SQUIRREL_VELOCITY = 3f;

    float stateTime = 0;

    public Squirrel(float x, float y) {
        super(x, y, SQUIRREL_WIDTH, SQUIRREL_HEIGHT);
        velocity.set(SQUIRREL_VELOCITY, 0);
    }

    public void update(float deltaTime) {
        position.add(velocity.x * deltaTime, velocity.y * deltaTime);
        bounds.lowerLeft.set(position).sub(SQUIRREL_WIDTH / 2,
```

```
                                        SQUIRREL_HEIGHT / 2);

        if(position.x < SQUIRREL_WIDTH / 2 ) {
            position.x = SQUIRREL_WIDTH / 2;
            velocity.x = SQUIRREL_VELOCITY;
        }
        if(position.x > World.WORLD_WIDTH - SQUIRREL_WIDTH / 2) {
            position.x = World.WORLD_WIDTH - SQUIRREL_WIDTH / 2;
            velocity.x = -SQUIRREL_VELOCITY;
        }
        stateTime += deltaTime;
    }
}
```

Белки — перемещающиеся объекты, поэтому их класс наследует от класса `DynamicGameObject`, что позволит нам задать векторы скорости и ускорения. Первое, что нам необходимо сделать, — задать размеры белки, а также ее скорость. Поскольку белки анимированы, необходимо также отслеживать время, которое белка находилась в определенном состоянии. У белки есть только одно состояние, как и у монеты: перемещение по горизонтали. Направление ее перемещения может быть определено с помощью компоненты вектора скорости по оси x, поэтому нет необходимости хранить время нахождения белки в этом состоянии. В конструкторе мы, конечно же, вызываем конструктор суперкласса, в который передается исходная позиция белки. Мы также устанавливаем вектор скорости белки равным (`SQUIRREL_VELOCITY`; `0`). Все белки сначала будут передвигаться вправо.

Метод `update()` обновляет позицию и ограничивающую фигуру белки, основываясь на скорости и прошедшем времени. Это стандартный этап интеграции Эйлера, который мы обсуждали и активно использовали в предыдущей главе. Необходимо также проверять столкновение белки с левым или правым краями мира. Если столкновение произошло, просто меняем значение вектора скорости на противоположный, и белка начинает двигаться в другом направлении. Ширина нашего мира ограничена 10 м, как мы уже говорили ранее. Последнее, что нам необходимо сделать, — обновить время состояния на значение `deltaTime`, что позволит определить, какую из двух анимаций следует использовать для отрисовки белки в дальнейшем.

Класс платформы

Код класса `Platform` приведен в листинге 9.12.

Листинг 9.12. `Platform.java`, класс платформы

```
package com.badlogic.androidgames.jumper;

import com.badlogic.androidgames.framework.DynamicGameObject;

public class Platform extends DynamicGameObject {
    public static final float PLATFORM_WIDTH = 2;
    public static final float PLATFORM_HEIGHT = 0.5f;
    public static final int PLATFORM_TYPE_STATIC = 0;
    public static final int PLATFORM_TYPE_MOVING = 1;
    public static final int PLATFORM_STATE_NORMAL = 0;
```

```
public static final int PLATFORM_STATE_PULVERIZING = 1;
public static final float PLATFORM_PULVERIZE_TIME = 0.2f * 4;
public static final float PLATFORM_VELOCITY = 2;
```

Конечно, платформы — объекты несколько более сложные, нежели белки. Рассмотрим константы, определенные в классе. Первые две константы задают ширину и высоту платформы, значения которых мы рассмотрели ранее. У каждой платформы есть свой тип, она может быть неподвижной или движущейся. Тип платформы описывается константами PLATFORM_TYPE_STATIC и PLATFORM_TYPE_MOVING. Платформа также может быть в одном из двух состояний: либо в нормальном состоянии (тогда она может быть статической или передвигающейся), либо она может рассыпаться. Эти состояния зашифрованы в константах PLATFORM_STATE_NORMAL и PLATFORM_STATE_PULVERIZING.

Рассыпание платформы — это процесс, ограниченный во времени. Поэтому необходимо определить время, требуемое для полного уничтожения платформы. Будем считать его равным 0,8 секунды. Это значение можно легко получить, приняв в расчет количество кадров анимации и продолжительность каждого кадра — это одна из небольших особенностей, не вписывающихся в шаблон MVC, которые следует учитывать, если вы пытаетесь следовать этому шаблону. Наконец, мы задаем скорость движущихся платформ равной 2 м/с, это значение также обсуждалось ранее. Передвигающаяся платформа ведет себя точно так же, как и белки, — перемещается в одном направлении до тех пор, пока не коснется края экрана, в этом случае она просто меняет направление.

```
int type;
int state;
float stateTime;

public Platform(int type, float x, float y) {
    super(x, y, PLATFORM_WIDTH, PLATFORM_HEIGHT);
    this.type = type;
    this.state = PLATFORM_STATE_NORMAL;
    this.stateTime = 0;
    if(type == PLATFORM_TYPE_MOVING) {
        velocity.x = PLATFORM_VELOCITY;
    }
}
```

Чтобы хранить тип, состояние и время, проведенное в определенном состоянии, нам понадобятся три члена класса. Они инициализируются в конструкторе, основываясь на типе платформы, который передается в конструктор как параметр (как и позиция центра платформы).

```
public void update(float deltaTime) {
    if(type == PLATFORM_TYPE_MOVING) {
        position.add(velocity.x * deltaTime, 0);
        bounds.lowerLeft.set(position).sub(PLATFORM_WIDTH / 2,
                                           PLATFORM_HEIGHT / 2);

        if(position.x < PLATFORM_WIDTH / 2) {
```

```
                velocity.x = -velocity.x;
                position.x = PLATFORM_WIDTH / 2;
            }
            if(position.x > World.WORLD_WIDTH - PLATFORM_WIDTH / 2) {
                velocity.x = -velocity.x;
                position.x = World.WORLD_WIDTH - PLATFORM_WIDTH / 2;
            }
        }

        stateTime += deltaTime;
    }
```

Метод `update()` перемещает платформу и проверяет, не вышла ли она за пределы игрового мира. В этом случае метод изменяет вектор ее ускорения. Точно такие же действия производил метод `Squirrel.update()`. В конце этого метода также обновляется время состояния.

```
    public void pulverize() {
        state = PLATFORM_STATE_PULVERIZING;
        stateTime = 0;
        velocity.x = 0;
    }
}
```

Последний метод этого класса называется `pulverize()`. Он переключает состояние платформы из `PLATFORM_STATE_NORMAL` в `PLATFORM_STATE_PULVERIZING`, а также сбрасывает время состояния и скорость. Это означает, что платформа перестает перемещаться. Данный метод вызывается в том случае, если класс `World` определяет столкновение Боба и платформы и определяет, что платформа должна рассыпаться, основываясь на сгенерированном случайном числе. Чуть позже мы поговорим об этом более подробно.

Класс Боба

Сначала нам необходимо поговорить о самом Бобе. Код, реализующий его класс, приведен в листинге 9.13.

Листинг 9.13. Класс `Bob.java`

```
package com.badlogic.androidgames.jumper;

import com.badlogic.androidgames.framework.DynamicGameObject;

public class Bob extends DynamicGameObject{
    public static final int BOB_STATE_JUMP = 0;
    public static final int BOB_STATE_FALL = 1;
    public static final int BOB_STATE_HIT = 2;
    public static final float BOB_JUMP_VELOCITY = 11;
    public static final float BOB_MOVE_VELOCITY = 20;
    public static final float BOB_WIDTH = 0.8f;
    public static final float BOB_HEIGHT = 0.8f;
```

Мы снова начинаем с описания некоторых констант. Боб может находиться в трех состояниях: он может прыгать вверх, падать вниз или быть погибшим. У него так-

же есть вертикальная скорость прыжка, которая направлена вдоль оси *y*, и горизонтальная скорость перемещения по экрану, которая направлена вдоль оси *x*. Последние две константы определяют размеры Боба в игровом мире.

Конечно же, необходимо также хранить состояние Боба и время, которое он в нем находится.

```
int state;
float stateTime;

public Bob(float x, float y) {
    super(x, y, BOB_WIDTH, BOB_HEIGHT);
    state = BOB_STATE_FALL;
    stateTime = 0;
}
```

В конструкторе класса Боба всего лишь вызывается конструктор суперкласса для того, чтобы центральная позиция Боба и его ограничивающая фигура были корректно инициализированы, а также инициализируются переменные `state` и `stateTime`.

```
public void update(float deltaTime) {
    velocity.add(World.gravity.x * deltaTime, World.gravity.y *
                                    deltaTime);
    position.add(velocity.x * deltaTime, velocity.y * deltaTime);
    bounds.lowerLeft.set(position).sub(bounds.width / 2, bounds.height / 2);

    if(velocity.y > 0 && state != BOB_STATE_HIT) {
        if(state != BOB_STATE_JUMP) {
            state = BOB_STATE_JUMP;
            stateTime = 0;
        }
    }

    if(velocity.y < 0 && state != BOB_STATE_HIT) {
        if(state != BOB_STATE_FALL) {
            state = BOB_STATE_FALL;
            stateTime = 0;
        }
    }
    if(position.x < 0)
        position.x = World.WORLD_WIDTH;
    if(position.x > World.WORLD_WIDTH)
        position.x = 0;

    stateTime += deltaTime;
}
```

Метод `update()` начинается с обновления позиции Боба и ограничивающей его фигуры, основываясь на гравитации и его текущей скорости. Обратите внимание на то, что скорость высчитывается с учетом гравитации и передвижения Боба как по горизонтали, так и по вертикали с помощью прыжков. Две следующие условные конструкции устанавливают состояние Боба равным либо `BOB_STATE_JUMPING`,

либо BOB_STATE_FALLING и сбрасывают его время состояния. Все эти действия производятся на основании компоненты *y* его скорости. Если она больше нуля — значит Боб прыгает, если меньше — падает. Однако эти действия выполняются лишь в случае, если Боб жив, но находится в неправильном состоянии. Если бы эти действия производились всегда, время состояния было бы равно нулю, что неблагоприятно сказалось бы на анимации Боба в дальнейшем. Этот метод также перемещает Боба с одного конца в другой, если он покинул границы игрового мира слева или справа. Наконец, в этом методе снова обновляется член класса stateTime.

Когда же Боб отделяет свою скорость от гравитации? Для этого служат другие методы.

```
    public void hitSquirrel() {
        velocity.set(0,0);
        state = BOB_STATE_HIT;
        stateTime = 0;
    }

    public void hitPlatform() {
        velocity.y = BOB_JUMP_VELOCITY;
        state = BOB_STATE_JUMP;
        stateTime = 0;
    }

    public void hitSpring() {
        velocity.y = BOB_JUMP_VELOCITY * 1.5f;
        state = BOB_STATE_JUMP;
        stateTime = 0;
    }
}
```

Метод hitSquirrel() вызывается классом World, если Боб прикоснется к белке. Если этого произошло, Боб перестает передвигаться и входит в состояние BOB_STATE_HIT. С этого момента на Боба действует лишь гравитация, игрок более не может управлять персонажем, а сам Боб не взаимодействует с платформами. Подобное поведение демонстрируется в игре «Супер Марио»: если к Марио прикоснется враг, Марио просто упадет вниз.

Метод hitPlatform() также вызывается классом World. Это делается в том случае, если Боб прикоснется к платформе при падении вниз. Если это произошло, скорость Боба по оси *y* становится равной BOB_JUMP_VELOCITY, а также соответственно устанавливаются его состояние и время состояния. С этого момента Боб будет двигаться вверх, пока гравитация снова не победит, заставив его опять падать.

Последний метод, hitSpring(), вызывается классом World, если Боб прикоснется к пружине. Он делает то же самое, что и метод hitPlatform(), но с одним различием: скорость Боба устанавливается равной значению BOB_JUMP_VELOCITY, умноженному на 1,5. Это означает, что Боб при касании пружины подпрыгнет немного выше по сравнению с результатом касания платформы.

Класс World

Последний класс, который нам необходимо обсудить, — `World`. Он немного больше по размеру, чем предыдущие классы, поэтому стоит разбить его на несколько частей. В листинге 9.14 показана первая часть его кода.

Листинг 9.14. Фрагменты классаWorld.java: константы, члены класса и инициализация

```
package com.badlogic.androidgames.jumper;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import com.badlogic.androidgames.framework.math.OverlapTester;
import com.badlogic.androidgames.framework.math.Vector2;

public class World {
    public interface WorldListener {
        public void jump();
        public void highJump();
        public void hit();
        public void coin();
    }
```

Первое, что мы определяем в этом классе, — интерфейс, который называется `WorldListener`. Вы спросите, для чего он нужен? Он необходим для решения небольшой проблемы, которую ставит перед нами MVC: когда следует проигрывать звуковые эффекты? Для решения этой проблемы можно просто вызывать метод `Assets.playSound()` из соответствующих классов эмуляции, но такой подход не совсем корректен с точки зрения проектирования. Вместо этого мы позволим пользователю класса `World` зарегистрировать `WorldListener`, который будет вызываться, когда Боб прыгает с платформы или пружины, касается белки или подбирает монету. Далее зарегистрируем слушателя, который будет воспроизводить соответствующие этим событиям звуки, что позволит освободить классы эмуляции от прямых зависимостей, связанных с отрисовкой и воспроизведением аудио.

```
    public static final float WORLD_WIDTH = 10;
    public static final float WORLD_HEIGHT = 15 * 20;
    public static final int WORLD_STATE_RUNNING = 0;
    public static final int WORLD_STATE_NEXT_LEVEL = 1;
    public static final int WORLD_STATE_GAME_OVER = 2;
    public static final Vector2 gravity = new Vector2(0, -12);
```

После этого необходимо определить несколько констант. Константы `WORLD_WIDTH` и `WORLD_HEIGHT` задают вертикальные и горизонтальные границы игрового мира. Помните, что область видимости в нашей игре имеет площадь 10×15 м. Основываясь на константах, представленных здесь, игровой мир будет простираться на 20 областей видимости или экранов по вертикали. Опять же, к этому значению я пришел после тестирования. Мы вернемся к этому моменту, когда будем обсуждать генерацию уровней. Камера игрового мира также может быть в трех состояниях: перемещаться,

ожидать начала уровня или находиться в неподвижном состоянии, если игра закончилась (в это время Боб падает очень далеко, за пределы области видимости). Здесь также определяется как константа вектор гравитационного ускорения.

```
public final Bob bob;
public final List<Platform> platforms;
public final List<Spring> springs;
public final List<Squirrel> squirrels;
public final List<Coin> coins;
public Castle castle;
public final WorldListener listener;
public final Random rand;

public float heightSoFar;
public int score;
public int state;
```

Здесь перечислены все переменные — члены класса `World`. Этот класс следит за Бобом, всеми платформами, пружинами, белками, монетами и замком. Он также имеет ссылку на интерфейс `WorldListener` и экземпляр класса `Random`, который будет использоваться для генерации случайных чисел для различных нужд. Последние три члена отслеживают самую большую координату по оси *y*, на которой был Боб, состояние камеры игрового мира и количество набранных игроком очков.

```
public World(WorldListener listener) {
    this.bob = new Bob(5, 1);
    this.platforms = new ArrayList<Platform>();
    this.springs = new ArrayList<Spring>();
    this.squirrels = new ArrayList<Squirrel>();
    this.coins = new ArrayList<Coin>();
    this.listener = listener;
    rand = new Random();
    generateLevel();

    this.heightSoFar = 0;
    this.score = 0;
    this.state = WORLD_STATE_RUNNING;
}
```

В конструкторе инициализируются все члены класса, а также сохраняется экземпляр класса `WorldListener`, переданный как параметр. Боб помещается по центру оси *x* немного выше нулевой координаты по оси *y* (в точке (5; 1)). Остальная часть кода говорит сама за себя, с одним лишь исключением: метод `generateLevel()` не так прозрачен.

Генерация игрового мира

Вы могли задаться вопросом — как же происходит создание и размещение объектов игрового мира? Для этого использован метод, который называется процедурной генерацией. Придумаем простой алгоритм, генерирующий случайный уровень (листинг 9.15).

Листинг 9.15. Фрагменты класса World.java: метод generateLevel()

```
private void generateLevel() {
    float y = Platform.PLATFORM_HEIGHT / 2;
    float maxJumpHeight = Bob.BOB_JUMP_VELOCITY * Bob.BOB_JUMP_VELOCITY
            / (2 * -gravity.y);
    while (y < WORLD_HEIGHT - WORLD_WIDTH / 2) {
        int type = rand.nextFloat() > 0.8f ? Platform.PLATFORM_TYPE_MOVING
                : Platform.PLATFORM_TYPE_STATIC;
        float x = rand.nextFloat()
                * (WORLD_WIDTH - Platform.PLATFORM_WIDTH)
                + Platform.PLATFORM_WIDTH / 2;

        Platform platform = new Platform(type, x, y);
        platforms.add(platform);

        if (rand.nextFloat() > 0.9f
                && type != Platform.PLATFORM_TYPE_MOVING) {
            Spring spring = new Spring(platform.position.x,
                    platform.position.y + Platform.PLATFORM_HEIGHT / 2
                            + Spring.SPRING_HEIGHT / 2);
            springs.add(spring);
        }

        if (y > WORLD_HEIGHT / 3 && rand.nextFloat() > 0.8f) {
            Squirrel squirrel = new Squirrel(platform.position.x
                    + rand.nextFloat(), platform.position.y
                    + Squirrel.SQUIRREL_HEIGHT + rand.nextFloat() * 2);
            squirrels.add(squirrel);
        }

        if (rand.nextFloat() > 0.6f) {
            Coin coin = new Coin(platform.position.x + rand.nextFloat(),
                    platform.position.y + Coin.COIN_HEIGHT
                            + rand.nextFloat() * 3);
            coins.add(coin);
        }

        y += (maxJumpHeight - 0.5f);
        y -= rand.nextFloat() * (maxJumpHeight / 3);
    }

    castle = new Castle(WORLD_WIDTH / 2, y);
}
```

Опишем основную идею алгоритма простыми словами.

1. Начинаем с низшей точки игрового мира $y = 0$.
2. Если вершина мира еще не достигнута, выполняем следующее:
   1) создаем платформу, неподвижную или двигающуюся в текущей позиции по оси y и случайной позиции по оси x;

2) генерируем случайное число между 0 и 1 и, если оно больше 0,9 и платформа неподвижна, создаем на этой платформе пружину;
3) если мы уже достигли высоты хотя бы трети мира, генерируем случайное число. Если оно больше 0,8, создаем белку и помещаем ее со случайным смещением по оси y от позиции платформы;
4) генерируем случайное число и, если оно больше 0,6, создаем монету и помещаем ее со случайным смещением по оси y от позиции платформы;
5) увеличиваем y на максимальную высоту нормального прыжка, немного уменьшаем ее (но не настолько, чтобы y стал меньше своего предыдущего значения) и переходим к пункту 2.

3. Помещаем в последней позиции по оси y замок, расположенный посередине оси x.

Изюминка этой процедуры — это способ увеличения позиции y в пункте 5 шага 2. Необходимо быть уверенным в том, что каждая последующая платформа может быть достигнута Бобом после прыжка с текущей платформы. Боб может прыгать так высоко, как ему может позволить гравитация. Начальная скорость его прыжка — 11 м/с по вертикали. Высоту прыжка Боба можно рассчитать по следующей формуле:

```
Высота = скорость * скорость / (2 * гравитация) = 11 * 11 / (2 * 13) ~= 4,6 м
```

Это означает, что если между платформами будет расстояние, равное по вертикали 4,6, Боб все еще сможет до них допрыгнуть. Чтобы убедиться, что Боб может достичь любой платформы, используем значение, которое немного меньше максимальной высоты его прыжка. Это гарантирует, что Боб сможет допрыгнуть с одной платформы на другую. Горизонтальное размещение платформы также случайно. Основываясь на том, что скорость горизонтального перемещения Боба равна 20 м/с, мы можем быть более чем уверены в том, что персонаж сможет достичь любой платформы как по вертикали, так и по горизонтали.

Другие объекты создаются случайно. Метод `Random.nextFloat()` возвращает случайное число между 0 и 1 при всех своих вызовах. Каждое число может выпасть с одинаковой вероятностью. Белки создаются, только когда генерируется случайное число больше 0,8. Это означает, что новая белка появится с вероятностью 20 % (1 – 0,8). Эти слова также верны и для всех остальных объектов, создаваемых на основе случайного числа. Изменив эти значения, можно создать в игровом мире большее или меньшее количество объектов.

Обновление игрового мира

После генерации игрового мира мы можем обновлять его объекты и проверять наличие столкновений. В листинге 9.16 показаны методы класса World, используемые для обновления игрового мира.

Листинг 9.16. Фрагменты класса World.java: методы, необходимые для обновления игрового мира

```
public void update(float deltaTime, float accelX) {
    updateBob(deltaTime, accelX);
```

```
        updatePlatforms(deltaTime);
        updateSquirrels(deltaTime);
        updateCoins(deltaTime);
        if (bob.state != Bob.BOB_STATE_HIT)
            checkCollisions();
        checkGameOver();
    }
```

Метод `update()` в дальнейшем будет вызываться для игрового экрана. В качестве параметров он получает промежуток времени, прошедший с предыдущего вызова, а также показания акселерометра по оси *x*. Этот метод ответственен за вызов других методов обновления, а также за проверку столкновений и завершения игры. У нас есть метод обновления для каждого типа объектов игрового мира.

```
    private void updateBob(float deltaTime, float accelX) {
        if (bob.state != Bob.BOB_STATE_HIT && bob.position.y <= 0.5f)
            bob.hitPlatform();
        if (bob.state != Bob.BOB_STATE_HIT)
            bob.velocity.x = -accelX / 10 * Bob.BOB_MOVE_VELOCITY;
        bob.update(deltaTime);
        heightSoFar = Math.max(bob.position.y, heightSoFar);
    }
```

Метод `updateBob()` отвечает за обновление состояния Боба. Первое, что он делает, — проверяет, не касается ли Боб нижней границы игрового мира, в этом случае он должен прыгнуть. Это означает, что в начале каждого уровня Боб подпрыгивает от основания игрового мира. Как только самая нижняя платформа выпадает из поля зрения, эта проверка, конечно же, больше не срабатывает. Далее обновляется горизонтальная скорость Боба, основываясь на показаниях акселерометра, которые метод получает в качестве аргумента. Как мы говорили ранее, показания акселерометра приводятся от вида [–10; 10] к виду [–1; 1] (от полного наклона влево до полного наклона вправо), а затем умножаются на стандартную скорость горизонтального перемещения Боба. Далее этот метод приказывает Бобу обновить свое состояние с помощью вызова `Bob.update()`. Последнее, что происходит в этом методе, — сохранение максимального значения высоты, которого сумел достичь Боб. В дальнейшем оно понадобится для определения, упал Боб или нет.

```
    private void updatePlatforms(float deltaTime) {
        int len = platforms.size();
        for (int i = 0; i < len; i++) {
            Platform platform = platforms.get(i);
            platform.update(deltaTime);
            if (platform.state == Platform.PLATFORM_STATE_PULVERIZING
                    && platform.stateTime > Platform.PLATFORM_PULVERIZE_TIME) {
                platforms.remove(platform);
                len = platforms.size();
            }
        }
    }
```

Далее обновляются все платформы в методе updatePlatforms(). Мы проходим по списку платформ и вызываем для каждой из них метод update(), передавая в него текущее значение параметра deltaTime. Если платформа уже рассыпается, проверяем, как долго проходит этот процесс. Если платформа находится в состоянии PLATFORM_STATE_PULVERIZING время, большее, чем значение PLATFORM_PULVERIZE_TIME, она просто удаляется из списка платформ.

```
private void updateSquirrels(float deltaTime) {
    int len = squirrels.size();
    for (int i = 0; i < len; i++) {
        Squirrel squirrel = squirrels.get(i);
        squirrel.update(deltaTime);
    }
}

private void updateCoins(float deltaTime) {
    int len = coins.size();
    for (int i = 0; i < len; i++) {
        Coin coin = coins.get(i);
        coin.update(deltaTime);
    }
}
```

Метод updateSquirrels() обновляет каждый экземпляр класса Squirrel с помощью его метода update(), передавая ему в качестве аргумента время, прошедшее с последнего вызова метода. То же самое делается и для метода updateCoins().

Определение столкновений и реакция на них

Если вы снова взглянете на оригинальный метод World.update(), то увидите, что далее он определяет столкновения между Бобом и всеми прочими объектами игрового мира. Это делается только в том случае, если Боб не находится в состоянии BOB_STATE_HIT, в котором он просто продолжает лететь вниз под воздействием гравитации. Взглянем на код методов, определяющих столкновения (листинг 9.17).

Листинг 9.17. Фрагменты классаWorld.java: методы, определяющие столкновения

```
private void checkCollisions() {
    checkPlatformCollisions();
    checkSquirrelCollisions();
    checkItemCollisions();
    checkCastleCollisions();
}
```

Метод checkCollisions() также является суперметодом, который просто вызывает другие методы, определяющие столкновения. Боб может прикоснуться к следующим объектам игрового мира: платформы, белки, монеты, пружины и замок. Для каждого типа объектов существует отдельный метод. Следует помнить, что этот суперметод вызывается после того, как обновились позиции и ограничивающие фигуры всех объектов игрового мира. Такую ситуацию можно рассматривать как фотоснимок состояния игрового мира в заданный момент. Все, что остается сде-

лать, — просмотреть это неподвижное изображение и проверить, не пересекаются ли какие-либо объекты. Далее мы можем начать действовать и убедиться, что пересекшиеся объекты отреагируют подобающим образом, изменив свои состояния, позиции, скорости и т. д.

```
private void checkPlatformCollisions() {
    if (bob.velocity.y > 0)
        return;

    int len = platforms.size();
    for (int i = 0; i < len; i++) {
        Platform platform = platforms.get(i);
        if (bob.position.y > platform.position.y) {
            if (OverlapTester
                    .overlapRectangles(bob.bounds, platform.bounds)) {
                bob.hitPlatform();
                listener.jump();
                if (rand.nextFloat() > 0.5f) {
                    platform.pulverize();
                }
                break;
            }
        }
    }
}
```

Метод `checkPlatformCollisions()` проверяет пересечение Боба со всеми платформами игрового мира. Мы быстро выходим из этого метода, если Боб сейчас движется вверх. В этом случае Боб может проходить сквозь платформы снизу вверх. Для «Большого прыгуна» такое поведение подходит хорошо; в играх вроде «Супер Марио» нам скорее понадобится, чтобы Боб падал вниз при ударе о блок снизу. Далее проходим по всем платформам в цикле и проверяем, находится ли над текущей платформой главный герой. Если да, проверяем, пересекается ли ограничивающий его прямоугольник с ограничивающим прямоугольником платформы. В этом случае мы указываем Бобу, что он задел платформу, вызовом метода `Bob.hitPlatform()`. Если вернуться назад к данному методу, мы увидим, что это спровоцирует прыжок и соответственно изменит состояние Боба. Далее вызывается метод `WorldListener.jump()`, информирующий слушателя о том, что Боб только что начал прыгать. Это будет использовано для того, чтобы слушатель воспроизвел соответствующий звуковой эффект. Последнее, что делает данный метод, — генерация случайного числа. Если оно больше 0,5, мы приказываем платформе рассыпаться. Она просуществует еще `PLATFORM_PULVERIZE_TIME` секунд (0,8), а затем будет удалена методом `updatePlatforms()`, показанным ранее. Когда мы будем отрисовывать эту платформу, используем ее время состояния для того, чтобы определить, какой именно кадр анимации платформы следует отобразить.

```
private void checkSquirrelCollisions() {
    int len = squirrels.size();
    for (int i = 0; i < len; i++) {
```

```
            Squirrel squirrel = squirrels.get(i);
            if (OverlapTester.overlapRectangles(squirrel.bounds, bob.bounds)) {
                bob.hitSquirrel();
                listener.hit();
            }
        }
    }
```

Метод `checkSquirrelCollisions()` проверяет пересечение прямоугольника, ограничивающего Боба, с прямоугольниками, которые ограничивают каждую белку. Если Боб коснется белки, мы прикажем ему перейти в состояние `BOB_STATE_HIT`, что заставит его упасть, и игрок никак не сможет управлять им. Мы также извещаем об этом `WorldListener`, чтобы он проиграл соответствующий звуковой эффект.

```
private void checkItemCollisions() {
    int len = coins.size();
    for (int i = 0; i < len; i++) {
        Coin coin = coins.get(i);
        if (OverlapTester.overlapRectangles(bob.bounds, coin.bounds)) {
            coins.remove(coin);
            len = coins.size();
            listener.coin();
            score += Coin.COIN_SCORE;
        }

    }

    if (bob.velocity.y > 0)
        return;

    len = springs.size();
    for (int i = 0; i < len; i++) {
        Spring spring = springs.get(i);
        if (bob.position.y > spring.position.y) {
            if (OverlapTester.overlapRectangles(bob.bounds, spring.bounds)) {
                bob.hitSpring();
                listener.highJump();
            }
        }
    }
}
```

Метод `checkItemCollisions()` проверяет, не столкнулся ли Боб с какой-либо монетой или пружиной. Если Боб коснулся какой-либо монеты, она удаляется из игрового мира. Далее метод сообщает слушателю о том, что была подобрана монета, и увеличивает текущее количество очков игрока на `COIN_SCORE`. Если Боб падает, мы также осуществляем проверку касания какой-либо пружины. Если оно произошло, мы сообщаем Бобу об этом, из-за чего он подпрыгнет выше, чем обычно. Об этом событии также информируется слушатель.

```
private void checkCastleCollisions() {
    if (OverlapTester.overlapRectangles(castle.bounds, bob.bounds)) {
        state = WORLD_STATE_NEXT_LEVEL;
    }
}
```

Последний метод проверяет столкновение Боба и замка. Если Боб касается замка, состояние мира принимает значение `WORLD_STATE_NEXT_LEVEL`, сигнализируя всем внешним сущностям (таким как игровой экран) о том, что следует перейти на следующий уровень, который также будет представлять собой случайно сгенерированный экземпляр класса `World`.

Игра окончена, дружище!

Код последнего метода класса `World`, который вызывается в последней строке метода `World.update()`, приведен в листинге 9.18.

Листинг 9.18. Окончание класса World.java: метод, проверяющий, не закончилась ли игра

```
    private void checkGameOver() {
        if (heightSoFar - 7.5f > bob.position.y) {
            state = WORLD_STATE_GAME_OVER;
        }
    }
}
```

Вспомните, как мы определили состояние окончания игры: Боб должен покинуть нижнюю часть области видимости. Область видимости, конечно же, управляется экземпляром класса `Camera2D`, который имеет позицию. Координата этой позиции по оси y всегда равна наибольшей координате, которую сумел достичь Боб. Камера будет следовать за главным героем только в том случае, если он будет двигаться вверх. Поскольку мы хотим разделить код отрисовки и эмуляции, нам не нужна ссылка на камеру. Поэтому мы лишь отслеживаем наибольшую координату по оси y, достигнутую Бобом, в методе `updateBob()` и храним ее значение в переменной `heightSoFar`. Мы знаем, что область видимости имеет высоту 15 м. Мы также знаем, что если координата Боба по оси y стала меньше, чем значение выражения `heightSoFar - 7.5`, то он покинул область видимости снизу и должен быть признан мертвым. Конечно, этот метод немного ненадежен, поскольку он основан на предположении, что высота области видимости всегда будет равна 15 м, а камера всегда будет находиться в высшей точке, которую сумел достичь Боб. Если бы мы разрешили масштабирование или использовали другой способ перемещения камеры, такой метод стал бы неверным. Вместо того чтобы все излишне усложнять, давайте просто оставим все как есть. Вы будете часто встречать подобные решения при разработке игр, поскольку очень трудно постоянно принимать решения, прозрачные с точки зрения инженерии программного обеспечения (о чем свидетельствует злоупотребление нами членами класса, имеющими тип `public` или `package private`).

Возможно, вы заинтересовались, почему до сих пор не был использован класс SpatialHashGrid, который мы разработали в предыдущей главе. Я объясню причину через мгновение. Сначала закончим написание нашей игры, реализовав класс GameScreen.

Игровой экран

Мы практически закончили написание игры «Большой прыгун». Последнее, что осталось реализовать, — игровой экран, который позволит игроку увидеть игровой мир и взаимодействовать с ним. Игровой экран состоит из пяти подэкранов, что показано на рис. 9.2. Имеется экран готовности, экран нормальной работы игры, экран перехода на следующий уровень, экран окончания игры и экран паузы. Игровой экран игры «Мистер Ном» был похож на этот, в нем не было лишь экрана перехода на новый уровень, поскольку в игре был всего один уровень. Мы будем использовать тот же подход, что и для игры «Мистер Ном»: у нас будут отдельные методы обновления и отображения для каждого подэкрана, которые обновляют и отрисовывают игровой мир, а также элементы пользовательского интерфейса, являющиеся частью подэкранов. Поскольку код класса игрового экрана несколько больше кода прочих классов, разобьем его на несколько листингов. В листинге 9.19 показана первая часть кода игрового экрана.

Листинг 9.19. Фрагменты класса GameScreen.java: члены класса и конструктор

```
package com.badlogic.androidgames.jumper;

import java.util.List;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.gl.Camera2D;
import com.badlogic.androidgames.framework.gl.FPSCounter;
import com.badlogic.androidgames.framework.gl.SpriteBatcher;
import com.badlogic.androidgames.framework.impl.GLScreen;
import com.badlogic.androidgames.framework.math.OverlapTester;
import com.badlogic.androidgames.framework.math.Rectangle;
import com.badlogic.androidgames.framework.math.Vector2;
import com.badlogic.androidgames.jumper.World.WorldListener;

public class GameScreen extends GLScreen {
    static final int GAME_READY = 0;
    static final int GAME_RUNNING = 1;
    static final int GAME_PAUSED = 2;
    static final int GAME_LEVEL_END = 3;
    static final int GAME_OVER = 4;

    int state;
    Camera2D guiCam;
```

```
Vector2 touchPoint;
SpriteBatcher batcher;
World world;
WorldListener worldListener;
WorldRenderer renderer;
Rectangle pauseBounds;
Rectangle resumeBounds;
Rectangle quitBounds;
int lastScore;
String scoreString;
```

Класс начинается с описания нескольких констант, определяющих пять состояний, в которых может находиться экран. Далее заданы члены класса. Есть камера для отрисовки элементов пользовательского интерфейса, а также вектор, что позволит преобразовывать координаты прикосновения к координатам игрового мира (как и на других экранах, к области видимости в 320×480 единиц, нашему целевому разрешению). В классе есть также `SpriteBatcher`, экземпляры классов `World` и `WorldListener`. Класс `WorldRenderer` мы рассмотрим чуть позже. Он просто принимает экземпляр класса `World` и отрисовывает его. Обратите внимание на то, что его конструктор принимает ссылку на экземпляры классов `SpriteBatcher` и `World` в качестве параметров. Это означает, что для отрисовки элементов пользовательского интерфейса и самого игрового мира используются одинаковые экземпляры класса `SpriteBatcher`. Остальные члены класса — это прямоугольники (объекты класса `Rectangles`) для различных элементов пользовательского интерфейса (например, для пунктов меню `RESUME` (Продолжить) и `QUIT` (Выйти), присутствующих на подэкране паузы) и два члена, которые отслеживают текущий результат игрока. Мы не хотим создавать новую строку каждый кадр, поэтому упростим работу сборщику мусора.

```
public GameScreen(Game game) {
    super(game);
    state = GAME_READY;
    guiCam = new Camera2D(glGraphics, 320, 480);
    touchPoint = new Vector2();
    batcher = new SpriteBatcher(glGraphics, 1000);
    worldListener = new WorldListener() {
        @Override
        public void jump() {
            Assets.playSound(Assets.jumpSound);
        }

        @Override
        public void highJump() {
            Assets.playSound(Assets.highJumpSound);
        }

        @Override
        public void hit() {
```

```
                Assets.playSound(Assets.hitSound);
            }

            @Override
            public void coin() {
                Assets.playSound(Assets.coinSound);
            }
        };
        world = new World(worldListener);
        renderer = new WorldRenderer(glGraphics, batcher, world);
        pauseBounds = new Rectangle(320- 64, 480- 64, 64, 64);
        resumeBounds = new Rectangle(160 – 96, 240, 192, 36);
        quitBounds = new Rectangle(160 – 96, 240 – 36, 192, 36);
        lastScore = 0;
        scoreString = "score: 0";
    }
```

В конструкторе инициализируются все переменные — члены класса. Единственное, что здесь представляет интерес, — реализация `WorldListener` как анонимного внутреннего класса. Он регистрируется экземпляром класса `World` и воспроизводит звуковые эффекты соответственно происходящим событиям.

Обновление игрового экрана

Далее рассмотрим методы обновления, которые позволят убедиться в том, что любой ввод данных пользователем будет корректно обработан, и при необходимости обновят экземпляр класса `World` (листинг 9.20).

Листинг 9.20. Фрагменты класса GameScreen.java: методы обновления игрового экрана

```
@Override
public void update(float deltaTime) {
    if(deltaTime > 0.1f)
        deltaTime = 0.1f;

    switch(state) {
    case GAME_READY:
        updateReady();
        break;
    case GAME_RUNNING:
        updateRunning(deltaTime);
        break;
    case GAME_PAUSED:
        updatePaused();
        break;
    case GAME_LEVEL_END:
        updateLevelEnd();
        break;
    case GAME_OVER:
        updateGameOver();
        break;
    }
}
```

Метод GLScreen.update() также является суперметодом, вызывающим другие методы обновления в зависимости от текущего состояния экрана. Обратите внимание на то, что мы ограничиваем параметр deltaTime значением, равным 0,1 секунды. Зачем это делается? В главе 6 мы говорили об ошибке в прямых ByteBuffers в ОС Android версии 1.5, генерирующей мусор. В нашей игре также может возникнуть эта проблема, если она будет запущена на устройстве с ОС Android 1.5. В любой момент игра может прерваться сборщиком мусора на несколько сотен миллисекунд. Это отразится и на параметре deltaTime, что заставит Боба «телепортировать» из одной точки в другую вместо плавного продвижения. Боб будет проходить сквозь платформы, даже не пересекаясь с ними, поскольку будет перемещаться на большое расстояние за один кадр. Ограничив это время чувствительным максимальным значением, равным 0,1 секунды, мы можем скомпенсировать этот эффект.

```
private void updateReady() {
    if(game.getInput().getTouchEvents().size() > 0) {
        state = GAME_RUNNING;
    }
}
```

Метод updateReady() вызывается на подэкране паузы. Все, что он делает, — ожидает прикосновения к экрану, после чего игровой экран перейдет в состояние GAME_RUNNING.

```
private void updateRunning(float deltaTime) {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    int len = touchEvents.size();
    for(int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);
        if(event.type != TouchEvent.TOUCH_UP)
            continue;

        touchPoint.set(event.x, event.y);
        guiCam.touchToWorld(touchPoint);

        if(OverlapTester.pointInRectangle(pauseBounds, touchPoint)) {
            Assets.playSound(Assets.clickSound);
            state = GAME_PAUSED;
            return;
        }
    }

    world.update(deltaTime, game.getInput().getAccelX());
    if(world.score != lastScore) {
        lastScore = world.score;
        scoreString = "" + lastScore;
    }
    if(world.state == World.WORLD_STATE_NEXT_LEVEL) {
        state = GAME_LEVEL_END;
    }
    if(world.state == World.WORLD_STATE_GAME_OVER) {
        state = GAME_OVER;
```

```
            if(lastScore >= Settings.highscores[4])
                scoreString = "new highscore: " + lastScore;
            else
                scoreString = "score: " + lastScore;
            Settings.addScore(lastScore);
            Settings.save(game.getFileIO());
        }
    }
```

Сначала метод `updateRunning()` проверяет, прикоснулся ли пользователь к кнопке «Пауза» в правом верхнем углу экрана. Если это так, игра переходит в состояние `GAME_PAUSED`. В противном случае экземпляр класса `World` обновляется, используя текущее значение параметра `deltaTime` и показания акселерометра, отвечающие за перемещение Боба по горизонтали. После этого также необходимо проверить, нужно ли обновить строки, содержащие количество очков пользователя. Выполняется также проверка на то, достиг ли Боб замка. В этом случае переходим к состоянию `GAME_NEXT_LEVEL`, в котором на экран будет выведено сообщение, аналогичное показанному на рис. 9.2. Далее экран будет ожидать нажатия экрана, чтобы сгенерировать новый уровень. Если игра закончилась, выводим на экран строку, содержащую количество очков пользователя. Она будет иметь вид или `score: #score`, или `new highscore: #score` в зависимости от того, превосходит ли текущее количество очков результаты, показанные ранее. Далее итоговый результат игрока добавляется в настройки и сохраняется на карте памяти. В дополнение игровой экран переходит в состояние `GAME_OVER`.

```
private void updatePaused() {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    int len = touchEvents.size();
    for(int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);
        if(event.type != TouchEvent.TOUCH_UP)
            continue;

        touchPoint.set(event.x, event.y);
        guiCam.touchToWorld(touchPoint);

        if(OverlapTester.pointInRectangle(resumeBounds, touchPoint)) {
            Assets.playSound(Assets.clickSound);
            state = GAME_RUNNING;
            return;
        }

        if(OverlapTester.pointInRectangle(quitBounds, touchPoint)) {
            Assets.playSound(Assets.clickSound);
            game.setScreen(new MainMenuScreen(game));
            return;

        }
    }
}
```

В методе updatePaused() проверяется, нажал ли пользователь пункты меню RESUME (Продолжить) или QUIT (Выйти), а также выполняется соответствующая реакция.

```
private void updateLevelEnd() {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    int len = touchEvents.size();
    for(int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);
        if(event.type != TouchEvent.TOUCH_UP)
            continue;
        world = new World(worldListener);
        renderer = new WorldRenderer(glGraphics, batcher, world);
        world.score = lastScore;
        state = GAME_READY;
    }
}
```

Метод updateLevelEnd() ожидает прикосновения пользователя к экрану. Если оно произошло, создаются новые экземпляры классов World и WorldRenderer. Кроме того, в этом методе экземпляру класса World указывается количество очков, набранных пользователем на протяжении игры. Игровой экран переходит в состояние GAME_READY, в котором он также ожидает прикосновения игрока к экрану.

```
private void updateGameOver() {
    List<TouchEvent> touchEvents = game.getInput().getTouchEvents();
    int len = touchEvents.size();
    for(int i = 0; i < len; i++) {
        TouchEvent event = touchEvents.get(i);
        if(event.type != TouchEvent.TOUCH_UP)
            continue;
        game.setScreen(new MainMenuScreen(game));
    }
}
```

Метод updateGameOver() также ожидает нажатия экрана. Если оно произошло, мы просто переходим к главному меню игры, что показано на рис. 9.2.

Отрисовка игрового экрана

После всех этих обновлений игровой экран должен отрисовать себя с помощью вызова метода GameScreen.present(). Взглянем на код этого метода, приведенный в листинге 9.21.

Листинг 9.21. Фрагменты класса GameScreen.java: метод отрисовки игрового экрана

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    gl.glEnable(GL10.GL_TEXTURE_2D);

    renderer.render();

    guiCam.setViewportAndMatrices();
```

```
        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
        batcher.beginBatch(Assets.items);
        switch(state) {
        case GAME_READY:
            presentReady();
            break;
        case GAME_RUNNING:
            presentRunning();
            break;
        case GAME_PAUSED:
            presentPaused();
            break;
        case GAME_LEVEL_END:
            presentLevelEnd();
            break;
        case GAME_OVER:
            presentGameOver();
            break;
        }
        batcher.endBatch();
        gl.glDisable(GL10.GL_BLEND);
    }
```

Отрисовка игрового экрана происходит в два этапа. Сначала отрисовывается игровой мир с помощью класса `WorldRenderer`, а затем отрисовываются все элементы пользовательского интерфейса в верхней части игрового мира, основываясь на текущем состоянии игрового экрана. Метод `render()` выполняет как раз эти функции. Как и для методов обновления, у нас есть отдельные методы отрисовки для каждого подэкрана.

```
    private void presentReady() {
        batcher.drawSprite(160, 240, 192, 32, Assets.ready);
    }
```

Метод `presentReady()` отображает кнопку паузы в верхнем правом углу экрана, а также строку, содержащую количество очков, набранное игроком, в левом верхнем углу экрана.

```
    private void presentRunning() {
        batcher.drawSprite(320 - 32, 480 - 32, 64, 64, Assets.pause);
        Assets.font.drawText(batcher, scoreString, 16, 480-20);
    }
```

Метод `presentRunning()` просто отрисовывает кнопку паузы и текущее количество очков игрока.

```
    private void presentPaused() {
        batcher.drawSprite(160, 240, 192, 96, Assets.pauseMenu);
        Assets.font.drawText(batcher, scoreString, 16, 480-20);
    }
```

Метод `presentPaused()` отображает элементы пользовательского интерфейса меню паузы и количество очков игрока.

```
private void presentLevelEnd() {
    String topText = "the princess is ...";
    String bottomText = "in another castle!";
    float topWidth = Assets.font.glyphWidth * topText.length();
    float bottomWidth = Assets.font.glyphWidth * bottomText.length();
    Assets.font.drawText(batcher, topText, 160 – topWidth / 2, 480 – 40);
    Assets.font.drawText(batcher, bottomText, 160 – bottomWidth / 2, 40);
}
```

Метод `presentLevelEnd()` отрисовывает строку `THE PRINCESS IS ...` (Принцесса...) в верхней части экрана и строку `IN ANOTHER CASTLE!` (В другом замке!) в нижней части экрана, как показано на рис. 9.2. Выполняются также некоторые вычисления для того, чтобы выровнять эти строки по центру.

```
private void presentGameOver() {
    batcher.drawSprite(160, 240, 160, 96, Assets.gameOver);
    float scoreWidth = Assets.font.glyphWidth * scoreString.length();
    Assets.font.drawText(batcher, scoreString, 160 – scoreWidth / 2,480-20);
}
```

Метод `presentGameOver()` отображает элементы пользовательского интерфейса, необходимые на экране конца игры, и количество очков игрока. Помните, что экран результатов устанавливается методом `updateRunning()` и его строки имеют вид либо `score: #score`, либо `new highscore: #value`.

Последние штрихи

Класс игрового экрана практически готов. Остальная часть кода приведена в листинге 9.22.

Листинг 9.22. Окончание класса `GameScreen.java`: методы `pause()`, `resume()` и `dispose()`

```
    @Override
    public void pause() {
        if(state == GAME_RUNNING)
            state = GAME_PAUSED;
    }

    @Override
    public void resume() {
    }

    @Override
    public void dispose() {
    }
}
```

Мы просто убеждаемся в том, что игровой экран находится в состоянии паузы, когда пользователь решает приостановить приложение. Последнее, что осталось реализовать, — класс `WorldRenderer`.

Класс WorldRenderer

Этот класс не должен вас удивить. Он просто использует экземпляр класса SpriteBatcher, который передается ему в конструктор для того, чтобы отрисовать игровой мир. В листинге 9.23 приведено начало его кода.

Листинг 9.23. Фрагменты класса WorldRenderer.java: константы, члены класса и конструктор

```
package com.badlogic.androidgames.jumper;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.gl.Animation;
import com.badlogic.androidgames.framework.gl.Camera2D;
import com.badlogic.androidgames.framework.gl.SpriteBatcher;
import com.badlogic.androidgames.framework.gl.TextureRegion;
import com.badlogic.androidgames.framework.impl.GLGraphics;

public class WorldRenderer {
    static final float FRUSTUM_WIDTH = 10;
    static final float FRUSTUM_HEIGHT = 15;
    GLGraphics glGraphics;
    World world;
    Camera2D cam;
    SpriteBatcher batcher;

    public WorldRenderer(GLGraphics glGraphics, SpriteBatcher batcher,
                         World world) {
        this.glGraphics = glGraphics;
        this.world = world;
        this.cam = new Camera2D(glGraphics, FRUSTUM_WIDTH, FRUSTUM_HEIGHT);
        this.batcher = batcher;
    }
```

Как обычно, начинаем с определения некоторых констант. В этом случае это ширина и высота области видимости, значения которых мы определили равными 10 и 15 м. У нас также есть несколько других членов — экземпляры классов GLGraphics, Camera2D, а также ссылка на экземпляр класса SpriteBatcher, которую конструктор получает от класса игрового экрана.

Конструктор в качестве параметров принимает экземпляры классов GLGraphics, SpriteBatcher и World. Следует соответственно инициализировать все члены класса. В листинге 9.24 приведен код отрисовки игрового мира.

Листинг 9.24. Окончание класса WorldRenderer.java: код отрисовки игрового мира

```
    public void render() {
        if(world.bob.position.y > cam.position.y )
            cam.position.y = world.bob.position.y;
        cam.setViewportAndMatrices();
        renderBackground();
        renderObjects();
    }
```

Метод `render()` разбивает отрисовку на два пакета: один для фонового изображения, а другой — для всех объектов игрового мира. Он также обновляет позицию камеры, основываясь на текущей координате Боба по вертикали. Если он находится выше координаты камеры по оси *y*, камера соответственно сдвигается. Обратите внимание на то, что использована камера, которая работает в единицах измерения игрового мира. Мы только единожды устанавливаем матрицы для фонового изображения и для объектов.

```
public void renderBackground() {
    batcher.beginBatch(Assets.background);
    batcher.drawSprite(cam.position.x, cam.position.y,
                       FRUSTUM_WIDTH, FRUSTUM_HEIGHT,
                       Assets.backgroundRegion);
    batcher.endBatch();
}
```

Метод `renderBackground()` просто отрисовывает фоновое изображение таким образом, что оно следует за камерой. Оно не прокручивается, но вместо этого отрисовывается так, чтобы полностью заполнять экран. Мы также не используем никакого смешивания для отрисовки фонового изображения, поэтому мы можем сжать его для того, чтобы получить небольшой прирост производительности.

```
public void renderObjects() {
    GL10 gl = glGraphics.getGL();
    gl.glEnable(GL10.GL_BLEND);
    gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);

    batcher.beginBatch(Assets.items);
    renderBob();
    renderPlatforms();
    renderItems();
    renderSquirrels();
    renderCastle();
    batcher.endBatch();
    gl.glDisable(GL10.GL_BLEND);
}
```

Метод `renderObjects()` отвечает за отрисовку второго пакета. В этот раз мы будем использовать смешивание, поскольку все наши объекты имеют прозрачные фоновые пикселы. Все объекты отрисовываются в одном пакете. Оглядываясь на конструктор класса `GameScreen`, можно увидеть, что экземпляр класса `SpriteBatcher`, который мы используем, может работать с 1000 спрайтов в одном пакете — этого более чем достаточно для игрового мира. Для каждого объекта существует свой метод, отрисовывающий его.

```
private void renderBob() {
    TextureRegion keyFrame;
    switch(world.bob.state) {
    case Bob.BOB_STATE_FALL:
        keyFrame = Assets.bobFall.getKeyFrame(world.bob.stateTime,
Animation.ANIMATION_LOOPING);
        break;
```

```
        case Bob.BOB_STATE_JUMP:
            keyFrame = Assets.bobJump.getKeyFrame(world.bob.stateTime,
Animation.ANIMATION_LOOPING);
            break;
        case Bob.BOB_STATE_HIT:
        default:
            keyFrame = Assets.bobHit;
        }

        float side = world.bob.velocity.x < 0? -1: 1;
       batcher.drawSprite(world.bob.position.x, world.bob.position.y, side * 1, 1,
keyFrame);
    }
```

Метод `renderBob()` отвечает за отрисовку Боба. Основываясь на состоянии Боба и времени состояния, мы выбираем один кадр из пяти (см. рис. 9.8). Основываясь на компоненте скорости Боба по оси x, мы также определяем, в какую сторону будет повернут Боб. Основываясь на этом, мы повернем в соответствующий текстурный регион. Помните, на имеющихся у нас кадрах Боб повернут только вправо. Обратите также внимание на то, что мы не используем параметры `BOB_WIDTH` или `BOB_HEIGHT` для того, чтобы определить размеры прямоугольника, который мы рисуем для Боба. Эти параметры предназначены для ограничивающих фигур, они необязательно пригодятся для прямоугольников, которые мы рисуем сейчас. Вместо этого мы применяем выбранный масштаб — 1×1 м к 32×32 пиксела. Это же мы повторим для всех спрайтов; мы будем использовать прямоугольник 1×1 (Боб, монеты, белки, пружины), прямоугольник $2 \times 0{,}5$ (платформы) или прямоугольник 2×2 (замок).

```
    private void renderPlatforms() {
        int len = world.platforms.size();
        for(int i = 0; i < len; i++) {
            Platform platform = world.platforms.get(i);
            TextureRegion keyFrame = Assets.platform;
            if(platform.state == Platform.PLATFORM_STATE_PULVERIZING) {
                    keyFrame =
Assets.brakingPlatform.getKeyFrame(platform.stateTime,
Animation.ANIMATION_NONLOOPING);
            }

            batcher.drawSprite(platform.position.x, platform.position.y,2,
                            0.5f, keyFrame);
        }
    }
```

Метод `renderPlatforms()` проходит в цикле по всем платформам игрового мира и выбирает текстурный регион, основываясь на состоянии платформы. Платформа может рассыпаться или не рассыпаться. В последнем случае мы просто будем использовать первый кадр, а в первом нам придется получать кадр анимации рассыпания платформы, основываясь на времени состояния платформы.

```
    private void renderItems() {
        int len = world.springs.size();
        for(int i = 0; i < len; i++) {
            Spring spring = world.springs.get(i);
            batcher.drawSprite(spring.position.x, spring.position.y, 1, 1, Assets.
spring);
        }

        len = world.coins.size();
        for(int i = 0; i < len; i++) {
            Coin coin = world.coins.get(i);
             TextureRegion keyFrame = Assets.coinAnim.getKeyFrame(coin.stateTime,
Animation.ANIMATION_LOOPING);
            batcher.drawSprite(coin.position.x, coin.position.y, 1, 1,keyFrame);
        }
    }
```

Метод `renderItems()` отрисовывает пружины и монеты. Для пружин используется один текстурный регион, определенный в активах программы. Для монет мы снова выбираем кадр анимации, основываясь на времени состояния монеты.

```
    private void renderSquirrels() {
        int len = world.squirrels.size();
        for(int i = 0; i < len; i++) {
            Squirrel squirrel = world.squirrels.get(i);
            TextureRegion keyFrame =
Assets.squirrelFly.getKeyFrame(squirrel.stateTime,
Animation.ANIMATION_LOOPING);
            float side = squirrel.velocity.x < 0?-1:1;
          batcher.drawSprite(squirrel.position.x, squirrel.position.y, side * 1, 1,
keyFrame);
        }
    }
```

Метод `renderSquirrels()` отрисовывает белок. Мы снова получаем кадр анимации основываясь на времени состояния белки, определяем направление ее движения, а также управляем ее шириной при отрисовке с помощью `SpriteBatcher`. Это необходимо, поскольку в нашем атласе текстур есть только изображение белки, повернутой влево.

```
    private void renderCastle() {
        Castle castle = world.castle;
        batcher.drawSprite(castle.position.x, castle.position.y, 2, 2,
                           Assets.castle);
    }
}
```

Последний метод называется `renderCastle()`. Он просто отрисовывает замок используя текстурный регион, определенный в классе `Assets`.

Это было довольно просто, не так ли? У нас есть только два пакета для отрисовки: один для фонового изображения и один для всех игровых объектов. Учитывая

все вышеописанное, понятно, что мы отрисовываем третий пакет для всех элементов пользовательского интерфейса. Всего приходится три раза изменять текстуры и три раза загружать новые вершины в GPU. Теоретически мы могли бы объединить пакеты пользовательского интерфейса и игровых объектов, но это было бы обременительно, а код стал бы неэлегантным. В соответствии с нашим руководством по оптимизации, написанном в главе 6, нашему приложению следует иметь мгновенную отрисовку. Проверим, так ли это.

Мы практически закончили. В нашу вторую игру — «Большой прыгун» — уже можно играть.

Оптимизировать или не оптимизировать?

Пришло время протестировать нашу новую игру. Единственное место в ней, где необходима высокая скорость, — это игровой экран. Я просто поместил экземпляр класса `FPSCounter` в класс `GameScreen` и вызывал метод `FPSCounter.logFrame()` в конце работы метода `GameScreen.render()`. Вот результаты для устройств Hero, Droid и Nexus One:

```
Hero (1.5):
01-02 20:58:06.417: DEBUG/FPSCounter(8251): fps: 57
01-02 20:58:07.427: DEBUG/FPSCounter(8251): fps: 57
01-02 20:58:08.447: DEBUG/FPSCounter(8251): fps: 57
01-02 20:58:09.447: DEBUG/FPSCounter(8251): fps: 56

Droid (2.1.1):
01-02 21:03:59.643: DEBUG/FPSCounter(1676): fps: 61
01-02 21:04:00.659: DEBUG/FPSCounter(1676): fps: 59
01-02 21:04:01.659: DEBUG/FPSCounter(1676): fps: 60
01-02 21:04:02.666: DEBUG/FPSCounter(1676): fps: 60

Nexus One (2.2.1):
01-02 20:54:05.263: DEBUG/FPSCounter(1393): fps: 61
01-02 20:54:06.273: DEBUG/FPSCounter(1393): fps: 61
01-02 20:54:07.273: DEBUG/FPSCounter(1393): fps: 60
01-02 20:54:08.283: DEBUG/FPSCounter(1393): fps: 61
```

60 кадров в секунду — это очень хороший результат. Hero несколько отстает, но только лишь из-за того, что его процессор немного слабее. Мы могли бы использовать класс `SpatialHashGrid` для того, чтобы немного ускорить эмуляцию игрового мира, но это упражнение я оставлю вам, дорогой читатель. Нет серьезной необходимости это делать, поскольку Hero всегда будет сталкиваться с проблемами (как и любое устройство с ОС Android 1.5). Гораздо больше неприятностей на этом устройстве доставляет сборка мусора. Мы знаем причину (ошибка в прямых `ByteBuffer`), но ничего не можем с ней поделать. Давайте надеяться, что Android 1.5 довольно скоро канет в Лету.

Предыдущие измерения я выполнял при отключенном в главном меню звуке. Проведем их снова при включенном звуке:

```
Hero (1.5):
01-02 21:01:22.437: DEBUG/FPSCounter(8251): fps: 43
01-02 21:01:23.457: DEBUG/FPSCounter(8251): fps: 48
01-02 21:01:24.467: DEBUG/FPSCounter(8251): fps: 49
01-02 21:01:25.487: DEBUG/FPSCounter(8251): fps: 49

Droid (2.1.1):
01-02 21:10:49.979: DEBUG/FPSCounter(1676): fps: 54
01-02 21:10:50.979: DEBUG/FPSCounter(1676): fps: 56
01-02 21:10:51.987: DEBUG/FPSCounter(1676): fps: 54
01-02 21:10:52.987: DEBUG/FPSCounter(1676): fps: 56

Nexus One (2.2.1):
01-02 21:06:06.144: DEBUG/FPSCounter(1470): fps: 61
01-02 21:06:07.153: DEBUG/FPSCounter(1470): fps: 61
01-02 21:06:08.173: DEBUG/FPSCounter(1470): fps: 62
01-02 21:06:09.183: DEBUG/FPSCounter(1470): fps: 61
```

Ох. У Hero при включенной фоновой музыке значительно падает производительность. Воспроизведение аудио также оказывает влияние и на Droid. А вот Nexus One не заметил разницы. Что мы можем с этим поделать? Вообще-то ничего. Основная проблема появляется не из-за звуков, а из-за фоновой музыки. Разбиение на потоки и декодирование MP3- или OGG-файла затрачивает циклы процессора; так работает наш мир. Просто учитывайте этот фактор при проведении измерений производительности.

Подводя итог

Мы создали вторую игру используя мощь OpenGL ES. Благодаря хорошему фреймворку реализовать ее было довольно просто. Использование атласа текстур и класса SpriteBatcher дало хорошие результаты при тесте производительности. Мы также обсудили, как отрисовать моноширинный растровый ASCII-шрифт. Хороший исходный дизайн наших игровых механик, а также прозрачное определение соотношения между единицами измерения игрового мира и пикселами значительно упростили создание игры. Представьте, как ужасно было бы проводить все расчеты в пикселах. Все вычисления были бы переполнены операциями деления — операциями, которые совсем не любят маломощные процессоры некоторых Android-устройств. Мы также проделали серьезную работу по разделению логики игры от ее внешнего вида. В итоге я бы назвал игру «Большой прыгун» успешной.

Теперь перейдем к главе 10 и рассмотрим некоторые вопросы, связанные с трехмерным программированием.

10 OpenGL ES: займемся 3D

Для написания игры «Большой прыгун» мы использовали двухмерный движок отрисовки OpenGL ES. Теперь пришло время полностью перейти к трем измерениям. На самом деле мы уже работали в трехмерном пространстве, когда определяли область видимости и вершины спрайтов. В последнем случае координата по оси *z* каждой вершины просто устанавливалась равной нулю по умолчанию. Разница между трехмерной и двухмерной отрисовками не такая уж и большая. Для трехмерного пространства верны следующие моменты.

- Вершины имеют координаты не только по осям *x* и *y*, но и по оси *z*.
- Вместо ортографической проекции мы будем использовать перспективную проекцию. Объекты, находящиеся далеко от камеры, будут казаться меньше.
- Преобразования, такие как поворот изображения, параллельный перенос и масштабирование, в трехмерном пространстве имеют больше степеней свободы. Вместо того, чтобы перемещать вершины на двухмерной плоскости, теперь можно свободно передвигать их по всем трем осям.
- Мы можем определить камеру с произвольной позицией и ориентацией в трехмерном пространстве.
- Порядок, в котором будут отрисовываться треугольники объектов, теперь очень важен. Объекты, располагающиеся дальше от камеры, будут перекрываться объектами, которые находятся ближе к камере.

Положительный момент заключается в том, что мы уже переложили грубую работу, касающуюся всех этих аспектов, на фреймворк. Следует лишь немного расширить несколько классов для того, чтобы полностью перейти в три измерения.

Перед стартом

Как обычно, в этой главе создадим несколько примеров. Для этого мы по образцу предыдущих глав опишем примеры, которые выполнять. Повторно используем весь фреймворк, созданный нами в предыдущих главах, включая классы `GLGame`, `GLScreen`, `Texture` и `Vertices`.

Первый пример этой главы называется GL3DBasicsStarter. Мы можем заново использовать код класса GLBasicsStarter из главы 6 и просто изменить имена пакетов классов-примеров, которые мы собираемся запустить, на com.badlogic.androidgames.gl3d. Следует также добавить в манифест каждый тест в виде элементов <activity>. Все тесты будут запускаться строго в альбомной ориентации, что определено в элементах <activity>.

Каждый тест — это экземпляр абстрактного класса GLGame, логика самого теста реализуется в виде экземпляра класса GLScreen, являющегося членом реализации GLGame для конкретного теста, как и в предыдущих главах. Я буду представлять для рассмотрения лишь самые важные фрагменты класса GLScreen, чтобы сэкономить немного места. Снова примем то же соглашение для имен — XXXTest и XXXScreen для реализаций классов GLGame и GLScreen для каждого теста.

Вершины в 3D

Из главы 7 вы узнали, что у вершины есть несколько атрибутов:

- позиция;
- цвет (необязательно);
- координаты текстуры (необязательно).

Мы создали вспомогательный класс Vertices, делающий за нас всю грязную работу. Тогда мы ограничили позиции вершины координатами по осям x и y. Все, что нужно для перехода в 3D, — модифицировать класс Vertices так, чтобы он поддерживал трехмерные позиции вершин.

Vertices3: сохраняем 3D-координаты

Напишем новый класс, называющийся Vertices3, чтобы работать с трехмерными координатами. Он будет основан на оригинальном классе Vertices. В листинге 10.1 показан его код.

Листинг 10.1. Класс Vertices3.java — координат стало больше!

```
package com.badlogic.androidgames.framework.gl;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.IntBuffer;
import java.nio.ShortBuffer;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.impl.GLGraphics;

public class Vertices3 {
```

```
final GLGraphics glGraphics;
final boolean hasColor;
final boolean hasTexCoords;
final int vertexSize;
final IntBuffer vertices;
final int[] tmpBuffer;
final ShortBuffer indices;

public Vertices3(GLGraphics glGraphics, int maxVertices, int maxIndices,
        boolean hasColor, boolean hasTexCoords) {
    this.glGraphics = glGraphics;
    this.hasColor = hasColor;
    this.hasTexCoords = hasTexCoords;
    this.vertexSize = (3 + (hasColor ? 4 : 0) + (hasTexCoords ? 2 : 0)) * 4;
    this.tmpBuffer = new int[maxVertices * vertexSize / 4];

    ByteBuffer buffer = ByteBuffer.allocateDirect(maxVertices *
                                                  vertexSize);
    buffer.order(ByteOrder.nativeOrder());
    vertices = buffer.asIntBuffer();

    if (maxIndices > 0) {
        buffer = ByteBuffer.allocateDirect(maxIndices * Short.SIZE / 8);
        buffer.order(ByteOrder.nativeOrder());
        indices = buffer.asShortBuffer();
    } else {
        indices = null;
    }
}

public void setVertices(float[] vertices, int offset, int length) {
    this.vertices.clear();
    int len = offset + length;
    for (int i = offset, j = 0; i < len; i++, j++)
        tmpBuffer[j] = Float.floatToRawIntBits(vertices[i]);
    this.vertices.put(tmpBuffer, 0, length);
    this.vertices.flip();
}

public void setIndices(short[] indices, int offset, int length) {
    this.indices.clear();
    this.indices.put(indices, offset, length);
    this.indices.flip();
}

public void bind() {
    GL10 gl = glGraphics.getGL();

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    vertices.position(0);
```

```
        gl.glVertexPointer(3, GL10.GL_FLOAT, vertexSize, vertices);

        if (hasColor) {
            gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
            vertices.position(3);
            gl.glColorPointer(4, GL10.GL_FLOAT, vertexSize, vertices);
        }

        if (hasTexCoords) {
            gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
            vertices.position(hasColor ? 7 : 3);
            gl.glTexCoordPointer(2, GL10.GL_FLOAT, vertexSize, vertices);
        }
    }

    public void draw(int primitiveType, int offset, int numVertices) {
        GL10 gl = glGraphics.getGL();

        if (indices != null) {
            indices.position(offset);
            gl.glDrawElements(primitiveType, numVertices,
                    GL10.GL_UNSIGNED_SHORT, indices);
        } else {
            gl.glDrawArrays(primitiveType, offset, numVertices);
        }
    }

    public void unbind() {
        GL10 gl = glGraphics.getGL();
        if (hasTexCoords)
            gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

        if (hasColor)
            gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
    }
}
```

Здесь практически ничего не изменилось по сравнению с классом `Vertices`, за исключением четырех небольших фрагментов. В конструкторе значение `vertexSize` теперь рассчитывается по-другому, поскольку позиция вершины имеет три координаты, а не две. В методе `bind()` мы с помощью вызова метода `glVertexPointer()` (первый аргумент) сообщаем OpenGL ES, что вершины имеют три координаты, а не две. Мы также увеличиваем смещение, значение которого устанавливается в вызове метода `vertices.position()`, для необязательных компонент, таких как цвет и координаты текстуры.

Это все, что нам нужно сделать. Использование класса `Vertices3` позволяет определять *x*-, *y*- и *z*-координаты каждой вершины при вызове метода `Vertices3.setVertices()`. Все остальное остается точно таким же. У нас могут быть заданы цвета для каждой вершины, а также координаты текстур, их индексы и т. д.

Пример

Напишем простой пример, который будет называться `Vertices3Test`. Мы хотим нарисовать два треугольника, у одного координата *z* каждой вершины будет равна –3, а у другого — –5. Мы также определим цвет каждой вершины. Поскольку мы еще не обсуждали использование перспективной проекции, будем применять ортографическую проекцию с такими ближней и дальней плоскостями отсечения, что треугольники будут находиться в области видимости (например, ближняя равна 10, а дальняя — –10). Сцена показана на рис. 10.1.

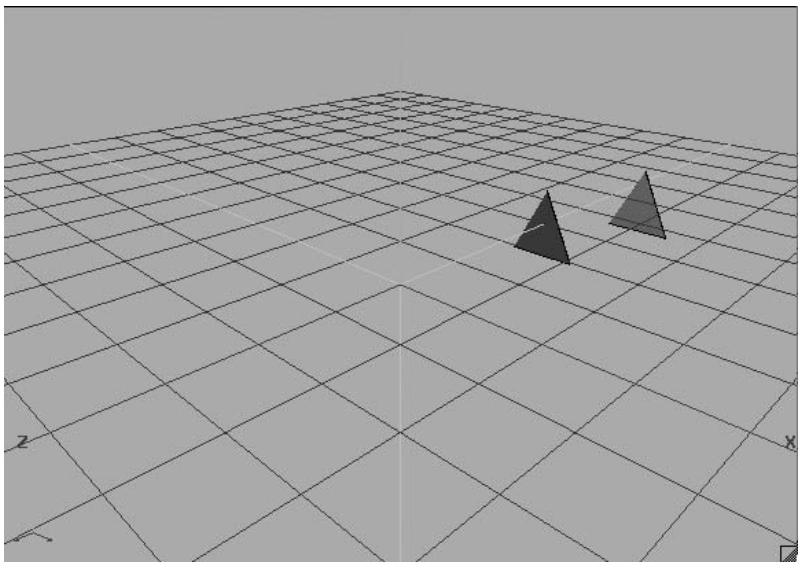

Рис. 10.1. Красный треугольник *(спереди)* и зеленый треугольник *(сзади)* в трехмерном пространстве

Красный треугольник располагается перед зеленым. Мы можем употребить слово «перед», поскольку камера размещается в начале координат и в OpenGL ES по умолчанию направлена вдоль отрицательной части оси *z* (этой информации нет в справочной статье о камере). Зеленый треугольник также немного смещен вправо таким образом, чтобы мы могли видеть его фрагмент, когда камера «смотрит» на сцену спереди. Большая его часть должна перекрываться красным треугольником. В листинге 10.2 приведен код отрисовки этой сцены.

Листинг 10.2. `Vertices3Test.java: рисуем два треугольника`

```
package com.badlogic.androidgames.gl3d;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Screen;
import com.badlogic.androidgames.framework.gl.Vertices3;
```

```
import com.badlogic.androidgames.framework.impl.GLGame;
import com.badlogic.androidgames.framework.impl.GLScreen;

public class Vertices3Test extends GLGame {

    @Override
    public Screen getStartScreen() {
        return new Vertices3Screen(this);
    }

    class Vertices3Screen extends GLScreen {
        Vertices3 vertices;

        public Vertices3Screen(Game game) {
            super(game);

            vertices = new Vertices3(glGraphics, 6, 0, true, false);
            vertices.setVertices(new float[] { -0.5f, -0.5f, -3, 1, 0, 0, 1,
                                    0.5f, -0.5f, -3, 1, 0, 0, 1,
                                    0.0f,  0.5f, -3, 1, 0, 0, 1,

                                    0.0f,  -0.5f, -5, 0, 1, 0, 1,
                                    1.0f,  -0.5f, -5, 0, 1, 0, 1,
                                    0.5f,  0.5f, -5, 0, 1, 0, 1}, 0, 7 * 6);
        }

        @Override
        public void present(float deltaTime) {
            GL10 gl = glGraphics.getGL();
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
            gl.glViewport(0, 0, glGraphics.getWidth(),
                          glGraphics.getHeight());
            gl.glMatrixMode(GL10.GL_PROJECTION);
            gl.glLoadIdentity();
            gl.glOrthof(-1, 1, -1, 1, 10, -10);
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            gl.glLoadIdentity();
            vertices.bind();
            vertices.draw(GL10.GL_TRIANGLES, 0, 6);
            vertices.unbind();
        }

        @Override
        public void update(float deltaTime) {
        }

        @Override
        public void pause() {
        }

        @Override
```

```
        public void resume() {
        }

        @Override
        public void dispose() {
        }
    }
}
```

Как вы можете заметить, здесь приведен весь исходный код. При рассмотрении следующих примеров будут показаны лишь самые значимые фрагменты, поскольку большая их часть будет оставаться такой же, за исключением имен классов.

В классе `Vertices3Screen` присутствует член `Vertices3`, который инициализируется в конструкторе. Всего есть шесть вершин, каждая из которых имеет цвет и не имеет текстурных координат. Поскольку ни один треугольник не имеет общих вершин с другим, мы не будем использовать индексированные геометрии. Эта информация передается конструктору `Vertices3`. Далее устанавливаем координаты вершин с помощью вызова метода `Vertices3.setVertices()`. Первые три строки определяют красный треугольник, расположенный спереди, а последние три строки описывают зеленый треугольник, расположенный сзади и смещенный вправо на 0,5 единицы. Третий параметр в каждой строке — это координата соответствующей вершины по оси *z*.

В методе `present()`, как и обычно, очищаем экран и устанавливаем область видимости. Далее загружаем матрицу ортографической проекции, устанавливаем обозреваемую область достаточно большой для того, чтобы вся сцена поместилась на экран. Наконец, просто отрисовываем два треугольника, находящихся внутри экземпляра класса `Vertices3`. На рис. 10.2 показан результат работы этой программы.

Рис. 10.2. Два странных треугольника

Это выглядит странно. Согласно нашей теории, красный треугольник (посередине) должен находиться перед зеленым треугольником. Камера располагается в начале координат и направлена вдоль отрицательной половины оси *z*. Из рис. 10.1 мы можем увидеть, что красный треугольник находится ближе к началу координат, чем зеленый. Что же здесь произошло?

OpenGL ES будет отрисовывать треугольники в том порядке, в котором они определены в экземпляре класса `Vertices3`. Поскольку первым определен красный треугольник, он и будет нарисован первым. Чтобы это исправить, можно изменить порядок треугольников. Но что если камера будет располалагться, например, сзади? Нам снова придется сортировать треугольники перед их отрисовкой, поэтому такое решение не подходит. Это можно легко исправить. Для начала избавимся от ортографической проекции и будем использовать перспективную.

Перспективная проекция: ближе, больше

До этого момента мы использовали ортографическую проекцию, имея в виду, что независимо от того, как далеко располагается объект от ближней плоскости отсечения, он всегда будет иметь одинаковый размер на экране. Однако наши глаза показывают нам другую картину мира. Чем дальше находится объект, тем меньшим он нам кажется. Это называется перспективной проекцией, мы уже немного говорили о ней в главе 4.

Разницу между ортографической и перспективной проекциями можно объяснить, используя форму области видимости. При ортографической проекции она представляет собой прямоугольник. При перспективной проекции она имеет форму пирамиды, усеченной у ближней плоскости отсечения; основание пирамиды — это дальняя плоскость отсечения; ее грани — это левая, правая, верхняя и нижняя плоскости отсечения. На рис. 10.3 показана перспективная область видимости, сквозь которую мы можем просматривать сцену.

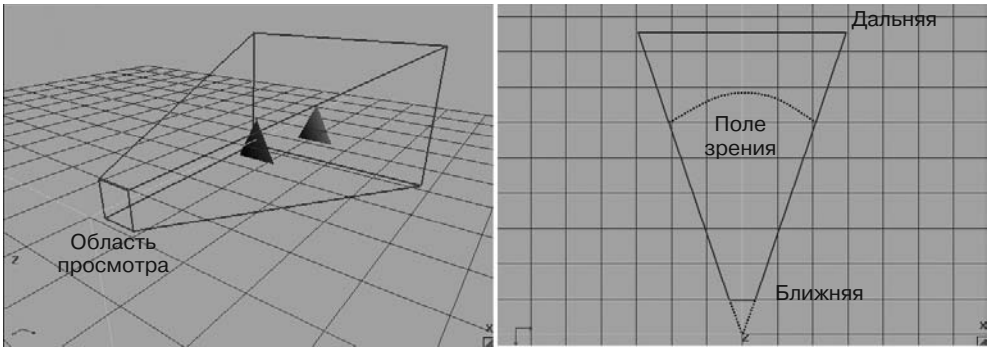

Рис. 10.3. Перспективная область видимости, содержащая сцену *(слева)*, взгляд сверху на область видимости *(справа)*

Перспективная область видимости определяется четырьмя параметрами:

- расстояние от камеры до ближней плоскости отсечения;
- расстояние от камеры до дальней плоскости отсечения;
- соотношение сторон окна просмотра определяется как ширина окна просмотра, деленная на его высоту, и используется для ближней плоскости отсечения;
- поле зрения, которое основано на ширине области видимости и, следовательно, на том, какую часть сцены она показывает.

Мы пока не рассматривали ни одной концепции, связанной с камерой. Мы просто считаем, что она неподвижна, находится в начале координат и направлена вдоль оси *z* в сторону отрицательных значений координат, что показано на рис. 10.3.

Расстояния до ближней и дальней плоскостей отсечения хорошо вам знакомы. Нам просто необходимо установить их значения так, чтобы в области видимости отображалась вся сцена. Что такое поле зрения, также легко понять, посмотрев на правое изображение рис. 10.3.

Понятие «соотношение сторон окна просмотра» немного сложнее. Зачем нужен этот параметр? Он позволяет убедиться, что наш мир не растянется, если соотношение сторон экрана, который необходимо отрисовать, не будет равно 1.

Ранее мы использовали метод `glOrthof()`, чтобы определить ортографическую область видимости в виде матрицы проекций. Для перспективной области видимости можем применить метод `glFrustumf()`. Однако есть и более простой способ.

Обычно OpenGL поставляется со вспомогательной библиотекой, которая называется `GLU`. Она содержит несколько вспомогательных функций для таких целей, как установка значений матрицы проекций и реализация систем камер. Эта библиотека также доступна на платформе Android, где она имеет вид класса, который тоже называется `GLU`. Она предоставляет несколько статических методов, которые можно вызвать без экземпляра класса `GLU`. Метод, который нас интересует в данный момент, называется `gluPerspective()`:

```
GLU.gluPerspective(GL10 gl, float fieldOfView, float aspectRatio, float near,
float far);
```

Этот метод перемножает текущую активную матрицу (например, проекционную или модельно-видовую) с матрицей перспективной проекции, как и метод `glOrthof()`. Первый параметр — это экземпляр класса `GL10`, обычно тот, который мы используем для всех остальных задач, связанных с OpenGL ES. Второй параметр — это поле зрения, заданное в углах, третий параметр — соотношение сторон окна просмотра, а последние два параметра определяют расстояние между позицией камеры и ближней, а также дальней областями отсечения. Поскольку у нас пока нет камеры, эти значения задаются относительно начала координат мира, заставляя нас смотреть

в сторону отрицательных значений координат оси *z*, как и на рис. 10.3. Это замечательно подходит для наших целей в данный момент. Мы убедимся, что все объекты, которые мы отрисовываем, останутся внутри фиксированной и неподвижной области видимости. Пока мы используем только метод `gluPerspective()`, мы не можем изменить позицию или ориентацию нашей виртуальной камеры. Мы всегда будем видеть только ту часть мира, которая лежит вдоль отрицательной половины оси *z*.

Модифицируем последний пример так, чтобы в нем была использована перспективная проекция. Я просто скопировал весь код из класса `Vertices3Test` в новый класс, который называется `PerspectiveTest`, а также переименовал `Vertices3Screen` в `PerspectiveScreen`. Единственное, что необходимо изменить, — метод `present()`. В листинге 10.3 показан новый код этого метода.

Листинг 10.3. Фрагменты класса PerspectiveTest.java: перспективная проекция

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    GLU.gluPerspective(gl, 67,
                   glGraphics.getWidth() / (float)glGraphics.getHeight(),
                   0.1f, 10f);
    gl.glMatrixMode(GL10.GL_MODELVIEW);
    gl.glLoadIdentity();
    vertices.bind();
    vertices.draw(GL10.GL_TRIANGLES, 0, 6);
    vertices.unbind();
}
```

Единственное отличие от предыдущего метода `present()` заключается в том, что теперь мы используем метод `GLU.gluPerspective()` вместо `glOrtho()`. Мы применяем поле зрения, равное 67°, это значение близко к полю зрения человеческого глаза. Увеличивая или уменьшая это значение, вы сможете отобразить больше объектов слева и справа. Следующее, что мы определяем, — соотношение сторон, которое равно отношению ширины экрана к его высоте. Обратите внимание на то, что оно является числом с плавающей точкой, поэтому перед делением нам придется преобразовать ширину в тип `float`. Последние аргументы — это расстояние между камерой и плоскостями отсечения. Полагая, что наша виртуальная камера находится в начале координат и направлена в сторону отрицательных значений по оси *z*, все объекты, имеющие координату по оси *z*, меньшую –0,1, но большую –10, будут находиться между ближней и дальней плоскостями отсечения и поэтому потенциально являются видимыми. На рис. 10.4 показан результат работы этого примера.

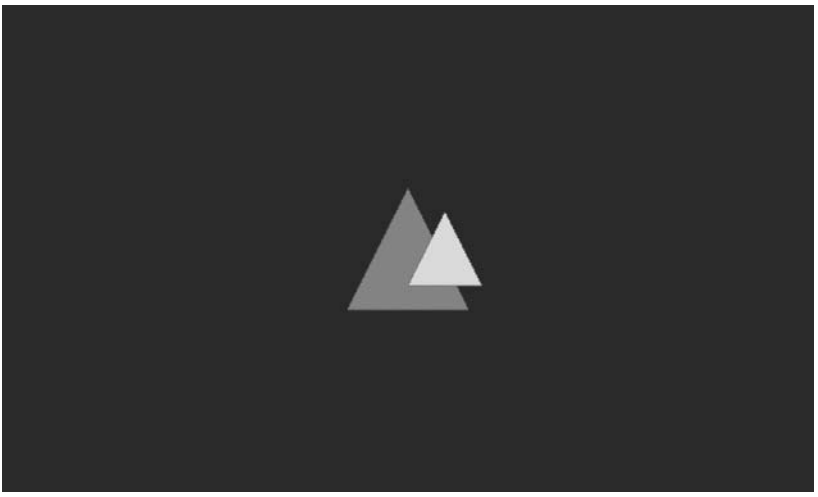

Рис. 10.4. Перспектива (в основном все верно)

Теперь мы на самом деле занимаемся 3D-графикой. Как вы можете видеть, у нас все еще есть проблема с порядком отрисовки треугольников. Исправим это с помощью всемогущего *z*-буфера.

Z-буфер: наводим порядок

Что такое *z*-буфер? В главе 4 мы рассмотрели кадровый буфер. Он хранит цвет каждого пиксела на экране. Когда OpenGL ES отрисовывает треугольник в кадровый буфер, он просто меняет цвет пикселов, составляющих этот треугольник. *Z*-буфер очень похож на кадровый буфер в том, что он также имеет хранилище для каждого пиксела на экране. Однако вместо того, чтобы хранить цвета, он хранит значения глубины. Значение глубины пиксела — это приблизительно нормализованное расстояние от соответствующей точки в 3D до ближней плоскости отсечения области видимости.

OpenGL ES запишет значение глубины для каждого пиксела треугольника в *z*-буфер по умолчанию (если *z*-буфер был создан вместе с кадровым буфером). Нам остается приказать OpenGL ES использовать эту информацию для того, чтобы решить, будет ли отрисовываемый пиксел ближе к ближней плоскости отсечения, чем тот, который уже присутствует. Для этого следует лишь вызвать метод `glEnable()`, передав ему соответствующий параметр:

```
GL10.glEnable(GL10.GL_DEPTH_TEST);
```

Это все, что необходимо сделать. Далее OpenGL ES сравнит значение глубины входящего пиксела со значением глубины пиксела, уже находящегося в *z*-буфере. Если оно меньше, то пиксел будет ближе к ближней плоскости отсечения, поэтому его следует отрисовывать перед тем пикселом, который уже находится в кадровом и *z*-буфере.

Процесс проиллюстрирован на рис. 10.5. В начале работы программы значения, помещенные в *z*-буфер, равны бесконечности (или очень большому числу). Когда отрисовывается первый треугольник, мы сравниваем значение глубины каждого его пиксела со значениями глубины каждого соответствующего пиксела в *z*-буфере. Если значение глубины пиксела меньше, чем значение, находящееся в *z*-буфере, пиксел проходит так называемый тест глубины, или *z*-тест. Его цвет запишется в кадровый буфер, а значение его глубины перезапишет соответствующее текущее значение, находящееся в *z*-буфере. Если же пиксел не проходит тест, ни его цвет, ни значение глубины не будут записаны в буферы. Это показано на рис. 10.5 для места, где отрисовывается второй треугольник. Некоторые пикселы имеют меньшие значения глубины, поэтому отрисовываются именно они; прочие пикселы не проходят тест.

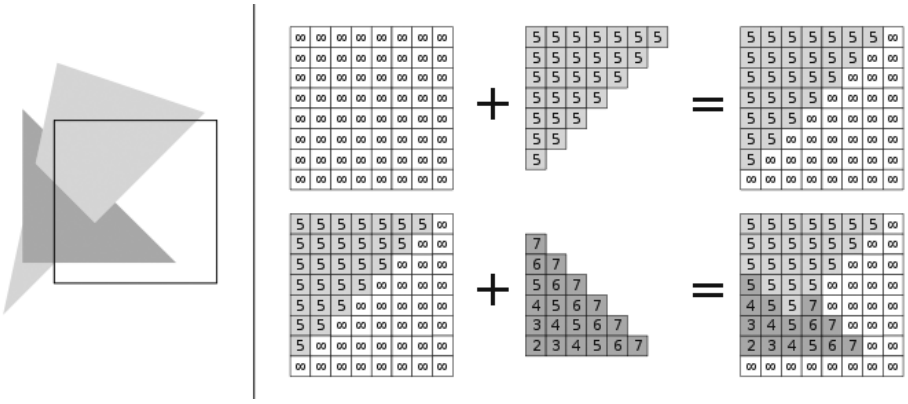

Рис. 10.5. Изображение в кадровом буфере *(слева)*; содержимое *z*-буфера после отрисовки каждого треугольника *(справа)*

Как и в случае с кадровым буфером, нам приходится очищать *z*-буфер каждый кадр, в противном случае значения глубины пикселов последнего кадра все еще будут находиться там. Чтобы сделать это, мы можем вызвать метод `glClear()` следующим образом:

```
gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
```

Это очистит кадровый буфер (или буфер цветов), как и *z*-буфер (или буфер глубины), оба за один раз.

Исправляем последний пример

Решим проблемы последнего примера с использованием *z*-буфера. Я просто скопировал весь код в новый класс, который называется `ZBufferTest`, и изменил метод `present()` нового класса `ZBufferScreen` так, как показано в листинге 10.4.

Листинг 10.4. Фрагменты класса ZBufferTest.java: использование z-буфера

```
@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
```

```
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        GLU.gluPerspective(gl, 67,
                glGraphics.getWidth() / (float)glGraphics.getHeight(),
                0.1f, 10f);
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();

        gl.glEnable(GL10.GL_DEPTH_TEST);

        vertices.bind();
        vertices.draw(GL10.GL_TRIANGLES, 0, 6);
        vertices.unbind();

        gl.glDisable(GL10.GL_DEPTH_TEST);
    }
```

Первое, что мы изменили, — аргументы, передаваемые методу `glClear()`. Теперь мы очищаем оба буфера вместо одного кадрового буфера. Мы также разрешаем тестирование глубины перед тем, как отрисуем два треугольника. После того как отрисовка заканчивается, мы запрещаем тестирование. Почему? Представьте, что мы хотим отрисовать двухмерные элементы пользовательского интерфейса в верхней части нашей 3D-сцены, например текущий счет или кнопки. Поскольку для этого необходимо использовать класс `SpriteBatcher`, который работает только в двух измерениях, нам не нужно иметь никаких осмысленных координат по оси *z* для вершин 2D-элементов. Нам также не нужно проводить тестирование глубины, поскольку мы будем явно задавать порядок отрисовки вершин на экране. Результат работы этого примера показан на рис. 10.6. Теперь он выглядит так, как мы и ожидали.

Рис. 10.6. *Z*-буфер в действии, теперь нет необходимости задавать порядок отрисовки

Наконец-то зеленый треугольник посередине корректно отрисовывается позади красного треугольника. Это происходит благодаря нашему новому лучшему другу — *z*-буферу. Но, как и в случае со многими другими друзьями, наступают времена, когда дружба несколько страдает от некоторых недостатков. Узнаем, какие подводные камни вас могут ожидать, если вы будете использовать *z*-буфер.

Смешиваем: за вами ничего нет

Предположим, что мы хотим разрешить смешивание для красного треугольника, имеющего координату $z = -3$. Установим, скажем, альфа-компонент цвета каждой вершины равным 0,5f, что заставит этот треугольник просвечиваться. В нашем случае должен быть виден зеленый треугольник, имеющий координату *z*, равную –5. Подумаем, что же будет делать OpenGL ES, а также о том, что еще может произойти.

OpenGL ES отрисует первый треугольник в *z*-буфер и буфер цвета.

- Далее OpenGL ES отрисует зеленый треугольник, поскольку он следует после красного треугольника в экземпляре класса `Vertices3`.
- Часть зеленого треугольника, находящаяся позади красного, не будет показана на экране, поскольку ее пикселы не пройдут тест глубины.
- Ничто не будет просвечиваться сквозь красный треугольник, поскольку через него ничто не просвечивалось в тот момент, когда он отрисовывался.

Когда мы используем смешивание в комбинации с *z*-буфером, нам следует убедиться в том, что все прозрачные объекты упорядочены по возрастанию расстояния от камеры, и отрисовывать их от последнего к первому. Все непрозрачные объекты должны быть отрисованы перед прозрачными. Их не обязательно сортировать.

Напишем простой пример, демонстрирующий это. Мы сохраним нашу текущую сцену, состоящую из двух треугольников, и установим альфа-компонент цветов вершин первого треугольника ($z = -3$) равным 0,5f. В соответствии с правилом сначала нам необходимо отрисовать непрозрачные объекты, в нашем случае зеленый треугольник ($z = -5$), а затем — все прозрачные, от дальнего к ближнему. В нашей сцене присутствует только один прозрачный объект: красный треугольник.

Мы скопируем весь код предыдущего примера в новый класс, который называется `ZBlendingTest`, и переименуем `ZBufferScreen`, находящийся в нем, в `ZBlendingScreen`. Нам нужно изменить цвета вершин первого треугольника, а также разрешить смешивание и отрисовку двух треугольников в заданном порядке в методе `present()`. В листинге 10.5 показан измененный код.

Листинг 10.5. Фрагменты класса ZBlendingTest.java: смешивание при доступном z-буфере

```
public ZBlendingScreen(Game game) {
    super(game);

    vertices = new Vertices3(glGraphics, 6, 0, true, false);
    vertices.setVertices(new float[] { -0.5f, -0.5f, -3, 1, 0, 0, 0.5f,
```

```
                                        0.5f, -0.5f, -3, 1, 0, 0, 0.5f,
                                        0.0f,  0.5f, -3, 1, 0, 0, 0.5f,
                                        0.0f,  -0.5f, -5, 0, 1, 0, 1,
                                        1.0f,  -0.5f, -5, 0, 1, 0, 1,
                                        0.5f,  0.5f, -5, 0, 1, 0, 1},
                                                              0, 7 * 6);
}

@Override
public void present(float deltaTime) {
    GL10 gl = glGraphics.getGL();
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
    gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    GLU.gluPerspective(gl, 67,
            glGraphics.getWidth() / (float)glGraphics.getHeight(),
            0.1f, 10f);
    gl.glMatrixMode(GL10.GL_MODELVIEW);
    gl.glLoadIdentity();

    gl.glEnable(GL10.GL_DEPTH_TEST);
    gl.glEnable(GL10.GL_BLEND);
    gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);

    vertices.bind();
    vertices.draw(GL10.GL_TRIANGLES, 3, 3);
    vertices.draw(GL10.GL_TRIANGLES, 0, 3);
    vertices.unbind();

    gl.glDisable(GL10.GL_BLEND);
    gl.glDisable(GL10.GL_DEPTH_TEST);
}
```

В конструкторе класса `ZBlendingScreen` изменим лишь альфа-компоненты цветов вершин первого треугольника, теперь они будут равны 0,5. Это сделает первый треугольник прозрачным. В методе `present()` выполняются привычные операции вроде очистки буферов и установки значений матриц. Мы также разрешаем смешивание и устанавливаем подходящую для этого функцию. Более интересно то, как именно мы будем отрисовывать два треугольника. Сначала отрисуем зеленый треугольник, который является вторым в экземпляре класса `Vertices3`. Он является непрозрачным. Все непрозрачные объекты должны быть отрисованы перед прозрачными. Далее отрисовываем прозрачный треугольник, который является первым в экземпляре класса `Vertices3`. Для обоих вызовов метода отрисовки `vertices.draw()`просто используем подходящее смещение и счетчики вершин, передавая их как второй и третий параметры. На рис. 10.7 показан результат работы этой программы.

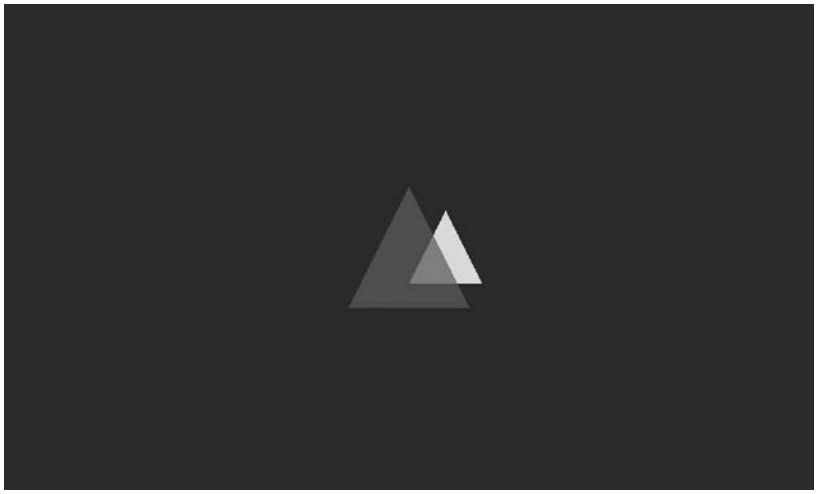

Рис. 10.7. Смешивание при включенном *z*-буфере

Изменим порядок, в котором будут отрисовываться два треугольника, следующим образом:

```
vertices.draw(GL10.GL_TRIANGLES, 0, 3);
vertices.draw(GL10.GL_TRIANGLES, 3, 3);
```

Рисуем первый треугольник, начиная с вершины 0, а затем второй, начиная с вершины 3. В результате сначала будет отрисован красный треугольник спереди, а затем зеленый треугольник сзади. На рис. 10.8 показан результат работы такой программы.

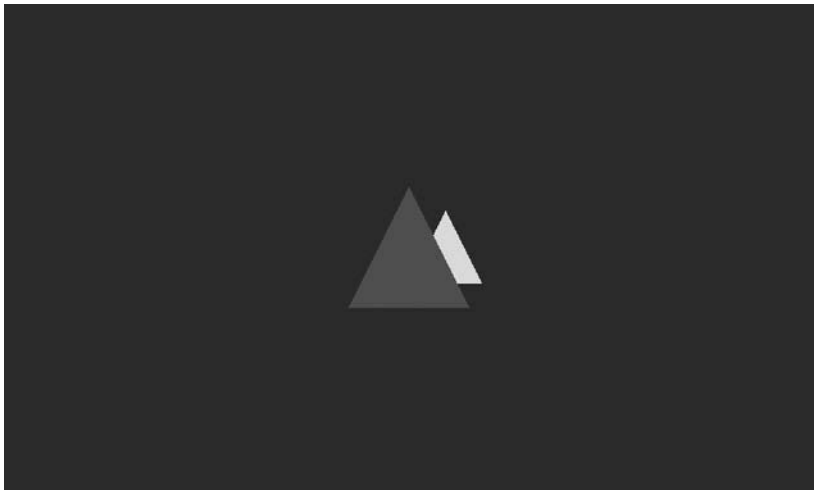

Рис. 10.8. Смешивание выполнено неверно; дальний треугольник должен быть виден сквозь ближний

Все наши объекты являются треугольниками, что делает примеры несколько примитивными. Мы вернемся к рассмотрению связки смешения и *z*-буфера, когда будем отрисовывать более сложные фигуры. Теперь подытожим все рассмотренное в виде последовательности действий.

1. Отрисовка непрозрачных объектов.
2. Сортировка всех прозрачных объектов от самого далекого от камеры до самого близкого.
3. Отрисовка всех прозрачных объектов в отсортированном порядке от самого далекого до самого близкого.

Сортировка может быть основана на расстоянии от центра объекта до камеры (так происходит в большинстве случаев). Вы столкнетесь с проблемами, если один из ваших объектов велик и может перекрывать несколько других объектов. Без довольно сложных трюков мы не сможем разобраться с этой проблемой. Существует несколько пуленепробиваемых решений, которые отлично сработают на обычных персональных компьютерах, но не могут быть реализованы на устройствах с ОС Android из-за того, что их функциональность GPU ограничена. К счастью, такая проблема довольно редка, и в большинстве случаев можно обойтись простой сортировкой.

Точность z-буфера и z-схватка

Всегда есть желание злоупотребить использованием ближней и дальней плоскостей отсечения так, чтобы они показывали как можно больше нашей замечательной сцены. Мы приложили большие усилия для того, чтобы создать множество объектов для игрового мира, и хотели бы, чтобы от них была польза. Единственная проблема заключается в том, что *z*-буфер имеет ограниченную точность. На большинстве устройств с ОС Android каждое значение глубины, хранящееся в *z*-буфере, может иметь размер не более 16 бит, в общей сложности получается 65 535 различных значений глубины. Поэтому вместо того, чтобы установить значения ближней и дальней плоскостей отсечения равными 0,00001 и 1 000 000 соответственно, выберите более обдуманные значения. В противном случае вы увидите артефакты, которые будет производить неверно сконфигурированная область видимости в комбинации с *z*-буфером.

В чем заключается проблема? Представьте, что мы установили значения ближней и дальней плоскостей отсечения такими, какими мы только что обозначили. Значение глубины пиксела — это расстояние от ближней плоскости отсечения, чем пиксел к ней ближе, тем меньше значение глубины. При величине буфера, равной 16 бит, мы делим расстояние от ближней до дальней плоскости отсечения на 65 535 сегментов. Каждый сегмент занимает 1 000 000 / 65 535 = 15 единиц измерения нашего мира. Если мы выберем в качестве единицы измерения метры, а все объекты будут иметь размер $1 \times 2 \times 1$ м и при этом находиться к одном сегменте,

z-буфер не будет столь полезным, поскольку все пикселы будут иметь одинаковое значение глубины.

ВНИМАНИЕ

Значения глубины, хранящиеся в z-буфере, не линейны, но общая идея верна.

Еще одной проблемой, связанной с *z*-буфером, является так называемая *z*-схватка. Она проиллюстрирована на рис. 10.9.

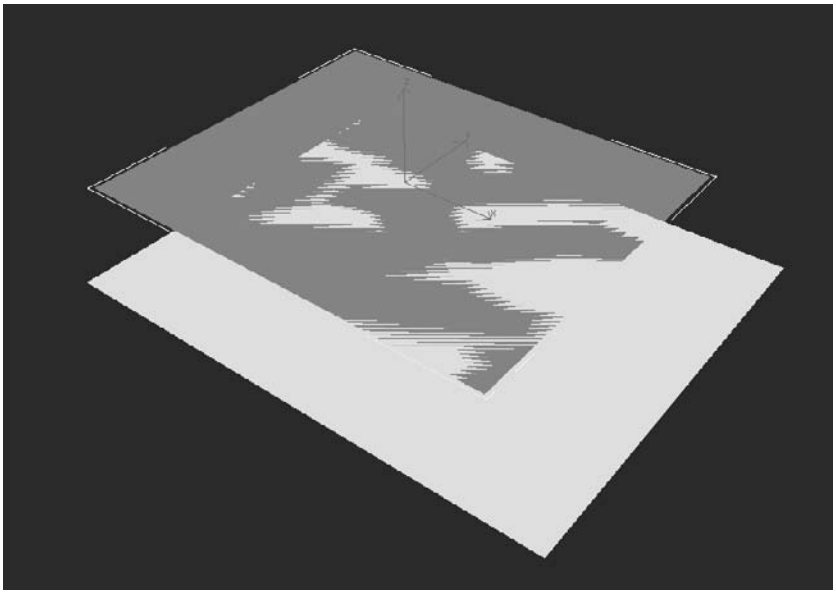

Рис. 10.9. Z-схватка

Два прямоугольника, изображенные на рис. 10.9, компланарны; это значит, что они лежат в одной и той же плоскости. Поскольку они пересекаются, у них также есть несколько общих пикселов, которые имеют одинаковые значения глубины. Однако из-за ограниченной точности вычислений с плавающей точкой GPU может неверно обработать одинаковые значения глубины для перекрывающихся пикселов. Пикселы, проходящие тест глубины, определяются в случайном порядке. С этой проблемой обычно можно справиться, отодвинув один из двух компланарных объектов от другого на небольшое расстояние. Значение подобного смещения зависит от нескольких факторов, поэтому лучше всего поэкспериментировать. В качестве итога можно сказать следующее.

- Не используйте значения, которые являются слишком маленькими или слишком большими для определения расстояний до ближней и дальней плоскостей отсечения.
- Избегайте применения компланарных объектов, немного их смещая.

Определяем 3D-ячейки

До этого момента мы использовали лишь несколько треугольников в качестве основы для объектов нашего мира. Пора перейти к более сложным объектам.

Мы уже говорили о том, что GPU — это просто очень большая машина для рисования треугольников. Все наши трехмерные объекты должны были состоять из треугольников. В предыдущих главах мы применяли два треугольника для того, чтобы создать плоский прямоугольник. Принципы, которые мы задействовали тогда, например размещение вершин, цвета, текстурирование и индексирование вершин, точно так же можно применить и в 3D. Треугольники теперь не ограничены двумерной плоскостью, мы можем свободно определять позицию каждой вершины в трехмерном пространстве.

Как же нам создать множество треугольников, составляющих трехмерный объект? Можно сделать это программно, как мы поступали при работе со спрайтами. Мы также можем использовать программы, позволяющие нам создавать 3D-объекты в стиле «что видишь, то и имеешь». В таких приложениях применено множество подходов от манипулирования отдельными треугольниками до простого определения нескольких параметров, с помощью которых будет выведена так называемая сеть треугольников (более красивое название для списка треугольников, с которым мы работаем).

Популярные пакеты программного обеспечения вроде Blender, 3ds Max, ZBrush и Wings 3D предоставляют пользователям широкий функционал для создания 3D-объектов. Некоторые из них бесплатны (например, Blender и Wings 3D), а некоторые распространяются на платной основе (например, 3ds Max и ZBrush). Все эти программы могут сохранять трехмерные модели в файлах с различными форматами. Сеть Интернет также переполнена бесплатными 3D-моделями. Мы напишем загрузчик для одного из самых простых и распространенных форматов в следующей главе.

В этой главе мы будем обходиться программными методами. Создадим один из простейших трехмерных объектов: куб.

Куб: Hello World в 3D

В нескольких последних главах мы уже использовали концепцию пространства моделей: это пространство, в котором определены наши модели, оно никак не связано с пространством создаваемого мира. Мы применяем соглашение о создании всех объектов вокруг начала координат пространства моделей, поэтому центр объекта совпадает в началом координат. Такая модель может быть повторно использована для отрисовки нескольких объектов в различных позициях мира. Эта особенность была применена в масштабном примере `BobTest`, рассмотренном в главе 7.

Первое, что необходимо определить для куба, — угловые точки. На рис. 10.10 показан куб со стороной, равной одной единице измерения (например, 1 м). Я так-

же немного разобрал этот куб для того, чтобы вы могли видеть отдельные грани, каждая из которых создана из двух треугольников. В реальности все грани сойдутся на границах куба, определяемых его угловыми точками.

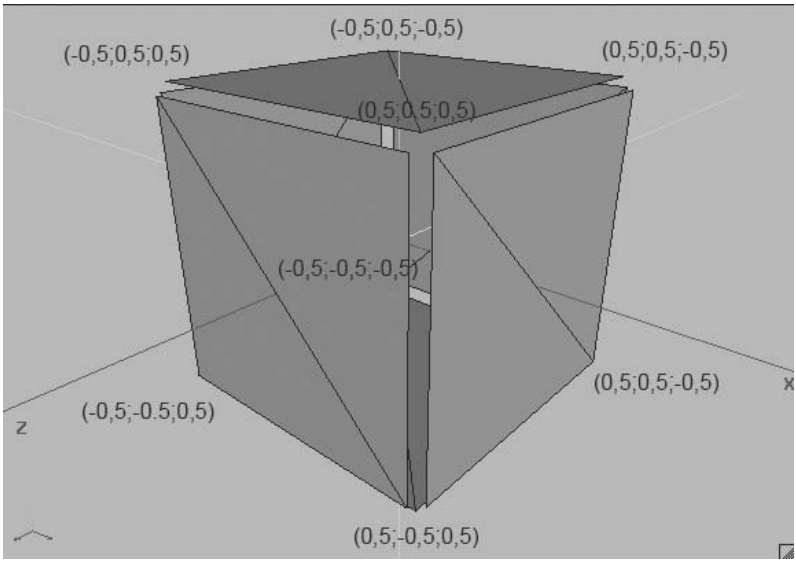

Рис. 10.10. Куб и его угловые точки

Куб имеет шесть граней, каждая из которых состоит из двух треугольников. Два треугольника каждой грани имеют по две общие вершины. Например, для передней грани куба общими являются вершины с координатами (–0,5; 0,5; 0,5) и (0,5; –0,5; 0,5). Для каждой грани необходимо определить четыре вершины, для всего куба получается $6 \cdot 4 = 24$ вершины. Однако нам нужно определить 36 индексов, а не 24. Это необходимо потому, что всего имеется $6 \cdot 2$ треугольника, каждый из которых использует 3 из 24 вершин. Мы можем создать сеть для этого куба, используя индексацию вершин, как показано в этом сниппете:

```
float[] vertices = { -0.5f, -0.5f,  0.5f,
                      0.5f, -0.5f,  0.5f,
                      0.5f,  0.5f,  0.5f,
                     -0.5f,  0.5f,  0.5f,

                      0.5f, -0.5f,  0.5f,
                      0.5f, -0.5f, -0.5f,
                      0.5f,  0.5f, -0.5f,
                      0.5f,  0.5f,  0.5f,

                      0.5f, -0.5f, -0.5f,
                     -0.5f, -0.5f, -0.5f,
                     -0.5f,  0.5f, -0.5f,
```

```
                        0.5f,  0.5f, -0.5f,

                       -0.5f, -0.5f, -0.5f,
                       -0.5f, -0.5f,  0.5f,
                       -0.5f,  0.5f,  0.5f,
                       -0.5f,  0.5f, -0.5f,

                       -0.5f,  0.5f,  0.5f,
                        0.5f,  0.5f,  0.5f,
                        0.5f,  0.5f, -0.5f,
                       -0.5f,  0.5f, -0.5f,

                       -0.5f, -0.5f,  0.5f,
                        0.5f, -0.5f,  0.5f,
                        0.5f, -0.5f, -0.5f,
                       -0.5f, -0.5f, -0.5f
};

short[] indices = { 0, 1, 3, 1, 2, 3,
                    4, 5, 7, 5, 6, 7,
                    8, 9, 11, 9, 10, 11,
                    12, 13, 15, 13, 14, 15,
                    16, 17, 19, 17, 18, 19,
                    20, 21, 23, 21, 22, 23,
};

Vertices3 cube = new Vertices3(glGraphics, 24, 36, false, false);
cube.setVertices(vertices, 0, vertices.length);
cube.setIndices(indices, 0, indices.length);
```

В этом фрагменте кода определяем лишь позиции вершин. Начинаем с передней грани, ее нижняя левая вершина находится в точке с координатами (–0,5; –0,5; 0,5). Далее определяем следующие три вершины этой грани, двигаясь против часовой стрелки. Следующей будет правая грань куба, за ней задняя, левая, верхняя и нижняя. Все они определяются по заданному шаблону. Сравните определения вершин с рис. 10.10.

Далее определяем индексы. Всего имеется 36 индексов — каждая строка приведенного кода определяет два треугольника, каждый из которых состоит из трех вершин. Индексы (0, 1, 3, 1, 2, 3) задают переднюю грань куба, следующие три — левую грань и т. д. Сравните эти индексы с вершинами, показанными в коде, а также с рис. 10.10.

После определения всех вершин и индексов мы можем сохранить их в экземпляре класса `Vertices3` для дальнейшей отрисовки, что мы и делаем в последних нескольких строках этого сниппета.

Что насчет координат текстур? Просто добавьте их к определению координат. Предположим, что у нас есть текстура размером 128 × 128, содержащая изображение одной грани ящика. Мы хотим использовать эту текстуру для каждой грани куба. На рис. 10.11 показано, как мы можем сделать это.

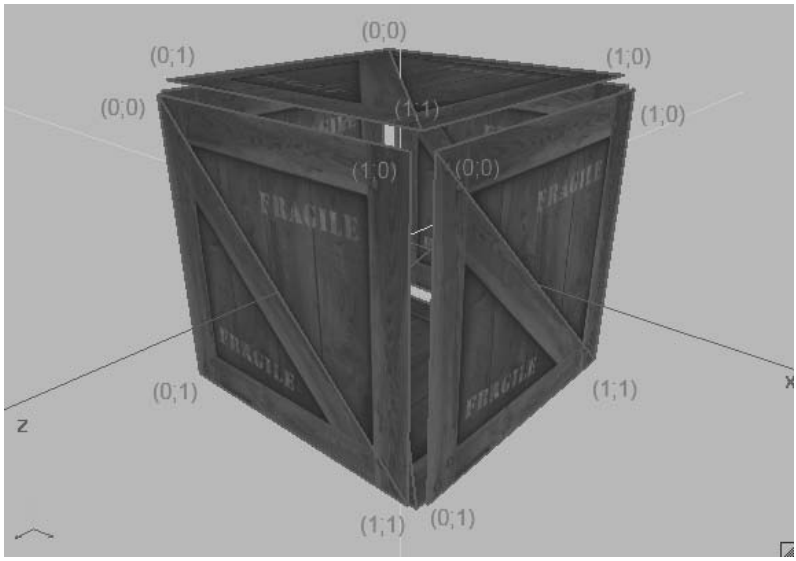

Рис. 10.11. Координаты текстур для каждой из вершин передней, левой и верхней граней (это же верно и для других граней)

Добавление текстурных координат для передней грани куба в коде будет выглядеть так:

```
float[] vertices = { -0.5f, -0.5f,  0.5f, 0, 1,
                      0.5f, -0.5f,  0.5f, 1, 1,
                      0.5f,  0.5f,  0.5f, 1, 0,
                     -0.5f,  0.5f,  0.5f, 0, 0,
                     // остальное аналогично
```

Конечно, необходимо указать экземпляру класса `Vertices3`, что он содержит и координаты текстур:

```
Vertices3 cube = new Vertices3(glGraphics, 24, 36, false, true);
```

Остается загрузить текстуру, разрешить наложение текстур с помощью метода `glEnable()`, а также привязать текстуру с помощью метода `Texture.bind()`. Напишем пример.

Пример

Мы хотим создать сеть для куба, показанную в предыдущих примерах кода, и применить текстуру ящика. Поскольку куб моделируется в пространстве моделей, расположенном вокруг начала его координат, нам придется использовать метод `glTranslatef()` для того, чтобы переместить его в пространство создаваемого мира так же, как мы поступили с моделью Боба в примере `BobTest`. Кроме того, мы хотим, чтобы наш куб вращался вокруг оси *y*. Этого мы можем достичь с помощью метода `glRotatef()`, что мы также делали в примере `BobTest`. В листинге 10.6 приведен полный код класса `CubeScreen`, содержащегося в классе `CubeTest`.

Листинг 10.6. Фрагменты класса CubeTest.java: отрисовка текстур куба

```
class CubeScreen extends GLScreen {
    Vertices3 cube;
    Texture texture;
    float angle = 0;

    public CubeScreen(Game game) {
        super(game);
        cube = createCube();
        texture = new Texture(glGame, "crate.png");
    }

    private Vertices3 createCube() {
        float[] vertices = { -0.5f, -0.5f,  0.5f, 0, 1,
                              0.5f, -0.5f,  0.5f, 1, 1,
                              0.5f,  0.5f,  0.5f, 1, 0,
                             -0.5f,  0.5f,  0.5f, 0, 0,

                              0.5f, -0.5f,  0.5f, 0, 1,
                              0.5f, -0.5f, -0.5f, 1, 1,
                              0.5f,  0.5f, -0.5f, 1, 0,
                              0.5f,  0.5f,  0.5f, 0, 0,

                              0.5f, -0.5f, -0.5f, 0, 1,
                             -0.5f, -0.5f, -0.5f, 1, 1,
                             -0.5f,  0.5f, -0.5f, 1, 0,
                              0.5f,  0.5f, -0.5f, 0, 0,

                             -0.5f, -0.5f, -0.5f, 0, 1,
                             -0.5f, -0.5f,  0.5f, 1, 1,
                             -0.5f,  0.5f,  0.5f, 1, 0,
                             -0.5f,  0.5f, -0.5f, 0, 0,

                             -0.5f,  0.5f,  0.5f, 0, 1,
                              0.5f,  0.5f,  0.5f, 1, 1,
                              0.5f,  0.5f, -0.5f, 1, 0,
                             -0.5f,  0.5f, -0.5f, 0, 0,

                             -0.5f, -0.5f,  0.5f, 0, 1,
                              0.5f, -0.5f,  0.5f, 1, 1,
                              0.5f, -0.5f, -0.5f, 1, 0,
                             -0.5f, -0.5f, -0.5f, 0, 0
        };

        short[] indices = { 0, 1, 3, 1, 2, 3,
                            4, 5, 7, 5, 6, 7,
                            8, 9, 11, 9, 10, 11,
                            12, 13, 15, 13, 14, 15,
                            16, 17, 19, 17, 18, 19,
                            20, 21, 23, 21, 22, 23,
        };

        Vertices3 cube = new Vertices3(glGraphics, 24, 36, false, true);
```

```
        cube.setVertices(vertices, 0, vertices.length);
        cube.setIndices(indices, 0, indices.length);
        return cube;
    }

    @Override
    public void resume() {
        texture.reload();
    }

    @Override
    public void update(float deltaTime) {
        angle += 45 * deltaTime;
    }

    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        GLU.gluPerspective(gl, 67,
                    glGraphics.getWidth() / (float) glGraphics.getHeight(),
                    0.1f, 10.0f);
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();

        gl.glEnable(GL10.GL_DEPTH_TEST);
        gl.glEnable(GL10.GL_TEXTURE_2D);
        texture.bind();
        cube.bind();
        gl.glTranslatef(0,0,-3);
        gl.glRotatef(angle, 0, 1, 0);
        cube.draw(GL10.GL_TRIANGLES, 0, 36);
        cube.unbind();
        gl.glDisable(GL10.GL_TEXTURE_2D);
        gl.glDisable(GL10.GL_DEPTH_TEST);
    }

    @Override
    public void pause() {
    }

    @Override
    public void dispose() {
    }
}
```

У нас есть поле для хранения сети куба, экземпляр класса `Texture`, а также переменная, имеющая тип числа с плавающей точкой, для хранения текущего угла поворота. В конструкторе мы создаем сеть куба и загружаем текстуру из файла

актива, который называется `crate.png`, текстуру, имеющую размеры 128 × 128 пикселов, представляющую собой изображение одной грани ящика.

Код, создающий куб, находится в методе `createCube()`. В нем просто устанавливаются вершины и индексы, а затем из них создается экземпляр класса `Vertices3`. Каждая вершина имеет позицию в трехмерном пространстве и координаты текстуры.

Метод `resume()` просто сообщает текстуре, что ей необходимо перезагрузиться. Помните, текстуры необходимо перезагружать всякий раз, когда теряется контекст OpenGL ES.

Метод `update()`увеличивает угол поворота, на который куб будет повернут по оси y.

Метод `present()` сначала устанавливает окно отображения, а также очищает буферы цвета и глубины. Далее устанавливается перспективная проекция и в видовую матрицу OpenGL ES загружается единичная матрица. Мы разрешаем проверку глубины и текстурирование, привязываем текстуру, а также сеть куба. Далее используется метод `glTranslatef()` для того, чтобы передвинуть куб в точку с координатами (0; 0; –3) пространства мира. С помощью метода `glRotatef()`поворачиваем куб в пространстве моделей вокруг оси y. Помните, что порядок, в котором преобразования применяются к сети, является обратным. Куб сначала будет повернут (в пространстве моделей), а затем его повернутая версия будет размещена в пространстве мира. Наконец, мы рисуем куб, убираем привязку `mesh` и отключаем проверку глубины и текстурирование. Нет необходимости отключать эти состояния. Я выполнил это только для того, чтобы в дальнейшем мы смогли отрисовать двухмерные элементы на верху трехмерной сцены. На рис. 10.12 показан результат работы нашей первой настоящей 3D-программы.

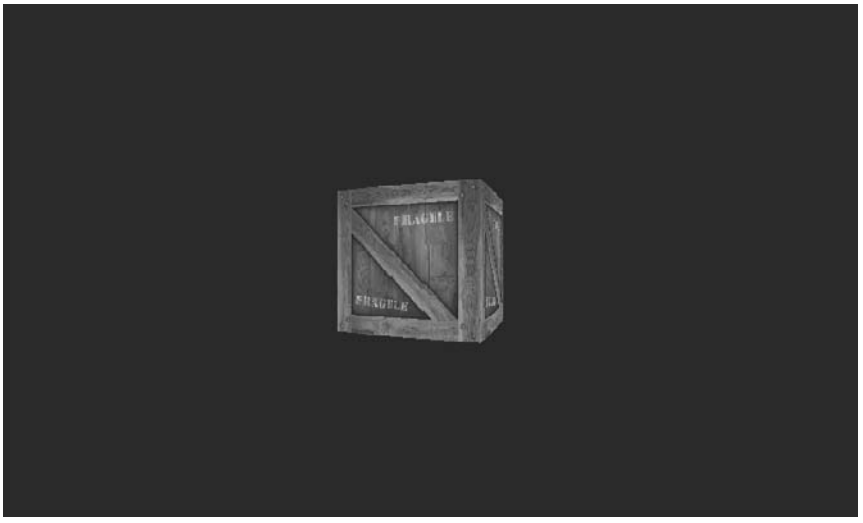

Рис. 10.12. Вращающаяся текстура куба в 3D

Снова матрицы и преобразования

В главе 6 мы немного говорили о матрицах. Вспомним некоторые их свойства:

- матрица переносит точки (или в нашем случае — вершины) на новую позицию. Это достигается умножением матрицы на позицию точки;
- матрица может переносить точки на каждой оси на некоторое значение;
- матрица может масштабировать точки. Это означает, что каждая координата точки умножается на некоторую константу;
- матрица может повернуть точку вокруг оси;
- умножение единичной матрицы на точку не повлияет на эту точку;
- результатом умножения одной матрицы на другую является новая матрица. Умножение точки на эту новую матрицу приведет к применению к точке обоих преобразований, закодированных к оригинальной матрице;
- Умножение матрицы на единичную матрицу не повлияет на саму матрицу.

OpenGL ES предоставляет три типа матриц.

- *Матрица проекций*. Используется для установки формы и размера окна просмотра. Эти параметры управляют типом проекции и тем, какая часть мира будет показываться пользователю.
- *Модельно-видовая матрица*. Применяется для преобразования моделей в модельном пространстве, а также для размещения модели в пространстве мира.
- *Матрица текстур*. Снова не обращаем на нее внимания, поскольку с ней нельзя корректно работать на многих устройствах.

Теперь, когда мы работаем в трехмерном пространстве, у нас есть гораздо больше настроек. К примеру, мы можем не только поворачивать модель вокруг оси *z*, как мы это делали с Бобом, но и вокруг любой оси. Единственное важное изменение — это появление дополнительной *z*-оси, которую также можно использовать для размещения объектов. На самом деле мы уже работали в 3D ранее, когда отрисовывали Боба в главе 6, мы просто игнорировали *z*-ось. Теперь мы можем осуществить и более масштабные задумки.

Матричный стек

До этого момента мы использовали матрицы в OpenGL ES примерно так:

```
gl.glMatrixMode(GL10.GL_PROJECTION);
gl.glLoadIdentity();
gl.glOrthof(-1, 1, -1, 1, -10, 10);
```

Первая инструкция устанавливает текущую активную матрицу. Все последующие операции с матрицами будут произведены над этой матрицей. В этом случае в качестве активной матрицы устанавливается единичная матрица, которая затем

умножается на матрицу ортографической проекции. Подобное мы уже делали с видовой матрицей:

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();
gl.glTranslatef(0, 0, -10);
gl.glRotate(45, 0, 1, 0);
```

В этом фрагменте кода действия производятся над модельно-видовой матрицей. Сначала загружается единичная матрица для того, чтобы очистить модельно-видовую матрицу перед вызовом. Далее эта матрица умножается на матрицу переноса и матрицу поворота. Важно сохранить именно такой порядок умножения, поскольку он определяет порядок применения преобразований к вершинам сетей. Последнее определенное нами преобразование будет применено к вершинам в первую очередь. В предыдущем случае мы сначала поворачиваем каждую вершину на 45° вокруг оси *y*. Далее все вершины перемещаются вдоль оси *z* на –10 единиц.

В обоих случаях все преобразования были закодированы в одной матрице: либо матрице проекций, либо видовой матрице. Но оказывается, что для каждого типа матриц существует целый стек матриц, который можно использовать.

В данный момент мы применяем лишь один элемент этого стека: вершину стека (ВС). ВС стека матриц применяется в OpenGL ES для преобразования вершин, независимо от того, идет ли речь о проекционной матрице или модельно-видовой матрице. Любая матрица, располагающаяся ниже ВС, не оказывает никакого влияния на точки, ожидая, пока она не станет ВС. Как же управлять этим стеком?

В OpenGL ES есть два метода, с помощью которых можно поместить в стек или вытолкнуть из него текущую ВС:

```
GL10.glPushMatrix();
GL10.glPopMatrix();
```

Аналогично методу `glTranslatef()` и подобным ему эти методы всегда работают с текущим активным матричным стеком, который устанавливается с помощью метода `glMatrixMode()`.

Метод `glPushMatrix()` принимает текущую ВС, копирует ее, а затем помещает в стек. Метод `glPopMatrix()` принимает текущую ВС и поднимает ее, удаляя вершину из стека. После этого элемент, находящийся под вершиной, становится новой ВС. Метод `glPopMatrix()` принимает текущую ВС и выталкивает ее из стека. Напишем небольшой пример:

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();
gl.glTranslate(0,0,-10);
```

До этого момента в стеке видовых матриц существовала всего одна матрица. «Сохраним» ее:

```
gl.glPushMatrix();
```

Теперь мы сделали копию текущей ВС и сместили вниз предыдущую ВС. У нас в стеке теперь есть две матрицы, каждая из которых кодирует параллельный перенос по оси *z* на –10 единиц.

```
gl.glRotatef(45, 0, 1, 0);
gl.glScalef(1, 2, 1);
```

Поскольку матричные операции всегда работают с ВС, теперь в верхней матрице закодированы операции масштабирования, поворота и параллельного переноса. Матрица, которую мы сместили вниз, по-прежнему содержит лишь операцию параллельного переноса. При отрисовке сети, заданной в пространстве моделей, вроде нашего куба, сначала она будет масштабирована по оси *y*, потом повернута по оси *y*, а затем перенесена на –10 единиц по оси *z*. Вытолкнем из стека его вершину:

```
gl.glPopMatrix();
```

Это действие удаляет ВС, а матрица под ней становится новой ВС. В нашем примере таковой является оригинальная матрица переноса. После вызова этого метода в стеке снова находится только одна матрица — та, которую мы инициализировали в самом начале примера. Если объект будет отрисован сейчас, он будет лишь перенесен на –10 единиц по оси *z*. Матрицы, содержащей операции масштабирования, поворота и переноса, больше не существует, поскольку она была вытолкнута из стека. На рис. 10.13 показано, что произойдет со стеком матриц при выполнении предшествующего кода.

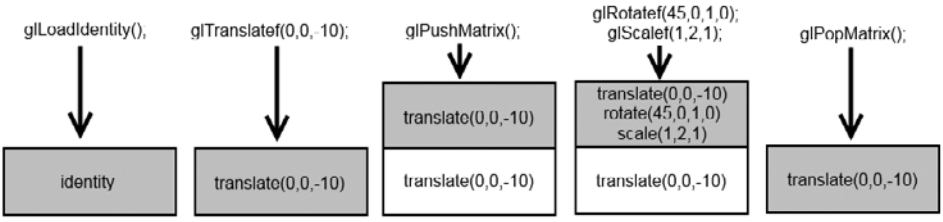

Рис. 10.13. Манипуляции с матричным стеком

Для чего можно использовать матричный стек? Первое, для чего он может быть использован, — запоминание преобразований, которые должны быть применены для всех объектов мира. Предположим, мы хотим, чтобы все объекты мира имели смещение на 10 единиц по каждой оси. Мы можем сделать следующее:

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();
gl.glTranslatef(10, 10, 10);
for( MyObject obj: myObjects) {
   gl.glPushMatrix();
```

```
    gl.glTranslatef(obj.x, obj.y, obj.z);
    gl.glRotatef(obj.angle, 0, 1, 0);
    // отрисовка модели объекта, заданного в пространстве модели,
    // например в кубе
    gl.glPopMatrix();
}
```

Этот шаблон мы используем и далее, когда будем обсуждать создание системы камер в 3D. Позиция камеры и ее ориентация часто кодируются в виде матрицы. Мы загрузим эту матрицу камеры, которая преобразует все объекты так, чтобы мы смогли увидеть их с позиции камеры. Однако есть кое-что получше матричного стека.

Иерархическая структура матричного стека

Что такое иерархическая структура? Примером подобной структуры является Солнечная система. В центре располагается Солнце. Вокруг него находятся планеты, которые вращаются вокруг него на определенном расстоянии. Вокруг некоторых планет есть спутники, которые вращаются вокруг этих планет. И Солнце, и планеты, и спутники крутятся вокруг своих центров (что-то вроде того). Мы можем построить такую систему с помощью матричного стека.

Солнце имеет позицию в нашем мире и крутится вокруг себя. Все планеты двигаются вместе с Солнцем, поэтому, если позиция Солнца поменяется, планеты также должны изменить свои позиции. Для размещения Солнца можно использовать метод `glTranslatef()`, а чтобы заставить его вращаться вокруг своей оси, — метод `glRotatef()`.

Планеты имеют позицию, относительную позиции Солнца, и так же, как и Солнце, вращаются вокруг своей оси. Вращение планет вокруг своей оси может быть осуществлено с помощью метода `glRotatef()`, а вращение вокруг Солнца — с использованием методов `glTranslatef()` и `glRotatef()`. Заставить планету перемещаться вместе с Солнцем можно с помощью дополнительного вызова метода `glTranslatef()`.

Спутники имеют позицию относительно планет, чьими спутниками они являются. Они также вращаются вокруг своей оси и вокруг планеты. Вращение спутника самого по себе может быть осуществлено с помощью метода `glRotatef()`, а вращение вокруг планеты — с применением методов `glTranslatef()` и `glRotatef()`. Заставить спутник перемещаться вместе с планетой можно с помощью метода `glTranslatef()`.

И поскольку планеты перемещаются вместе с Солнцем, спутники также должны перемещаться вместе с ним, чего можно достигнуть использованием метода `glTranslatef()`.

Здесь мы можем наблюдать так называемые взаимоотношения предков и потомков. Солнце является предком каждой планеты, а каждая планета является предком каждого спутника. Каждая планета является потомком Солнца, а каждый

спутник — потомком планеты. Это означает, что позиция каждого потомка задается относительно его предка, а не начала координат мира.

У Солнца нет предка, поэтому его позиция задана относительно начала координат. Планета является потомком Солнца, поэтому ее позиция задана относительно Солнца; Луна является потомком планеты, поэтому ее позиция задана относительно планеты. Вы можете рассматривать центр каждого предка как начало системы координат, в которой определяются его потомки.

Вращение вокруг своей оси каждого из объектов в системе не зависит от предков. Это же верно и в том случае, если бы мы захотели промасштабировать объект. Данные операции относительны лишь центра самого объекта. То же самое происходит внутри пространства моделей.

Простая «солнечная» система, центром которой является ящик

Создадим небольшой пример, очень простую солнечную систему, центром которой будет являться ящик. Допустим, имеется один ящик, расположенный в точке (0; 0; –6) системы координат создаваемого мира. Вокруг этого ящика-«солнца» на расстоянии, равном 3 единицы, мы хотим поместить ящик-«планету», вращающийся вокруг «солнца». «Планета» должна быть меньше «солнца», выберем для нее размер 0,2 единицы. Вокруг ящика-«планеты» поместим ящик-«спутник». Расстояние между «планетой» и «спутником» будет равно 1 единице, а ящик-«спутник» будет уменьшен до 0,1 единицы. Все объекты вращаются вокруг их относительных предков в плоскости *xz*, а также вокруг собственных осей *y*. На рис. 10.14 показан примерный вид этой модели.

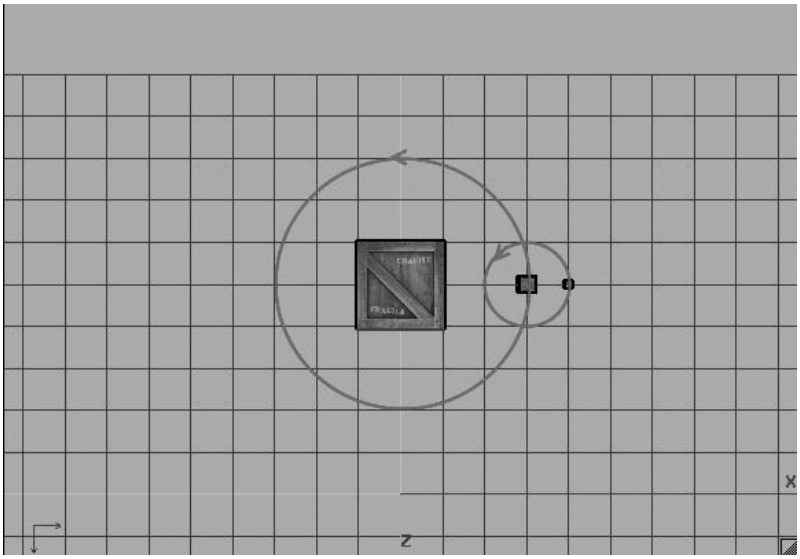

Рис. 10.14. Наша «солнечная» система

Класс HierarchicalObject

Определим простой класс, который будет содержать общий объект солнечной системы со следующими свойствами:

- позиция относительно центра его предка;
- угол поворота вокруг предка;
- угол поворота вокруг своей оси *y*;
- масштаб;
- список потомков;
- ссылка на экземпляр класса `Vertices3`, необходимый для отрисовки.

Класс `HierarchicalObject` должен обновлять свои углы поворота и углы поворота потомков, а также отрисовывать себя и всех своих потомков. Этот процесс рекурсивен, поскольку каждый потомок отрисовывает собственных потомков. Мы будем использовать методы `glPushMatrix()` и `glPopMatrix()` для сохранения преобразований, примененных к предкам, чтобы их потомки могли перемещаться вместе с ними. Код этого класса показан в листинге 10.7.

Листинг 10.7. Класс HierarchicalObject.java, представление объекта внутри системы ящика

```
package com.badlogic.androidgames.gl3d;

import java.util.ArrayList;
import java.util.List;
import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.gl.Vertices3;

public class HierarchicalObject {
    public float x, y, z;
    public float scale = 1;
    public float rotationY, rotationParent;
    public boolean hasParent;
    public final List<HierarchicalObject> children = new
ArrayList<HierarchicalObject>();
    public final Vertices3 mesh;
```

Первые три члена содержат позицию объекта относительно его предка (или относительно начала координат мира, если предка нет). Следующий член включает в себя масштаб объекта. Член `rotationY` хранит угол поворота объекта вокруг себя, а член `rotationParent` — угол поворота вокруг центра предка.

Член `hasParent` предоставляет информацию о том, если ли у объекта предок. Если предка нет, к объекту не применяется поворот вокруг предка. Этот член создан для «солнца» нашей системы. Наконец, в классе есть список потомков, а также ссылка на экземпляр класса `Vertices3`, содержащий сеть куба, которая будет использована для отрисовки каждого объекта.

```
    public HierarchicalObject(Vertices3 mesh, boolean hasParent) {
        this.mesh = mesh;
        this.hasParent = hasParent;
    }
```

Конструктор просто получает экземпляр класса `Vertices3`, а также булеву переменную, определяющую, имеет ли объект предка.

```
    public void update(float deltaTime) {
        rotationY += 45 * deltaTime;
        rotationParent += 20 * deltaTime;
        int len = children.size();
        for (int i = 0; i < len; i++) {
            children.get(i).update(deltaTime);
        }
    }
```

В методе `update()` сначала обновляются члены `rotationY` и `rotationParent`. Каждый объект будет поворачиваться на 45° в секунду вокруг себя и на 20° в секунду вокруг своего предка. Этот метод будет также вызываться рекурсивно для каждого потомка объекта.

```
    public void render(GL10 gl) {
        gl.glPushMatrix();
        if (hasParent)
            gl.glRotatef(rotationParent, 0, 1, 0);
        gl.glTranslatef(x, y, z);
        gl.glPushMatrix();
        gl.glRotatef(rotationY, 0, 1, 0);
        gl.glScalef(scale, scale, scale);
        mesh.draw(GL10.GL_TRIANGLES, 0, 36);
        gl.glPopMatrix();
        int len = children.size();
        for (int i = 0; i < len; i++) {
            children.get(i).render(gl);
        }
        gl.glPopMatrix();
    }
}
```

Метод `render()` является самым интересным. Первое, что в нем происходит, — текущая ВС модельно-видовой матрицы, которая станет активной извне объекта, помещается глубже в стек. Поскольку метод рекурсивен, таким образом сохранятся все преобразования предков.

Далее к объекту применяются все преобразования, которые повернут его вокруг предка и поместят его в мире относительно центра предка. Помните, что преобразования применяются в обратном порядке, поэтому на самом деле сначала объект переместится вслед за предком, а затем повернется вокруг него. Преобразование поворота применяется только в том случае, если у объекта есть предок. Ящик-«солнце»

не имеет предка в создаваемой системе, поэтому он не будет поворачиваться. Эти преобразования относительны предка объекта, они также будут применены и к потомкам объекта. Перемещение «планеты» вокруг «солнца» также передвинет «закрепленный» за ней «спутник». Следующее действие — ВС снова перемещается глубже в стек. До этого момента она содержала преобразования предка, а также преобразования объекта относительно предка. Необходимо сохранить эту матрицу до тех пор, пока она будет прикрепляться к потомкам объекта. Вращение объекта вокруг себя и его масштабирование к потомкам не применяются, поэтому эти преобразования применяются к копии ВС (которая создается при перемещении ВС вглубь стека).

Далее применяются преобразования поворота вокруг себя и масштабирования, а затем объект отрисовывается с помощью сети ящика, на которую в объекте содержится ссылка. Подумаем о том, что случится с вершинами пространства моделей из-за воздействия матрицы, являющейся ВС. Вспомните порядок применения преобразований: от последнего к первому.

Сначала размер ящика будет масштабирован. Далее он повернется вокруг своей оси. Эти два преобразования применяются к вершинам в пространстве моделей. Потом вершины будут параллельно перенесены относительно позиции предка. Если объект не имеет предка, его вершины будут параллельно перенесены в пространстве мира. Если объект имеет предка, вершины будут перенесены в его пространстве (он будет являться центром своей системы). Объект также повернется вокруг предка, если он его имеет. Если вы пройдете обратно по цепи рекурсии, то увидите, что также применяем преобразования к предку объекта и т. д. С помощью этого механизма «спутник» сначала переместится в координатной системе предка, а затем в координатной системе «солнца», эквивалентной пространству мира.

Как только текущий объект отрисовывается, ВС выталкивается из стека, и новая ВС содержит только преобразования и поворот объекта относительно своего предка. Нам не хочется, чтобы к потомку применялись «местные» преобразования объекта (например, поворот по оси y и масштабирование). Все, что остается сделать, — рекурсивно перейти к потомку.

ПРИМЕЧАНИЕ

Позицию объекта, представляемого классом HierarchicalObject, следует хранить в виде вектора для более простой работы с ним. Однако нам следует еще написать класс Vector3. Сделаем это в следующей главе.

Собираем все воедино

Используем только что созданный нами класс `HierarchicalObject` в подходящей программе. Для этого я просто скопировал весь код примера `CubeTest`, в котором также есть метод `createCube()`. Его мы используем повторно. Я переименовал класс `HierarchyTest`, а также `CubeScreen` в `HierarchyScreen`. Нужно создать иерархию объектов и вызвать методы `HierarchicalObject.update()` и `HierarchicalObject.render()` в подходящих местах программы. В листинге 10.8 приведены самые важные фрагменты примера `HierarchyTest`.

Листинг 10.8. Фрагменты класса HierarchyTest.java: реализация простой иерархической системы

```
class HierarchyScreen extends GLScreen {
        Vertices3 cube;
        Texture texture;
        HierarchicalObject sun;
```

В класс добавился лишь один новый член, который называется sun. Он представляет собой корень создаваемой иерархии. Поскольку все прочие объекты хранятся внутри объекта sun в виде его потомков, нам не нужно хранить его явно.

```
        public HierarchyScreen(Game game) {
            super(game);
            cube = createCube();
            texture = new Texture(glGame, "crate.png");

            sun = new HierarchicalObject(cube, false);
            sun.z = -5;

            HierarchicalObject planet = new HierarchicalObject(cube, true);
            planet.x = 3;
            planet.scale = 0.2f;
            sun.children.add(planet);

            HierarchicalObject moon = new HierarchicalObject(cube, true);
            moon.x = 1;
            moon.scale = 0.1f;
            planet.children.add(moon);
        }
```

В конструкторе инициализируется наша иерархическая система. Сначала загружается текстура и создается сеть куба, которая будет использована всеми объектами. Далее создается «солнце». У него нет предка, оно находится в точке (0; 0; –5) относительно начала системы координат создаваемого мира (места, где располагается виртуальная камера). Далее создается ящик-«планета», который вращается вокруг «солнца». Он находится в точке (0; 0; 3) относительно «солнца» и имеет масштаб, равный 0,2. Поскольку длина грани ящика в пространстве моделей равна 1, после применения коэффициента масштаба ящик будет отрисован с гранью, длина которой равна 0,2 единицы. Важный момент — «планета» добавляется в качестве потомка «солнца». Для «спутника» также будет осуществлено что-то похожее. Он находится в позиции (0; 0; 1) относительно «планеты» и имеет масштаб, равный 0,1 единицы. Он добавляется к «планете» в качестве потомка. Результат инициализации вы можете увидеть на рис. 10.14, имеющем такую же систему координат.

```
        @Override
        public void update(float deltaTime) {
            sun.update(deltaTime);
        }
```

В методе update() мы просто указываем «солнцу» обновиться. Оно рекурсивно вызовет те же методы для всех своих потомков, которые вызовут этот метод для своих потомков. Это обновит угол поворота всех объектов иерархии.

```
        @Override
        public void present(float deltaTime) {
            GL10 gl = glGraphics.getGL();
            gl.glViewport(0, 0, glGraphics.getWidth(),
                                glGraphics.getHeight());
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
            gl.glMatrixMode(GL10.GL_PROJECTION);
            gl.glLoadIdentity();
            GLU.gluPerspective(gl, 67, glGraphics.getWidth()
                    / (float) glGraphics.getHeight(), 0.1f, 10.0f);
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            gl.glLoadIdentity();
            gl.glTranslatef(0, -2, 0);

            gl.glEnable(GL10.GL_DEPTH_TEST);
            gl.glEnable(GL10.GL_TEXTURE_2D);
            texture.bind();
            cube.bind();

            sun.render(gl);

            cube.unbind();
            gl.glDisable(GL10.GL_TEXTURE_2D);
            gl.glDisable(GL10.GL_DEPTH_TEST);
        }
// остальное как в классе CubeScreen
```

Наконец, рассмотрим метод render(). Начнем с простой инициализации окна просмотра и очистки буферов цвета и глубины. Кроме того, инициализируется матрица перспективной проекции, а также единичная матрица загружается в качестве видовой матрицы OpenGL ES. Последующий вызов метода glTranslatef() довольно интересен: это сместит «солнечную» систему вниз по оси y на 2 единицы. Таким образом мы сможем посмотреть на систему сверху. Это действие можно рассматривать и как перемещение камеры на 2 единицы вверх. Такая интерпретация является ключевой для выбора подходящей системы камер, которую мы рассмотрим в следующей главе.

Как только все основные компоненты инициализированы, разрешаются тестирование глубины и текстурирование, производится привязка текстуры и сети куба, а «солнцу» дается указание отрисовать себя. Поскольку все объекты иерархии используют одинаковую текстуру и сеть, связывать их необходимо всего лишь один раз. Вызов этого метода отрисует солнце и всех его потомков рекурсивно, что было подчеркнуто в предыдущем разделе. Наконец, отключаем тестирование глубины и текстурирование просто ради развлечения. На рис. 10.15 показан результат работы программы.

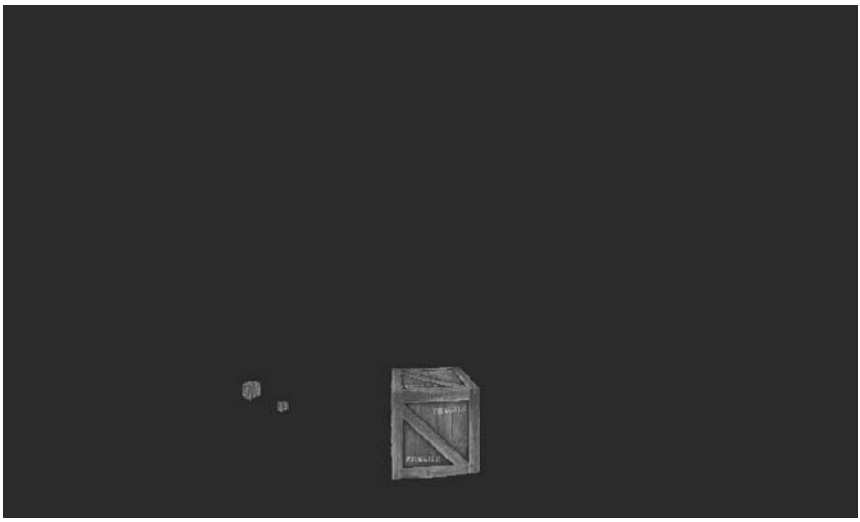

Рис. 10.15. Наша «солнечная» система в действии

Отлично, все работает так, как мы и ожидали. Наше «солнце» вращается только вокруг себя. «Планета» вращается вокруг «солнца» на расстоянии в 3 единицы, также вращается вокруг себя, а ее размеры в пять раз меньше размеров «солнца». «Спутник» вращается вокруг «планеты», а также двигается вместе с ней благодаря стеку матриц. Он также имеет локальные преобразования в виде поворота вокруг себя и масштабирования.

Класс `HierarchicalObject` является довольно общим, поэтому вы можете использовать его для различных целей. Добавьте больше «планет» и «спутников» или даже «спутников» «спутников». Хорошенько поработайте со стеком матриц. Вы сможете многого добиться благодаря большому количеству практики. Вам необходимо визуализировать у себя в голове результат применения всех преобразований.

ПРИМЕЧАНИЕ

Не слишком увлекайтесь работой с матричным стеком. Его глубина ограничена обычно между 16 и 32 матрицами в зависимости от GPU и драйвера. Четыре уровня иерархии — это максимум, который мне когда-либо приходилось использовать в приложении.

Простая система камер

В последнем примере мы увидели подсказку, как можно реализовать систему камер в 3D.

Мы использовали метод `glTranslatef()`, чтобы переместить весь мир вниз на две единицы по оси *y*. Поскольку расположение камеры фиксировано (она находится в начале координат и обращена вдоль отрицательной половины оси *z*), такой подход создает впечатление того, что это камера была передвинута на 2 единицы. Координаты всех объектов по оси *y* по-прежнему равны 0.

Это очень похоже на выражение «Если гора не идет к Магомету, Магомет идет к горе». Вместо перемещения камеры мы передвинули весь мир. Предположим, что камеру необходимо расположить в позиции (10; 4; 2). Все, что для этого нужно, — использовать метод `glTranslatef()` следующим образом:

```
gl.glTranslatef(-10,-4,-2);
```

Если же нужно повернуть камеру вокруг оси y на 45°, этот метод следует вызвать именно так:

```
gl.glRotatef(-45,0,1,0);
```

Можно совместить эти два метода, как мы поступаем для «нормальных» объектов:

```
gl.glTranslatef(-10,-4,-2);
gl.glRotatef(-45,0,1,0);
```

Секрет заключается в том, что аргументы метода преобразования инвертируются. Вспомним предыдущий пример. Мы знаем, что «настоящая» камера обречена находиться в начале координат и смотреть в одну сторону. Применяя обратные преобразования, мы изменяем картинку, отображаемую камерой. Использование виртуальной камеры, повернутой на 45°, является равнозначным применению фиксированной камеры и повороту мира на −45°. Это же верно и для параллельного переноса. Виртуальная камера должна быть помещена в точку (10; 4; 2). Но поскольку реальная камера всегда находится в начале координат, нам нужно лишь перенести все объекты мира используя инвертированный вектор этой позиции, равный (−10; −4; −2).

Если мы изменим следующие три строки метода `present()` предыдущего примера:

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();
gl.glTranslatef(0, -2, 0);
```

на эти четыре:

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();
gl.glTranslatef(0, -3, 0);
gl.glRotatef(45, 1, 0, 0);
```

то получим результат, показанный на рис. 10.16.

По задумке наша камера теперь находится в точке (0; 3; 0) и смотрит вниз на сцену под углом −45° (аналогично повороту камеры на −45° вокруг оси x). На рис. 10.17 показана сцена, отображаемая из новой точки.

Можно определить очень простую камеру с четырьмя атрибутами:

- позиция в пространстве мира;
- поворот вокруг оси x — аналогично наклону головы вверх и вниз;
- поворот вокруг оси y — аналогично повороту головы влево и вправо;
- поворот вокруг оси z — аналогично наклону головы влево и вправо.

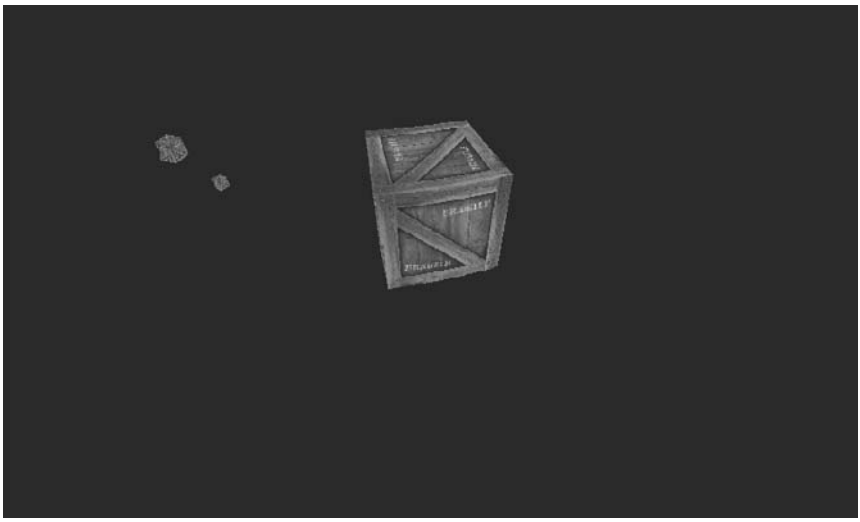

Рис. 10.16. Смотрим на мир сверху из точки (0; 3; 0)

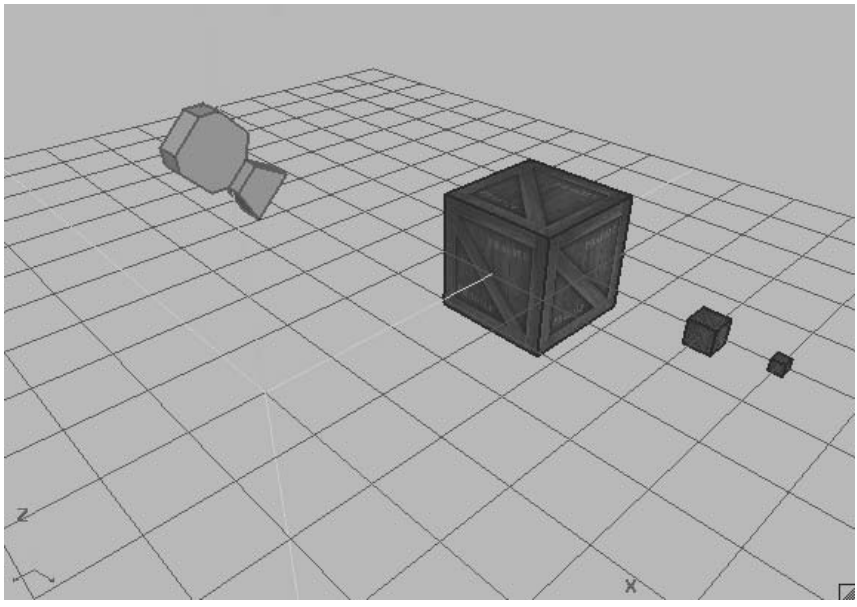

Рис. 10.17. Размещение и ориентация камеры

Основываясь на этих атрибутах, мы можем использовать методы OpenGL ES для создания матрицы камеры. Такая камера называется камерой поворота на углы Эйлера. Множество игр жанра FPS используют подобную камеру для эмуляции поворота головы. Обычно значение координаты по оси *z* не изменяется, в отличие от координат по осям *x* и *y*. Порядок, в котором применяются преобразования, очень

важен. В играх жанра FPS сначала производится поворот по оси *x*, а затем по оси *y*:

```
gl.glTranslatef(-cam.x,- cam.y,-cam.z);
gl.glRotatef(cam.yaw, 0, 1, 0);
gl.glRotatef(cam.pitch, 1, 0, 0);
```

Такая упрощенная модель используется во многих играх. Если бы мы применяли преобразование и по оси *z*, то смогли бы наблюдать эффект, называемый «шарнирный замок» (gimbal lock). Этот эффект отменит один из поворотов, основываясь на конкретной конфигурации.

ПРИМЕЧАНИЕ

Объяснить феномен шарнирного замка на словах или даже при помощи изображений очень сложно. Поскольку мы осуществляем вращение только по осям *x* и *y*, проблем не возникает. Чтобы понять, что же такое шарнирный замок, я советую вам поискать в Интернете соответствующее видео. Эту проблему нельзя обойти при помощи поворотов на углы Эйлера. Ее решение очень сложно математически, поэтому оно не входит в эту книгу.

Второй подход к созданию очень простой системы камер — использование метода `GLU.glLookAt()`.

```
GLU.gluLookAt(GL10 gl,
              float eyeX, float eyeY, float eyeZ,
              float centerX, float centerY, float centerZ,
              float upX, float upY, float upZ);
```

Как и метод `GLU.gluPerspective()`, он умножит текущую активную матрицу на матрицу преобразования. В этом случае таковой матрицей является матрица камер, которая преобразует мир:

- `gl` — это просто экземпляр класса `GL10`, который используется для отрисовки;
- `eyex`, `eyey` и `eyez` — определяют позицию камеры в мире;
- `centerx`, `centery` и `centerz` — задают точку мира, на которую смотрит камера;
- `upX`, `upY` и `upZ` — определяют так называемый верхний вектор. Представьте, что он является стрелой, выходящей из верхушки вашей головы и указывающей вверх. Наклоните голову влево или вправо — и стрела будет указывать в том же направлении, что и верхушка вашей головы.

Верхний вектор обычно устанавливается равным (0; 1; 0), даже если это и не совсем верно. Метод `gluLookAt()` может заново нормализовать данный вектор в большинстве случаев. На рис. 10.18 показана наша сцена, камера которой располагается в точке (3; 3; 0) и смотрит на точку (0; 0; –5), как и «действительный» верхний вектор.

Мы можем заменить код метода `HierarchyScreen.present()`, который изменили ранее, следующим сниппетом:

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();
GLU.gluLookAt(gl, 3, 3, 0, 0, 0, -5, 0, 1, 0);
```

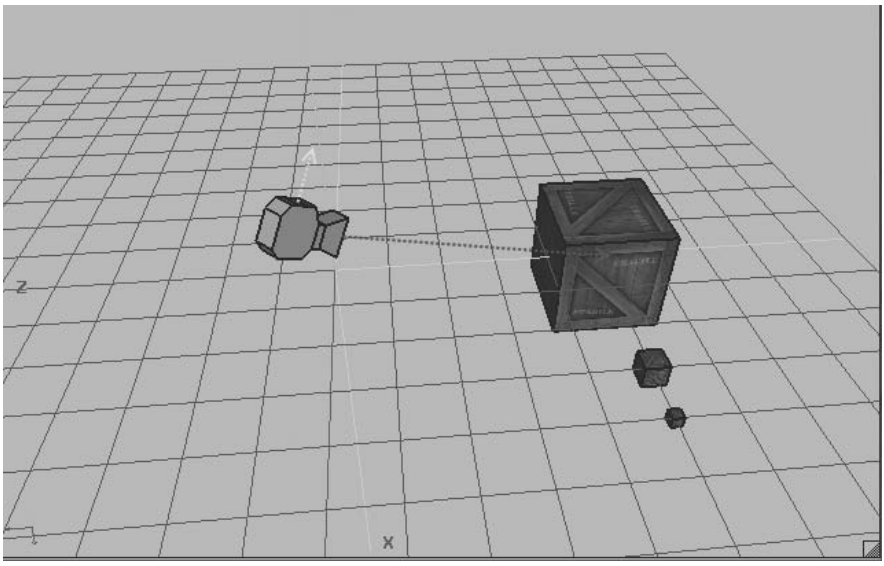

Рис. 10.18. Камера находится в позиции (3; 3; 0) и смотрит в точку (0; 0; –3)

В этот раз я также закомментировал вызов метода `sun.update()`, поэтому иерархия будет выглядеть так, как и на рис. 10.18. На рис. 10.19 показан результат использования камеры.

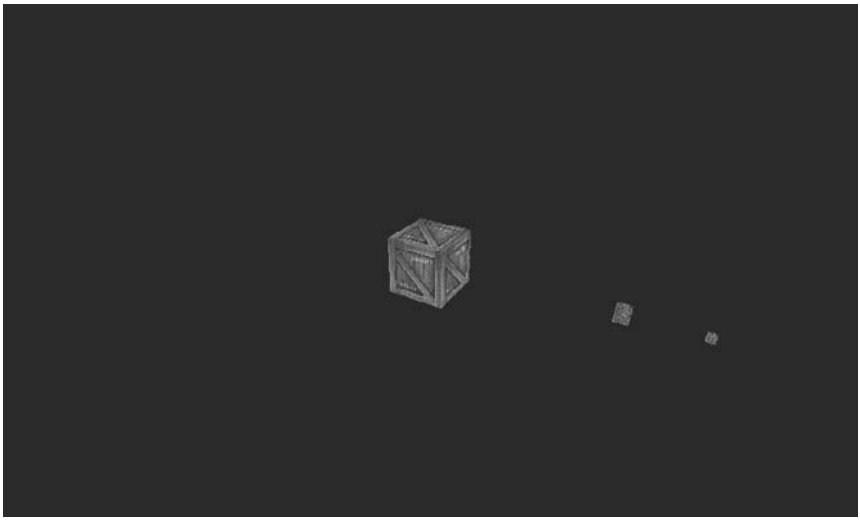

Рис. 10.19. Камера в действии

Этот вид камеры отлично подходит в том случае, когда мы хотим следовать за персонажем или необходим более качественный контроль за просмотром сцены при определении лишь позиции камеры и точки, в которую она будет смотреть.

Пока прекратим разговор о камерах. В следующей главе напишем два простых класса для камеры, годящейся для игры жанра FPS, и для камеры, которая может следовать за объектом.

Подводя итог

Теперь вы знаете основы программирования 3D-графики с помощью OpenGL ES. Вы получили информацию, как инициализировать перспективное окно просмотра, как определить позиции вершин в трех измерениях, а также о том, что такое *z*-буфер. Кроме того, вы узнали, что *z*-буфер может быть как другом, так и врагом, в зависимости от правильности его использования. Мы создали наш первый 3D-объект: текстурированный куб. Оказалось, что это довольно легко. Наконец, мы еще немного поговорили о матрицах и преобразованиях, а также создали иерархическую и очень простую систему камер. Но это даже не вершина айсберга. В следующей главе мы снова рассмотрим несколько тем, которые мы обсуждали еще в главе 7, но теперь уже с точки зрения 3D-программирования. Мы также изучим несколько новых приемов, которые очень пригодятся для создания нашей финальной игры. Я рекомендую вам побольше поработать с примерами этой главы. Создайте новые фигуры, а также совершенно безумные преобразования и системы камер.

11 Трюки при разработке 3D-игр

3D-программирование — это очень сложное и широкое поле деятельности. В данной главе рассматриваются темы, являющиеся минимальными требованиями к программисту, который желает написать простую 3D-игру.

- Мы снова рассмотрим наш знакомый вектор и присоединим к нему еще одну координату.
- Освещение — это важная часть любой трехмерной игры. Мы изучим, как реализовать простое освещение с помощью OpenGL ES.
- Определение объектов программно выглядит довольно громоздко. Мы рассмотрим простой формат 3D-файлов, с помощью которого можно загружать и отрисовывать 3D-модели, созданные с помощью ПО для 3D-моделирования.
- В главе 8 мы обсуждали представление объектов и определение столкновений. В этой главе мы рассмотрим те же вопросы применительно к трехмерному пространству. Мы также снова коснемся некоторых физических концепций, которые мы обсудили в главе 10, в этот раз также применительно к 3D.

  Начнем с 3D-векторов.

Перед стартом

Как обычно, в данной главе мы создадим несколько простых примеров. Чтобы это сделать, мы просто создадим новый проект и скопируем весь исходный код фреймворка, который разработали ранее.

Как и в предыдущих главах, у нас будет единая начальная активность, предоставляющая нам доступные тесты в виде списка. Мы назовем ее `GLAdvancedStarter` и сделаем ее активностью, используемой по умолчанию. Просто скопируем активность под названием `GL3DBasicsStarter` и заменим имена классов тестов. Нам также понадобится добавить каждую из тестовых активностей в подходящий элемент `<activity>` манифеста.

Как и прежде, каждый тест будет расширять возможности класса `GLGame`; код будет создан как класс `GLScreen`, связанный с экземплярами класса `GLGame`. Для экономии места я буду приводить лишь самые важные фрагменты класса `GLScreen`. Все тесты и начальная активность будут находиться в пакете `com.badlogic.androidgames.gladvanced`. Некоторые классы станут частью нашего фреймворка и войдут в соответствующие пакеты фреймворков.

Векторы в 3D

В главе 8 мы обсудили векторы и их интерпретацию в двухмерном пространстве. Как вы могли догадаться, все рассмотренные нами моменты также будут работать и в 3D. Все, что нужно сделать, — добавить лишь еще одну координату к вектору, которая будет называться *z*-координатой.

Операции, рассмотренные для двухмерных векторов, могут быть с легкостью перенесены в третье измерение. В 3D вектор определяется следующим выражением:

```
v = (x, y, z)
```

Сложение трехмерных векторов выглядит следующим образом:

```
c = a + b = (a.x, a.y, b.z) + (b.x, b.y, b.z) = (a.x + b.x, a.y + b.y, a.z + b.z)
```

Вычитание работает точно так же:

```
c = a – b = (a.x, a.y, b.z) – (b.x, b.y, b.z) = (a.x – b.x, a.y – b.y, a.z – b.z)
```

Умножение вектора на скалярное значение производится так:

```
a' = a x scalar = (a.x x scalar, a.y x scalar, a.z x scalar)
```

Измерение длины трехмерного вектора — также несложная операция; мы просто добавляем координату по оси *z* к уравнению Пифагора:

```
|a| = sqrt(a.x x a.x + a.y x a.y + a.z x a.z)
```

Основываясь на этом, мы можем нормализовать векторы к единице измерения длины:

```
a' = (a.x / |a|, a.y / |a|, a.z / |a|)
```

Все интерпретации векторов, рассмотренные нами в главе 8, также применимы и к 3D.

- Позиции записываются в виде нормальных координат вектора *x*, *y* и *z*.
- Скорости и ускорения также представляются как 3D-векторы. Каждый компонент является определенным количеством атрибута по одной оси, например метры в секунду для скорости или метры на секунду в квадрате, если речь идет об ускорении.
- Направления (оси) можно представить как простые 3D-векторы. Мы уже делали это в главе 8, когда использовали возможности поворота в OpenGL ES.

- Расстояния можно измерить следующим образом — из координат конца вектора нужно вычесть координаты начала, а затем измерить результирующую длину вектора.

Еще одна операция, которая может оказаться полезной, — вращение 3D-вектора вокруг 3D-осей. Ранее мы уже использовали этот принцип с помощью метода OpenGL ES `glRotatef()`. Однако нельзя применять этот принцип для того, чтобы повернуть вектор, содержащий позиции или направления игровых объектов, поскольку он работает только для вершин, которые мы отправляем в GPU. К счастью, в Android API есть класс `Matrix`, позволяющий эмулировать воздействие OpenGL ES на GPU. Напишем класс `Vector3`, реализующий все эти особенности. Его код приведен в листинге 11.1, я буду объяснять его по мере необходимости.

Листинг 11.1. Класс `Vector3.java`, аналог класса `Vector` в 3D

```
package com.badlogic.androidgames.framework.math;

import android.opengl.Matrix;
import android.util.FloatMath;

public class Vector3 {
    private static final float[] matrix = new float[16];
    private static final float[] inVec = new float[4];
    private static final float[] outVec = new float[4];
    public float x, y, z;
```

Класс начинается с объявления нескольких статических массивов чисел с плавающей точкой. Они понадобятся далее для реализации нового метода `rotate()` класса `Vector3`. Просто запомните, что член `matrix` содержит 16 элементов, а члены `inVec` и `outVec` — 4 элемента.

Члены `x`, `y` и `z`, определенные далее, говорят сами за себя. Они хранят компоненты вектора:

```
    public Vector3() {
    }

    public Vector3(float x, float y, float z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public Vector3(Vector3 other) {
        this.x = other.x;
        this.y = other.y;
        this.z = other.z;
    }

    public Vector3 cpy() {
```

```
    return new Vector3(x, y, z);
    }

    public Vector3 set(float x, float y, float z) {
        this.x = x;
        this.y = y;
        this.z = z;
        return this;
    }

    public Vector3 set(Vector3 other) {
        this.x = other.x;
        this.y = other.y;
        this.z = other.z;
        return this;
    }
```

Как и класс Vector2, наш класс Vector3 имеет несколько конструкторов и методов, устанавливающих значения, а также метод cpy(), поэтому мы можем с легкостью копировать векторы или задавать их значения, используя компоненты, рассчитанные в программе.

```
    public Vector3 add(float x, float y, float z) {
        this.x += x;
        this.y += y;
        this.z += z;
        return this;
    }

    public Vector3 add(Vector3 other) {
        this.x += other.x;
        this.y += other.y;
        this.z += other.z;
        return this;
    }

    public Vector3 sub(float x, float y, float z) {
        this.x -= x;
        this.y -= y;
        this.z -= z;
        return this;
    }

    public Vector3 sub(Vector3 other) {
        this.x -= other.x;
        this.y -= other.y;
        this.z -= other.z;
        return this;
    }

    public Vector3 mul(float scalar) {
        this.x *= scalar;
```

```
        this.y *= scalar;
        this.z *= scalar;
        return this;
    }
```

Разнообразные методы `add()`, `sub()` и `mul()` являются расширением одноименных методов, которые присутствовали в классе `Vector2`. Добавлена всего одна новая координата — `z`. Данные методы реализуют те особенности, которые мы рассмотрели несколько страниц назад. Довольно прямолинейно, не правда ли?

```
    public float len() {
        return FloatMath.sqrt(x * x + y * y + z * z);
    }

    public Vector3 nor() {
        float len = len();
        if (len != 0) {
            this.x /= len;
            this.y /= len;
            this.z /= len;
        }
        return this;
    }
```

Методы `len()` и `nor()` также практически полностью копируют одноименные методы класса `Vector2`. Все, что мы сейчас сделали, — добавили новую координату в расчеты.

```
    public Vector3 rotate(float angle, float axisX, float axisY,
                          float axisZ) {
        inVec[0] = x;
        inVec[1] = y;
        inVec[2] = z;
        inVec[3] = 1;
        Matrix.setIdentityM(matrix, 0);
        Matrix.rotateM(matrix, 0, angle, axisX, axisY, axisZ);
        Matrix.multiplyMV(outVec, 0, matrix, 0, inVec, 0);
        x = outVec[0];
        y = outVec[1];
        z = outVec[2];
        return this;
    }
```

А вот и новый метод `rotate()`. Как было отмечено ранее, он использует класс `Matrix` из Android API. Этот класс состоит в основном из нескольких статических методов наподобие `Matrix.setIdentityM()` или `Matrix.rotateM()`. Они работают с массивами чисел с плавающей точкой наподобие тех, что мы определили ранее. Матрица хранится как 16 чисел с плавающей точкой, а вектор должен иметь четыре элемента. Я не буду вдаваться в подробности, описывая внутреннюю работу класса. Все, что нам нужно, — это способ эмулировать возможности матрицы OpenGL ES с помощью Java. Данный класс предлагает именно то, что нам необходимо. Все его методы работают с матрицами и делают это так же, как и методы OpenGL ES `glRotatef()`, `glTranslatef()` или `glIdentityf()`.

Этот метод начинается с установки компонента вектора, используется массив inVec, определенный нами ранее. Далее вызывается метод Matrix.setIdentityM() для члена matrix нашего класса. Это его очистит. С помощью OpenGL ES мы используем метод glIdentityf(), чтобы проделать то же самое с матрицами, остающимися в GPU. Далее вызывается метод Matrix.rotateM(). В качестве аргумента он принимает массив чисел с плавающей точкой, смещение для этого массива, угол в градусах, на который необходимо повернуть вектор, а также ось, вокруг которой следует осуществить поворот. Этот метод эквивалентен методу glRotatef(). Он умножит заданную матрицу на матрицу поворота. Наконец, мы вызываем метод Matrix.multiplyMV(), который умножит вектор, хранящийся в массиве inVec, на matrix. Это применит к вектору все преобразования, хранимые в члене matrix. Результат будет помещен в outVec. Остаток метода просто получает новые компоненты вектора из массива outVec и сохраняет их в членах класса Vector3.

ПРИМЕЧАНИЕ

Вы можете использовать класс Matrix не только для того, чтобы поворачивать векторы. Он работает с переданными матрицами точно так же, как и OpenGL ES.

```
    public float dist(Vector3 other) {
        float distX = this.x – other.x;
        float distY = this.y – other.y;
        float distZ = this.z – other.z;
        return FloatMath.sqrt(distX * distX + distY * distY + distZ *
                              distZ);
    }

    public float dist(float x, float y, float z) {
        float distX = this.x – x;
        float distY = this.y – y;
        float distZ = this.z – z;
        return FloatMath.sqrt(distX * distX + distY * distY + distZ *
                              distZ);
    }

    public float distSquared(Vector3 other) {
        float distX = this.x – other.x;
        float distY = this.y – other.y;
        float distZ = this.z – other.z;
        return distX * distX + distY * distY + distZ * distZ;
    }

    public float distSquared(float x, float y, float z) {
        float distX = this.x – x;
        float distY = this.y – y;
        float distZ = this.z – z;
        return distX * distX + distY * distY + distZ * distZ;
    }
}
```

И наконец, мы видим привычные методы `dist()` и `distSquared()`, рассчитывающие расстояние между двумя векторами в 3D.

Обратите внимание, метод `angle()` я скопировал без изменений из класса `Vector2`. В то время, как возможно измерить угол между двумя векторами в 3D, мы все равно не получим значение в промежутке от 0 до 360. Обычно мы измеряем угол между двумя векторами на плоскостях *xy*, *zy* и *xz*, используя только два компонента каждого вектора и применяя метод `Vector2.angle()`. Такая функциональность не нужна в нашей последней игре, поэтому вернемся к главной теме главы.

Думаю, вы согласитесь, что нам не нужно рассматривать пример использования этого класса. Мы можем просто вызвать его так же, как мы это делали с классом `Vector2` в главе 8. Перейдем к следующей теме: освещение в OpenGL ES.

Освещение в OpenGL ES

Освещение в OpenGL ES — это полезная особенность, которая может придать 3D-играм приятный оттенок. Чтобы использовать подобную функциональность, сначала нам необходимо понять модель освещения OpenGL ES.

Как работает освещение

Задумаемся о том, как работает освещение. Для начала нам потребуется источник света, испускающий свет. Понадобится также освещаемый объект. Наконец, нам также понадобится сенсор вроде глаз или камеры, принимающий фотоны, которые посылаются источником света и отражаемые объектом. Освещение меняет воспринимаемый цвет объекта в зависимости от:

- типа источника освещения;
- цвета или интенсивности источника света;
- позиции источника света и его направления относительно освещаемого объекта;
- материала и текстуры объекта.

Интенсивность, с которой свет отражается объектом, зависит от множества факторов. Самый главный фактор, на который мы обращаем внимание, — это угол, с которым световой луч падает на поверхность. Чем ближе этот угол к прямому, тем больше интенсивность, с которой свет отразится от объекта. Это проиллюстрировано на рис. 11.1.

Как только световой луч упадет на поверхность, он отразится в двух различных направлениях. Большая часть света отразится рассеянно. Это означает, что отраженные световые лучи неравномерно «рассыпаны» в случайном порядке по поверхности объекта. Некоторые лучи отражаются зеркально. Это означает, что световые лучи отразятся назад так, будто они падают на идеальное зеркало. На рис. 11.2 показана разница между рассеянным и зеркальным отражениями.

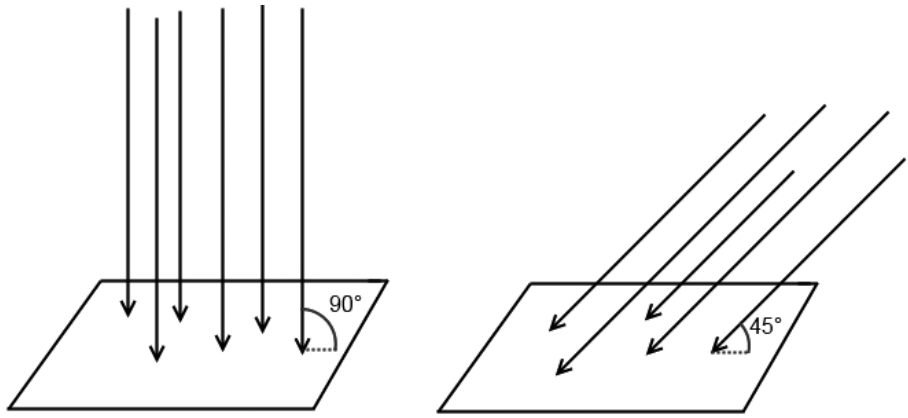

Рис. 11.1. Чем ближе угол к прямому, тем больше интенсивность отраженного света

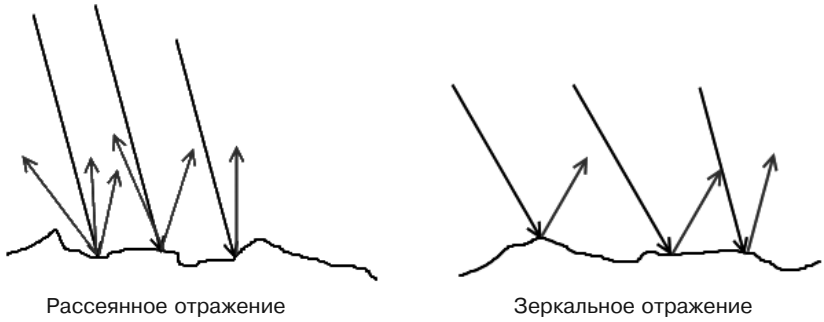

Рис. 11.2. Рассеянное и зеркальное отражения

Зеркальное отражение будет проявляться как блики на объектах. Будет ли свет отражаться от объекта зеркально, зависит от материала, из которого он сделан. Объекты с неровной или шероховатой, как кожа, поверхностью, скорее всего, не будут иметь зеркальных бликов. Объекты, имеющие гладкую поверхность наподобие стекла или мрамора, будут демонстрировать эти световые артефакты. Конечно, стекло или мрамор не являются идеально гладкими, но по сравнению с деревом или человеческой кожей они являются таковыми.

Когда свет падает на поверхность, его отражение также меняет свой цвет в зависимости от химического состава освещаемого объекта. Объекты, которые кажутся нам красными, отражают только красные «порции» света и поглощают все другие частоты. Черный объект — это такой объект, который поглощает практически весь свет, который падает на него.

OpenGL ES позволяет имитировать реальное поведение, определяя источники света и материалы объектов.

Источники освещения

Нас окружает множество разнообразных источников освещения. Солнце постоянно посылает свои фотоны. Мониторы излучают свет, окружающий нас приятным свечением по ночам. Лампочки и фары помогают нам избегать столкновений с различными предметами в темноте. OpenGL ES позволяет создавать четыре типа источников света.

- *Подсветка*. Является сама по себе не источником света, а результатом появления фотонов от других световых источников. Вместе эти случайные фотоны создают некоторый постоянный уровень освещения, не имеющий направления и освещающий все объекты одинаково.
- *Точечные источники света*. Имеют позицию в пространстве и испускают свет во всех направлениях. Например, точечным источником света является лампочка.
- *Направленные источники освещения*. Выражаются как направления в OpenGL ES. Предполагается, что они находятся бесконечно далеко. В идеале Солнце может являться таким источником. Мы можем предположить, что все световые лучи, исходящие от Солнца, попадают на Землю под одинаковым углом из-за расстояния между Землей и Солнцем.
- *Светильники*. Эти источники похожи на точечные источники освещения тем, что имеют заданную позицию в пространстве. Кроме того, у них есть направление, в котором они излучают световые лучи. Они создают световой конус, ограниченный некоторым радиусом. Примером такого источника света является уличный фонарь.

Мы будем рассматривать только подсветку, а также точечные и направленные источники света. Светильники часто сложно использовать на ограниченных GPU Android-устройств из-за способа расчета освещения в OpenGL ES. Скоро вы поймете, почему это так.

Помимо позиции и направления источника света OpenGL ES позволяет определять цвет или интенсивность света. Эти характеристики выражаются с помощью цвета RGBA. Однако OpenGL ES требует определять четыре различных цвета для одного источника вместо одного.

- Подсветка — интенсивность/цвет, вносящий вклад в создание затенения объекта. Объект будет освещен одинаково со всех сторон, независимо от его позиции или ориентации относительно источника света.
- Рассеянный — интенсивность/цвет света, которым будет освещен объект после расчета рассеянного отражения. Грани объекта, которые не «смотрят» на источник света, не будут освещены, как и в реальной жизни.
- Зеркальный — интенсивность/цвет, похожий на рассеянный цвет. Однако он влияет только на те точки объекта, которые имеют определенную ориентацию по отношению к источнику света и сенсору.

- Эмиссивный — очень сложный расчет цвета, имеющий чрезвычайно ограниченное применение в приложениях с физикой реального мира, поэтому мы не будем его рассматривать.

Чаще всего будем применять рассеянные и зеркальные интенсивности источника света, а двум другим укажем значения по умолчанию. Кроме того, большую часть времени будем использовать одинаковый цвет RGBA как для рассеянной, так и для зеркальной интенсивности.

Материалы

Все объекты в нашем мире состоят из какого-либо материала. Каждый материал определяет, как свет, падающий на объект, будет отражаться и изменять цвет отраженного света. OpenGL ES позволяет определять те же четыре цвета RGBA для материала, что и для источника света.

- Подсветка — цвет, который объединяется с фоновым цветом любого источника света на сцене.
- Рассеянный — цвет, который объединяется с рассеянным цветом любого источника света.
- Зеркальный — цвет, который объединяется с зеркальным цветом любого источника света. Он используется для создания бликов на поверхности объекта.
- Эмиссивный — продолжаем игнорировать этот тип цвета, поскольку он практически не применяется в нашем контексте.

Рисунок 11.3 иллюстрирует первые три типа свойств материала/источника света: подсветка, рассеянный и зеркальный.

Рис. 11.3. Различные типы материалов/источников света: только подсветка *(слева)*, только рассеянный *(посередине)*, подсветка и рассеянный цвет с зеркальными бликами *(справа)*

На рис. 11.3 показано влияние различных свойств материалов и источников света на цвет. Подсветка освещает размер равномерно. Рассеянный свет отразится в зависимости от угла, под которым на объект падают световые лучи; площади, которые непосредственно повернуты к источнику света, будут освещены ярче, площади, до которых свет не может добраться, будут темными. На правом изображении вы можете увидеть комбинацию подсветки, рассеянного и зеркального света. Зеркальный свет проявляет себя как белые блики на сфере.

Как OpenGL ES рассчитывает освещение: нормали вершин

Вы знаете, что интенсивность света, отраженного от объекта, зависит от его угла падения на объект. OpenGL ES использует этот факт для расчета освещения. Он применяет для этого нормали вершин, которые необходимо определять в коде так же, как и координаты текстур или цвета вершин. На рис. 11.4 показана сфера и ее нормали вершин.

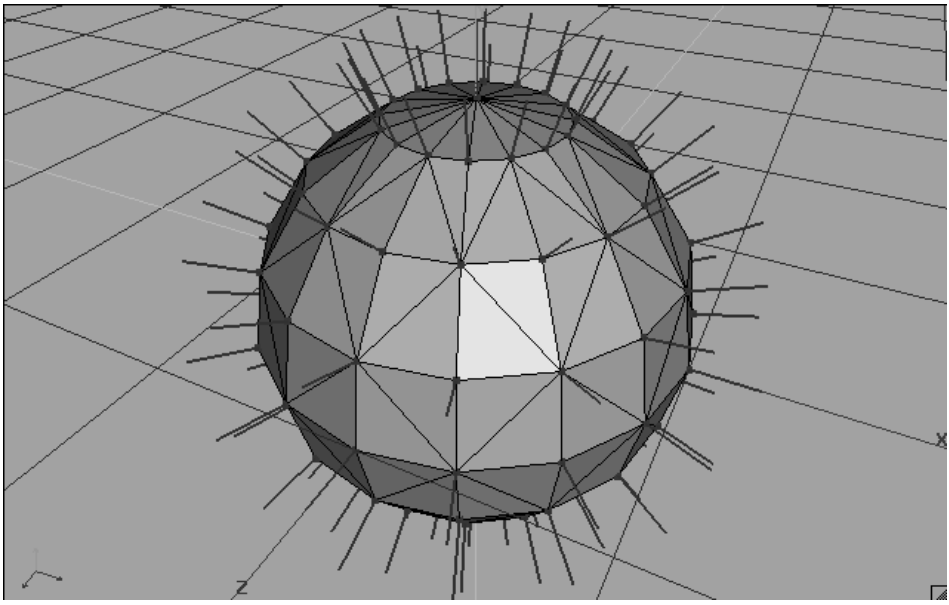

Рис. 11.4. Сфера и ее нормали вершин

Нормали — это единичные векторы, указывающие направление, к которому повернута поверхность. В нашем случае поверхность — это треугольник. Вместо определения нормали поверхности мы определяем нормаль вершины. Разница между этими нормалями заключается в том, что нормаль вершины может не указывать в ту же сторону, что и нормаль поверхности. Это четко видно на рис. 11.4, где каждая нормаль вершины является усредненной нормалью всех треугольников, к которым принадлежит вершина. Такое усреднение производится для создания гладкой затененности объекта.

При отрисовке объекта с использованием освещения и нормалей вершин OpenGL ES определит угол между каждой вершиной и источником света. Если он знает этот угол, то может рассчитать цвет вершины, основываясь на свойствах материала. Конечным результатом является цвет каждой вершины, который далее применяется к каждому треугольнику в комбинации с рассчитанными цветами

других вершин. Этот использованный цвет будет объединен с любыми текстурными преобразованиями, которые мы применим к объекту.

Это звучит довольно пугающе, но на самом деле не все так плохо. Нам нужно разрешить использование освещения и определить источники освещения, материал отрисовываемого объекта и нормали вершин (в дополнение к параметрам вершин, которые мы обычно определяем, например позицию или координаты текстур). Рассмотрим, как это можно реализовать с помощью OpenGL ES.

На практике

Теперь выполним все действия, необходимые для того, чтобы работать с освещением с помощью OpenGL ES. Создадим несколько небольших вспомогательных классов, которые немного упростят работу с источниками света, и поместим их в пакет `com.badlogic.androidgames.framework.gl`.

Разрешение и запрещение освещения

Как и для прочих состояний OpenGL ES, сначала следует подключить названную функциональность. Это можно сделать следующим образом:

```
gl.glEnable(GL10.GL_LIGHTING);
```

После этого освещение будет применено ко всем отрисовываемым объектам. Чтобы получить результат, необходимо определить источники света и материалы, а также нормали вершин. Как только мы закончим отрисовывать все необходимые объекты, освещение можно отключить:

```
gl.glDisable(GL10.GL_LIGHTING);
```

Определение источников освещения

OpenGL ES предоставляет 4 типа источников освещения: подсветка, точечный, направленный и светильник. Рассмотрим, как определить первые три. Чтобы светильники были эффективными и хорошо выглядели, каждая модель должна состоять из огромного количества треугольников. Для множества теперешних мобильных устройств это невозможно.

OpenGL ES позволяет определять максимум 8 источников освещения одновременно, а также один глобальный источник подсветки. Каждый из 8 источников освещения имеет идентификатор, от `GL10.GL_LIGHT0` до `GL10.GL_LIGHT7`. Если нужно изменить свойства одного из этих источников освещения, это можно сделать, определив соответствующий ему ID.

Разрешить использование источников освещения можно с помощью следующего синтаксиса:

```
gl.glEnable(GL10.GL_LIGHT0);
```

Далее OpenGL ES получит свойства этого источника освещения и применит их ко всем отрисовываемым объектам. Если нам нужно запретить использова-

ние источника освещения, мы можем сделать это с помощью следующего утверждения:

```
gl.glDisable(GL10.GL_LIGHT0);
```

Подсветка — это особый случай, поскольку у нее нет идентификатора. На сцене OpenGL ES может существовать только одна подсветка. Рассмотрим этот источник освещения подробнее.

Подсветка

Подсветка — это особый тип освещения. У него нет позиции или направления, только цвет, который применяется ко всем освещаемым объектам одинаково. OpenGL ES позволяет определять глобальную подсветку следующим образом:

```
float[] ambientColor = { 0.2f, 0.2f, 0.2f, 1.0f };
gl.glLightModelfv(GL10.GL_LIGHT_MODEL_AMBIENT, color, 0);
```

Массив `ambientColor` содержит значения RGBA цвета подсветки, представленные как числа с плавающей точкой в диапазоне от 0 до 1. Метод `glLightModelfv()` принимает в качестве первого параметра константу, определяющую, что мы хотим установить цвет источника фонового освещения, массив чисел с плавающей точкой, который содержит цвет источника, и смещение для массива чисел с плавающей точкой, из которого метод начнет считывать значения RGBA. Поместим код, решающий эту задачу, в небольшой класс. Его код приведен в листинге 11.2.

Листинг 11.2. Класс AmbientLight.java, простая абстракция глобальной подсветки OpenGL ES

```
package com.badlogic.androidgames.framework.gl;

import javax.microedition.khronos.opengles.GL10;

public class AmbientLight {
    float[] color = {0.2f, 0.2f, 0.2f, 1};

    public void setColor(float r, float g, float b, float a) {
        color[0] = r;
        color[1] = g;
        color[2] = b;
        color[3] = a;
    }

    public void enable(GL10 gl) {
        gl.glLightModelfv(GL10.GL_LIGHT_MODEL_AMBIENT, color, 0);
    }
}
```

Все, что мы делаем, — сохраняем цвет подсветки в массиве чисел с плавающей точкой и предоставляем два метода: один из них используется для установки цвета, а другой, чтобы указать OpenGL ES, что использовать следует именно этот цвет. По умолчанию применяется серый цвет.

Точечные источники освещения

Точечные источники освещения имеют позицию, а также фоновые, рассеянные и зеркальные цвет/интенсивность (мы не рассматриваем эмиссивные цвет/интенсивность). Определить разные типы цветов можно следующим образом:

```
gl.glLightfv(GL10.GL_LIGHT3, GL10.GL_AMBIENT, ambientColor, 0);
gl.glLightfv(GL10.GL_LIGHT3, GL10.GL_DIFFUSE, diffuseColor, 0);
gl.glLightfv(GL10.GL_LIGHT3, GL10.GL_SPECULAR, specularColor, 0);
```

Первый параметр — это идентификатор источника света. В этом случае мы используем четвертый источник. Следующий параметр определяет атрибут, который мы хотим изменить. Третий параметр — это массив чисел с плавающей точкой, содержащий значения RGBA, а последний — это смещение в данном массиве. Определить позицию источника так же просто:

```
float[] position = {x, y, z, 1};
gl.glLightfv(GL10.GL_LIGHT3, GL10.GL_POSITION, position, 0);
```

Мы снова определяем атрибут, который хотим изменить (в данном случае позицию), массив из четырех элементов содержит *x*-, *y*- и *z*-координату источника света в создаваемом мире. Обратите внимание, четвертый элемент массива должен быть равен единице, если источник света имеет позицию! Поместим это во вспомогательный класс. Его код содержится в листинге 11.3.

Листинг 11.3. Класс PointLight.java, простая абстракция точечных источников света OpenGL ES

```
package com.badlogic.androidgames.framework.gl;

import javax.microedition.khronos.opengles.GL10;

public class PointLight {
    float[] ambient = { 0.2f, 0.2f, 0.2f, 1.0f };
    float[] diffuse = { 1.0f, 1.0f, 1.0f, 1.0f };
    float[] specular = { 0.0f, 0.0f, 0.0f, 1.0f };
    float[] position = { 0, 0, 0, 1 };
    int lastLightId = 0;
    public void setAmbient(float r, float g, float b, float a) {
        ambient[0] = r;
        ambient[1] = g;
        ambient[2] = b;
        ambient[3] = a;
    }

    public void setDiffuse(float r, float g, float b, float a) {
        diffuse[0] = r;
        diffuse[1] = g;
        diffuse[2] = b;
        diffuse[3] = a;
    }

    public void setSpecular(float r, float g, float b, float a) {
```

```
        specular[0] = r;
        specular[1] = g;
        specular[2] = b;
        specular[3] = a;
    }

    public void setPosition(float x, float y, float z) {
        position[0] = x;
        position[1] = y;
        position[2] = z;
    }

    public void enable(GL10 gl, int lightId) {
        gl.glEnable(lightId);
        gl.glLightfv(lightId, GL10.GL_AMBIENT, ambient, 0);
        gl.glLightfv(lightId, GL10.GL_DIFFUSE, diffuse, 0);
        gl.glLightfv(lightId, GL10.GL_SPECULAR, specular, 0);
        gl.glLightfv(lightId, GL10.GL_POSITION, position, 0);
        lastLightId = lightId;
    }

    public void disable(GL10 gl) {
        gl.glDisable(lastLightId);
    }
}
```

Наш вспомогательный класс содержит фоновые, рассеянные и зеркальные цветовые компоненты света, а также позицию (четвертый элемент равен единице). В дополнение мы храним последний идентификатор, используемый для данного источника, поэтому становится возможно создать метод `disable()`, который отключит свет при необходимости. Также у нас есть метод `enable()`, который принимает экземпляр класса `GL10` и идентификатор источника света (например `GL10.GL_LIGHT6`). Он разрешает использование освещения, устанавливает его атрибуты и сохраняет использованный идентификатор. Метод `disable()` просто запрещает использование освещения, используя член класса `lastLightId`, установленный в методе `enable()`.

Мы используем разумные значения по умолчанию для фонового, рассеянного и зеркального цветов при инициализации массивов-членов класса. Свет будет белым и не будет создавать никаких бликов, поскольку его зеркальная составляющая черная.

Направленные источники света

Направленные источники света практически идентичны точечным. Единственное различие заключается в том, что они имеют направление вместо позиции. Способ выражения направления несколько запутан. Вместо использования вектора, указывающего направление, OpenGL ES ожидает, что мы определим одну точку. Затем направление будет определено с помощью вектора, соединяющего эту точку

и начало координат. Следующий сниппет позволяет создать направленный источник света, исходящий с правой стороны мира:

```
float[] dirPos = {1, 0, 0, 0};
gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_POSITION, dirPos, 0);
```

Мы можем преобразовать его в вектор:

```
dir = -dirPos = {-1, 0, 0, 0}
```

Остальные атрибуты, вроде фонового или рассеянного цвета, идентичны аналогичным атрибутам точечного источника света. В листинге 11.4 показан код небольшого вспомогательного класса, использующегося для создания направленных источников света.

Листинг 11.4. Класс DirectionLight.java, простая абстракция направленных источников света в OpenGL ES

```
package com.badlogic.androidgames.framework.gl;

import javax.microedition.khronos.opengles.GL10;

public class DirectionalLight {
    float[] ambient = { 0.2f, 0.2f, 0.2f, 1.0f };
    float[] diffuse = { 1.0f, 1.0f, 1.0f, 1.0f };
    float[] specular = { 0.0f, 0.0f, 0.0f, 1.0f };
    float[] direction = { 0, 0, -1, 0 };
    int lastLightId = 0;

    public void setAmbient(float r, float g, float b, float a) {
        ambient[0] = r;
        ambient[1] = g;
        ambient[2] = b;
        ambient[3] = a;
    }

    public void setDiffuse(float r, float g, float b, float a) {
        diffuse[0] = r;
        diffuse[1] = g;
        diffuse[2] = b;
        diffuse[3] = a;
    }

    public void setSpecular(float r, float g, float b, float a) {
        specular[0] = r;
        specular[1] = g;
        specular[2] = b;
        specular[3] = a;
    }

    public void setDirection(float x, float y, float z) {
        direction[0] = -x;
```

```
        direction[1] = -y;
        direction[2] = -z;
    }

    public void enable(GL10 gl, int lightId) {
        gl.glEnable(lightId);
        gl.glLightfv(lightId, GL10.GL_AMBIENT, ambient, 0);
        gl.glLightfv(lightId, GL10.GL_DIFFUSE, diffuse, 0);
        gl.glLightfv(lightId, GL10.GL_SPECULAR, specular, 0);
        gl.glLightfv(lightId, GL10.GL_POSITION, direction, 0);
        lastLightId = lightId;
    }

    public void disable(GL10 gl) {
        gl.glDisable(lastLightId);
    }
}
```

Этот вспомогательный класс практически идентичен классу `PointLight`. Единственное различие заключается в том, что в массиве `direction` четвертый элемент равен единице. Кроме того, вместо метода `setPosition()` появился метод `setDirection()`. Он позволяет определять направление, например так: (–1; 0; 0), в этом случае свет будет исходить с правой стороны. Внутри метода все компоненты вектора меняют свой знак, таким образом мы преобразовываем направление к формату, ожидаемому OpenGL ES.

Определяем материалы

Материал определяется несколькими атрибутами. Как и в случае с любыми другими «объектами» OpenGL ES, материал — это состояние, которое будет активно до тех пор, пока мы не изменим его снова или пока не потеряется контекст OpenGL ES. Чтобы установить текущие атрибуты материалов, мы можем сделать следующее:

```
gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_AMBIENT, ambientColor, 0);
gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_DIFFUSE, diffuseColor, 0);
gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SPECULAR, specularColor, 0);
```

Как и обычно, нам необходимо определить фоновый, рассеянный и зеркальный RGBA-цвета. Это можно сделать так же, как и ранее, — с помощью массивов чисел с плавающей точкой, состоящих из четырех элементов.

Объединить эти действия в один вспомогательный класс очень просто. Результат вы можете увидеть в листинге 11.5.

Листинг 11.5. Класс Material.java, простая абстракция материалов OpenGL ES

```
package com.badlogic.androidgames.framework.gl;

import javax.microedition.khronos.opengles.GL10;

public class Material {
```

```
    float[] ambient = { 0.2f, 0.2f, 0.2f, 1.0f };
    float[] diffuse = { 1.0f, 1.0f, 1.0f, 1.0f };
    float[] specular = { 0.0f, 0.0f, 0.0f, 1.0f };

    public void setAmbient(float r, float g, float b, float a) {
        ambient[0] = r;
        ambient[1] = g;
        ambient[2] = b;
        ambient[3] = a;
    }

    public void setDiffuse(float r, float g, float b, float a) {
        diffuse[0] = r;
        diffuse[1] = g;
        diffuse[2] = b;
        diffuse[3] = a;
    }

    public void setSpecular(float r, float g, float b, float a) {
        specular[0] = r;
        specular[1] = g;
        specular[2] = b;
        specular[3] = a;
    }

    public void enable(GL10 gl) {
        gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_AMBIENT, ambient, 0);
        gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_DIFFUSE, diffuse, 0);
        gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SPECULAR, specular, 0);
    }
}
```

Здесь тоже нет ничего удивительного. Мы просто сохраняем три компонента, описывающих материал, а также предоставляем функции для установки их значений и метод `enable()`, которые передают их OpenGL ES.

У OpenGL ES есть еще один козырь в рукаве, когда речь идет о материалах. Обычно он вместо метода `glMaterialfv()` использует нечто, называемое цветом материала. Это означает, что вместо фонового и рассеянного цветов, определяемых методом `glMaterialfv()`, **OpenGL ES примет цвет вершин наших моделей в качестве фонового и рассеянного цветов материала.** Чтобы разрешить использование этой особенности, необходимо просто вызвать ее:

```
gl.glEnable(GL10.GL_COLOR_MATERIAL);
```

Обычно я именно так и поступаю, потому что фоновый и рассеянный цвета часто одинаковы. Поскольку я не использую зеркальные блики в большинстве моих игр и демонстраций, я вполне могу применять такой способ и совсем не вызывать метод `glMaterialfv()`. Какой способ задействовать вам — решаете только вы.

Определяем нормали

Чтобы в OpenGL ES работало освещение, необходимо определить нормали вершин для каждой вершины модели. Нормаль вершины должна представлять собой единичный вектор, указывающий (обычно) в ту сторону, в которую повернута поверхность, к которой принадлежит вершина. На рис. 11.5 проиллюстрированы нормали вершин для нашего куба.

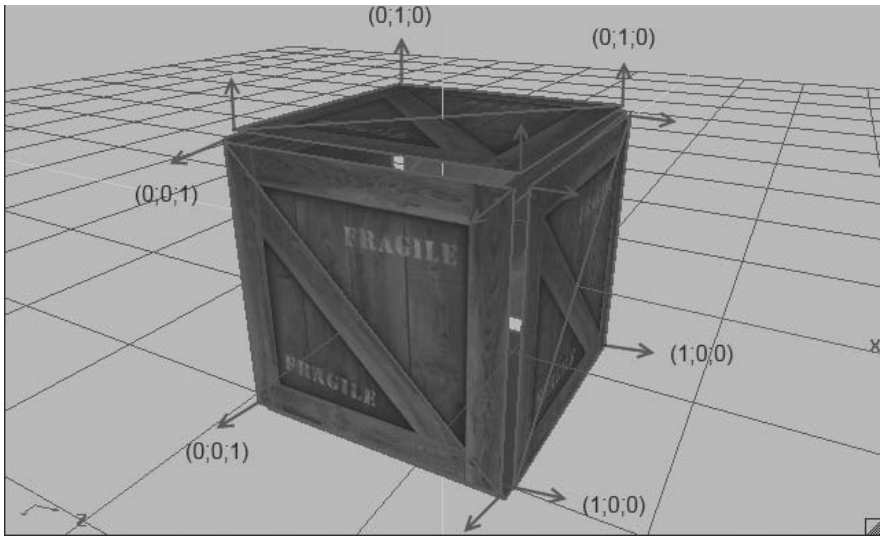

Рис. 11.5. Нормали вершин для каждой вершины нашего куба

Нормаль вершины — это еще один атрибут вершины, такой же, как позиция или цвет. Чтобы воспользоваться нормалями вершин, нам необходимо еще раз изменить класс `Vertices3`. Для того чтобы указать OpenGL ES, где он может найти нормали для каждой вершины, мы будем использовать метод `glNormalPointer()`, точно так же, как мы ранее применяли методы `glVertexPointer()` или `glColorPointer()`. В листинге 11.6 показана финальная версия класса `Vertices3`.

Листинг 11.6. Класс `Vertices3.java`, финальная версия, поддерживающая нормали

```
package com.badlogic.androidgames.framework.gl;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.IntBuffer;
import java.nio.ShortBuffer;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.impl.GLGraphics;

public class Vertices3 {
    final GLGraphics glGraphics;
```

```
    final boolean hasColor;
    final boolean hasTexCoords;
    final boolean hasNormals;
    final int vertexSize;
    final IntBuffer vertices;
    final int[] tmpBuffer;
    final ShortBuffer indices;
```

В классе появился новый член hasNormals, отслеживающий, имеют ли вершины нормали.

```
    public Vertices3(GLGraphics glGraphics, int maxVertices, int maxIndices,
            boolean hasColor, boolean hasTexCoords, boolean hasNormals) {
        this.glGraphics = glGraphics;
        this.hasColor = hasColor;
        this.hasTexCoords = hasTexCoords;
        this.hasNormals = hasNormals;
        this.vertexSize = (3 + (hasColor ? 4 : 0) + (hasTexCoords ? 2 : 0) + (has-
Normals ? 3    : 0)) * 4;
        this.tmpBuffer = new int[maxVertices * vertexSize / 4];

        ByteBuffer buffer = ByteBuffer.allocateDirect(maxVertices * vertexSize);
        buffer.order(ByteOrder.nativeOrder());
        vertices = buffer.asIntBuffer();

        if (maxIndices > 0) {
            buffer = ByteBuffer.allocateDirect(maxIndices * Short.SIZE / 8);
            buffer.order(ByteOrder.nativeOrder());
            indices = buffer.asShortBuffer();
        } else {
            indices = null;
        }
    }
```

Конструктор теперь принимает также параметр hasNormals. Нам еще необходимо модифицировать расчет члена vertexSize, добавив три числа с плавающей точкой на каждую вершину там, где это возможно.

```
    public void setVertices(float[] vertices, int offset, int length) {
        this.vertices.clear();
        int len = offset + length;
        for (int i = offset, j = 0; i < len; i++, j++)
            tmpBuffer[j] = Float.floatToRawIntBits(vertices[i]);
        this.vertices.put(tmpBuffer, 0, length);
        this.vertices.flip();
    }

    public void setIndices(short[] indices, int offset, int length) {
        this.indices.clear();
        this.indices.put(indices, offset, length);
        this.indices.flip();
    }
```

Как вы можете видеть, методы setVertices() и setIndices() остаются без изменений.

```
public void bind() {
    GL10 gl = glGraphics.getGL();

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    vertices.position(0);
    gl.glVertexPointer(3, GL10.GL_FLOAT, vertexSize, vertices);

    if (hasColor) {
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
        vertices.position(3);
        gl.glColorPointer(4, GL10.GL_FLOAT, vertexSize, vertices);
    }

    if (hasTexCoords) {
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        vertices.position(hasColor ? 7 : 3);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, vertexSize, vertices);
    }

    if (hasNormals) {
        gl.glEnableClientState(GL10.GL_NORMAL_ARRAY);
        int offset = 3;
        if (hasColor)
            offset += 4;
        if (hasTexCoords)
            offset += 2;
        vertices.position(offset);
        gl.glNormalPointer(GL10.GL_FLOAT, vertexSize, vertices);
    }
}
```

В только что продемонстрированном методе bind() используем те же приемы с буфером ByteBuffer, что и ранее, но в этот раз добавляем нормали с помощью метода glNormalPointer(). Для вычисления смещения указателя нормали необходимо принять в расчет то, заданы ли координаты текстур и цвета.

```
public void draw(int primitiveType, int offset, int numVertices) {
    GL10 gl = glGraphics.getGL();

    if (indices != null) {
        indices.position(offset);
        gl.glDrawElements(primitiveType, numVertices,
                GL10.GL_UNSIGNED_SHORT, indices);
    } else {
        gl.glDrawArrays(primitiveType, offset, numVertices);
    }
}
```

Как вы можете видеть, метод `draw()` также не изменился; все действо происходит в методе `bind()`.

```
    public void unbind() {
        GL10 gl = glGraphics.getGL();
        if (hasTexCoords)
            gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

        if (hasColor)
            gl.glDisableClientState(GL10.GL_COLOR_ARRAY);

        if (hasNormals)
            gl.glDisableClientState(GL10.GL_NORMAL_ARRAY);
    }
}
```

Наконец, мы несколько изменяем метод `unbind()`. Запрещаем использование указателей нормали, если таковые имелись, соответственно очищая состояние OpenGL ES.

Применить измененный класс `Vertices3` так же просто, как и ранее. Рассмотрим небольшой пример:

```
float[] vertices = { -0.5f, -0.5f, 0, 0, 0, 1,
                      0.5f, -0.5f, 0, 0, 0, 1,
                      0.0f,  0.5f, 0, 0, 0, 1 };
Vertices3 vertices = new Vertices3(glGraphics, 3, 0, false, false, true);
vertices.setVertices(vertices);
```

Создаем массив чисел с плавающей точкой для хранения трех вершин, каждая из которых имеет позицию (первые три числа в каждой строке) и нормаль (последние три числа в каждой строке). В этом случае мы задаем треугольник в плоскости *xy*, его нормали указывают в направлении положительной части оси *z*.

Все, что нам остается, — создать экземпляр класса `Vertices3` и установить значения вершин. Довольно легко, не правда ли?

Вся работа по привязке, рисованию и отвязке выполняется точно так же, как и в предыдущей версии класса. Как и ранее, мы можем добавить цвета вершин и координаты текстур.

Собираем все воедино

Соберем все вместе. Нам необходимо нарисовать сцену, имеющую глобальную подсветку, точечные и направленные источники света. Они будут освещать куб, расположенный в начале координат. Нам также нужно вызвать метод `gluLookAt()`, чтобы расположить камеру. На рис. 11.6 показан внешний вид нашего мира.

Как и для всех прочих примеров, создадим класс, который будет называться `LightTest`, как обычно расширяющий класс `GLGame`. Он будет возвращать экземпляр класса `LightScreen` с помощью метода `getStartScreen()`. Класс `LightScreen` наследует от класса `GLScreen` (листинг 11.7).

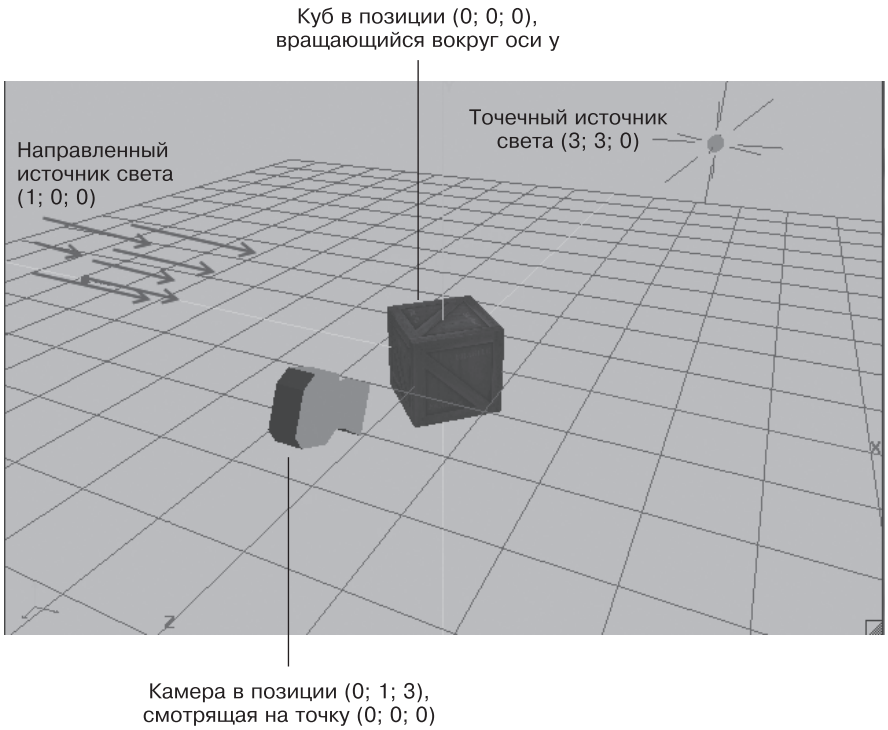

Рис. 11.6. Наша первая освещенная сцена

Листинг 11.7. Фрагменты класса LightTest.java, создание освещения с помощью OpenGL ES

```
class LightScreen extends GLScreen {
    float angle;
    Vertices3 cube;
    Texture texture;
    AmbientLight ambientLight;
    PointLight pointLight;
    DirectionalLight directionalLight;
    Material material;
```

Начнем с описания нескольких членов класса. Член `angle` хранит информацию о текущем угле поворота куба вокруг оси *y*. Член `Vertices3` хранит вершины модели куба, которые мы скоро определим. В дополнение у нас есть экземпляры классов `AmbientLight`, `PointLight` и `DirectionalLight`, а также экземпляр класса `Material`.

```
    public LightScreen(Game game) {
        super(game);

        cube = createCube();
        texture = new Texture(glGame, "crate.png");
```

```
        ambientLight = new AmbientLight();
        ambientLight.setColor(0, 0.2f, 0, 1);
        pointLight = new PointLight();
        pointLight.setDiffuse(1, 0, 0, 1);
        pointLight.setPosition(3, 3, 0);
        directionalLight = new DirectionalLight();
        directionalLight.setDiffuse(0, 0, 1, 1);
        directionalLight.setDirection(1, 0, 0);
        material = new Material();
    }
```

Далее следует конструктор. Здесь создаются вершины модели куба, а также загружается текстура ящика, как и в предыдущей главе. Мы также инициализируем источники освещения и материалы. Цвет подсветки — светло-зеленый. Направленный источник — красного цвета и располагается в точке (3; 3; 0) нашего мира. Направленный источник света имеет синий рассеянный цвет, свет падает слева. Для материала используем значения по умолчанию (несколько фоновый, белый для рассеянной составляющей и черный для зеркальной).

```
    @Override
    public void resume() {
        texture.reload();
    }
```

В методе `resume()` убеждаемся, что наша текстура (пере)загружается, если контекст будет потерян.

```
    private Vertices3 createCube() {
        float[] vertices = { -0.5f, -0.5f, 0.5f, 0, 1, 0, 0, 1,
                              0.5f, -0.5f, 0.5f, 1, 1, 0, 0, 1,
                              0.5f,  0.5f, 0.5f, 1, 0, 0, 0, 1,
                             -0.5f,  0.5f, 0.5f, 0, 0, 0, 0, 1,

                              0.5f, -0.5f,  0.5f, 0, 1, 1, 0, 0,
                              0.5f, -0.5f, -0.5f, 1, 1, 1, 0, 0,
                              0.5f,  0.5f, -0.5f, 1, 0, 1, 0, 0,
                              0.5f,  0.5f,  0.5f, 0, 0, 1, 0, 0,

                              0.5f, -0.5f, -0.5f, 0, 1, 0, 0, -1,
                             -0.5f, -0.5f, -0.5f, 1, 1, 0, 0, -1,
                             -0.5f,  0.5f, -0.5f, 1, 0, 0, 0, -1,
                              0.5f,  0.5f, -0.5f, 0, 0, 0, 0, -1,

                             -0.5f, -0.5f, -0.5f, 0, 1, -1, 0, 0,
                             -0.5f, -0.5f,  0.5f, 1, 1, -1, 0, 0,
                             -0.5f,  0.5f,  0.5f, 1, 0, -1, 0, 0,
                             -0.5f,  0.5f, -0.5f, 0, 0, -1, 0, 0,

                             -0.5f,  0.5f,  0.5f, 0, 1, 0, 1, 0,
                              0.5f,  0.5f,  0.5f, 1, 1, 0, 1, 0,
```

```
                            0.5f,  0.5f, -0.5f, 1, 0, 0, 1, 0,
                           -0.5f,  0.5f, -0.5f, 0, 0, 0, 1, 0,

                           -0.5f, -0.5f, -0.5f, 0, 1, 0, -1, 0,
                            0.5f, -0.5f, -0.5f, 1, 1, 0, -1, 0,
                            0.5f, -0.5f,  0.5f, 1, 0, 0, -1, 0,
                           -0.5f, -0.5f,  0.5f, 0, 0, 0, -1, 0 };
        short[] indices = { 0, 1, 2, 2, 3, 0,
                           4, 5, 6, 6, 7, 4,
                           8, 9, 10, 10, 11, 8,
                           12, 13, 14, 14, 15, 12,
                           16, 17, 18, 18, 19, 16,
                           20, 21, 22, 22, 23, 20,
                           24, 25, 26, 26, 27, 24 };
        Vertices3 cube = new Vertices3(glGraphics, vertices.length / 8,
indices.length, false, true, true);
        cube.setVertices(vertices, 0, vertices.length);
        cube.setIndices(indices, 0, indices.length);
        return cube;
    }
```

Метод `createCube()` практически не изменился с предыдущих примеров. Однако в этот раз мы добавляем нормали для каждой вершины, что показано на рис. 11.5. Помимо этого все остается прежним.

```
    @Override
    public void update(float deltaTime) {
        angle += deltaTime * 20;
    }
```

В методе `update()` просто увеличиваем угол поворота куба.

```
    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glClearColor(0.2f, 0.2f, 0.2f, 1.0f);
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        gl.glEnable(GL10.GL_DEPTH_TEST);
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());

        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        GLU.gluPerspective(gl, 67, glGraphics.getWidth()
                / (float) glGraphics.getHeight(), 0.1f, 10f);
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();
        GLU.gluLookAt(gl, 0, 1, 3, 0, 0, 0, 0, 1, 0);

        gl.glEnable(GL10.GL_LIGHTING);

        ambientLight.enable(gl);
```

```
        pointLight.enable(gl, GL10.GL_LIGHT0);
        directionalLight.enable(gl, GL10.GL_LIGHT1);
        material.enable(gl);

        gl.glEnable(GL10.GL_TEXTURE_2D);
        texture.bind();

        gl.glRotatef(angle, 0, 1, 0);
        cube.bind();
        cube.draw(GL10.GL_TRIANGLES, 0, 6 * 2 * 3);
        cube.unbind();

        pointLight.disable(gl);
        directionalLight.disable(gl);

        gl.glDisable(GL10.GL_TEXTURE_2D);
        gl.glDisable(GL10.GL_DEPTH_TEST);
    }
```

Здесь уже интереснее. Первые несколько строк являются шаблонным кодом, предназначенным для очистки буфера цветов и глубины, разрешения тестирования глубины и установки области видимости.

Далее мы устанавливаем матрицу проекций равной перспективной матрице проекций с помощью метода `gluPerspective()`, а также используем метод `gluLookAt()` для модельно-видовой матрицы, благодаря чему камера работает так же, как на рис. 11.6.

Затем разрешаем использование освещения. К этому моменту еще не определен ни один источник света, поэтому мы задаем их в следующих нескольких строках с помощью вызова метода `enable()` для источников света и материалов.

Как обычно, также разрешаем текстурирование и привязываем текстуру ящика. Наконец, вызываем метод `glRotatef()` для поворота куба и затем отрисовываем его вершины с помощью удачно размещенных вызовов экземпляра класса `Vertices3`.

В конце метода мы отключаем точечные и направленные источники освещения (помните, подсветка — это глобальное состояние), а также текстурирование и тестирование глубины. Это все, что касается освещения в OpenGL ES!

```
    @Override
    public void pause() {
    }

    @Override
    public void dispose() {
    }
}
```

Остальная часть класса пуста; нам не нужно производить какие-либо действия в случае паузы. На рис. 11.7 показан результат работы программы.

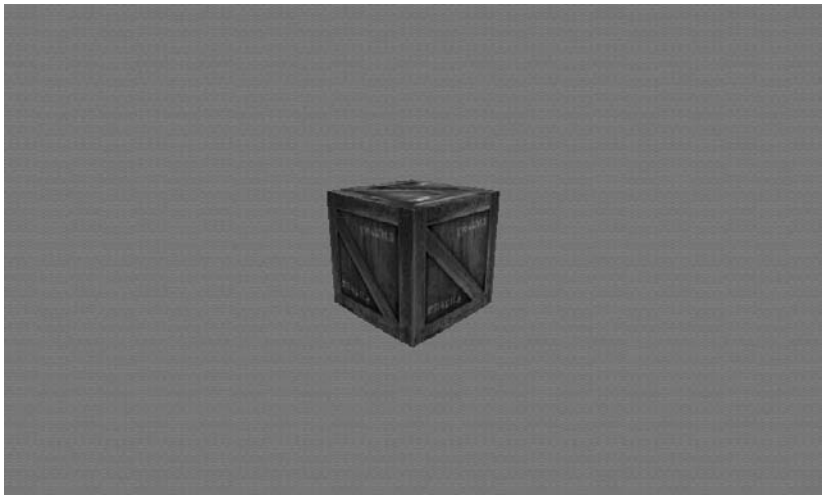

Рис. 11.7. Сцена, изображенная на рис. 11.6, отрисованная с помощью OpenGL ES[1]

Несколько примечаний к освещению в OpenGL ES

Хотя использование освещения может добавить изюминку вашей игре, у него есть свои ограничения и ловушки. Есть несколько моментов, о которых вы должны знать.

- Использование освещения потребляет слишком много ресурсов, особенно это заметно на медленных устройствах. Применяйте освещение осторожно. Чем больше источников освещения вы опишете, тем больше вычислений потребуется, чтобы отрисовать сцену.
- Определять позицию/направление точечных/направленных источников света следует после того, как будут загружены матрицы камеры и до того, как модельно-видовая матрица будет умножена на какие-либо другие матрицы для перемещения и поворота объектов! Это критично. Если не следовать этим указаниям, возможно появление необъяснимых световых артефактов.
- При использовании метода `glScalef()` для изменения размера модели ее нормали также будут масштабированы. Это плохо, поскольку OpenGL ES ожидает, что нормали будут иметь параметры в заданных единицах измерения. Чтобы обойти эту проблему, вы можете использовать команду `glEnable(GL10.GL_NORMALIZE)` или при некоторых обстоятельствах `glEnable(GL10.GL_RESCALE_NORMAL)`. Полагаю, следует использовать первую команду, поскольку применение второй имеет ограничения и подводные камни. Проблема заключается в том, что нормализация или повторное масштабирование нормалей требует большой вычислительной мощности. Лучшее решение с точки зрения производительности — не масштабировать освещенные объекты.

[1] Этот рисунок был сделан немного более контрастным при подготовке книги к изданию. На самом деле изображение менее контрастное. — *Примеч. ред.*

Mip-текстурирование

Если вы достаточно поработали с предыдущими примерами и отодвинули куб подальше от камеры, вы могли заметить, что текстуры начинают выглядеть зернисто и наполняются небольшими артефактами по мере уменьшения куба. Этот эффект называется наложением, или алайзингом, — известный эффект, присутствующий при всех типах обработки сигналов. На рис. 11.8, *справа* показан этот эффект, а на рис. 11.8, *слева* — результат применения приема, который называется mip-текстурированием.

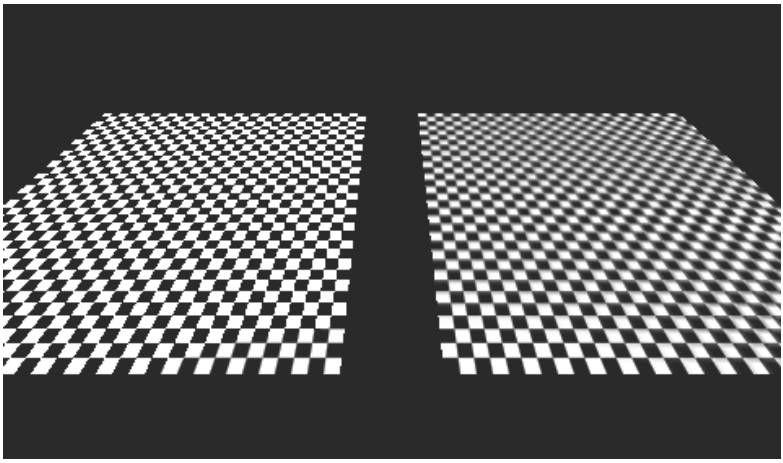

Рис. 11.8. Алайзинговые артефакты (*справа*)
и результат применения mip-текстурирования *(слева)*

Я не буду вдаваться в детали, почему происходит алайзинг; все, что вам нужно знать, — это способ, позволяющий преодолеть его. Он называется mip-текстурированием. Ключ к решению проблем, вызванных алайзингом, — использование изображений с низким разрешением для тех частей объекта, которые выглядят меньше на экране или располагаются дальше от точки обзора. Это часто называют пирамидой или цепью mip-текстур. Основываясь на разрешении изображения по умолчанию, например 256×256 пикселов, мы создаем его меньшие версии, деля каждую его сторону на 2 на каждом уровне пирамиды mip-текстур. На рис. 11.9 показана текстура ящика на различных уровнях mip-текстур.

Чтобы создать mip-текстуру с помощью OpenGL ES, необходимо выполнить два следующих действия:

- установить значение минимизирующего фильтра равным одной из констант, имеющих вид `GL_XXX_MIPMAP_XXX` (обычно `GL_LINEAR_MIPMAP_NEAREST`);
- создать изображения для каждого уровня цепи mip-текстур, изменяя размер оригинального изображения, и загрузить их в OpenGL ES. Цепь mip-текстур прикрепляется к одной текстуре, а не к нескольким.

Рис. 11.9. Цепочка mip-текстур

Для изменения размера базового изображения для цепи mip-текстур можно использовать классы `Bitmap` и `Canvas`, которые предоставляет Android API. Немного изменим класс `Texture` (листинг 11.8).

Листинг 11.8. Класс `Texture.java`, финальная версия класса `Texture`

```
package com.badlogic.androidgames.framework.gl;

import java.io.IOException;
import java.io.InputStream;

import javax.microedition.khronos.opengles.GL10;

import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Rect;
import android.opengl.GLUtils;

import com.badlogic.androidgames.framework.FileIO;
import com.badlogic.androidgames.framework.impl.GLGame;
import com.badlogic.androidgames.framework.impl.GLGraphics;

public class Texture {
    GLGraphics glGraphics;
    FileIO fileIO;
    String fileName;
    int textureId;
    int minFilter;
    int magFilter;
    public int width;
    public int height;
    boolean mipmapped;
```

Добавился только один новый член, который называется mipmapped. Он хранит информацию о том, имеет ли текстура цепь mip-текстур.

```
public Texture(GLGame glGame, String fileName) {
    this(glGame, fileName, false);
}

public Texture(GLGame glGame, String fileName, boolean mipmapped) {
    this.glGraphics = glGame.getGLGraphics();
    this.fileIO = glGame.getFileIO();
    this.fileName = fileName;
    this.mipmapped = mipmapped;
    load();
}
```

Для совместимости мы сохранили старый конструктор, в котором вызывается новый. Новый конструктор принимает третий аргумент, который позволяет определить, необходимо ли текстуре быть mip-текстурированной.

```
private void load() {
    GL10 gl = glGraphics.getGL();
    int[] textureIds = new int[1];
    gl.glGenTextures(1, textureIds, 0);
    textureId = textureIds[0];

    InputStream in = null;
    try {
        in = fileIO.readAsset(fileName);
        Bitmap bitmap = BitmapFactory.decodeStream(in);
        if (mipmapped) {
            createMipmaps(gl, bitmap);
        } else {
            gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
            GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
            setFilters(GL10.GL_NEAREST, GL10.GL_NEAREST);
            gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
            width = bitmap.getWidth();
            height = bitmap.getHeight();
            bitmap.recycle();
        }
    } catch (IOException e) {
        throw new RuntimeException("Couldn't load texture '" + fileName
                + "'", e);
    } finally {
        if (in != null)
            try {
                in.close();
            } catch (IOException e) {
            }
    }
}
```

Метод `load()` остается практически без изменений. Он лишь дополнен вызовом метода `createMipmaps()` на случай, если текстура должна быть **mip-текстурированной**. Не являющиеся таковыми экземпляры класса `Texture` создаются так же, как и раньше.

```
private void createMipmaps(GL10 gl, Bitmap bitmap) {
    gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
    width = bitmap.getWidth();
    height = bitmap.getHeight();
    setFilters(GL10.GL_LINEAR_MIPMAP_NEAREST, GL10.GL_LINEAR);

    int level = 0;
    int newWidth = width;
    int newHeight = height;
    while (true) {
        GLUtils.texImage2D(GL10.GL_TEXTURE_2D, level, bitmap, 0);
        newWidth = newWidth / 2;
        newHeight = newHeight / 2;
        if (newWidth <= 0)
            break;
        Bitmap newBitmap = Bitmap.createBitmap(newWidth, newHeight,
                bitmap.getConfig());
        Canvas canvas = new Canvas(newBitmap);
        canvas.drawBitmap(bitmap,
                new Rect(0, 0, bitmap.getWidth(), bitmap.getHeight()),
                new Rect(0, 0, newWidth, newHeight), null);
        bitmap.recycle();
        bitmap = newBitmap;
        level++;
    }

    gl.glBindTexture(GL10.GL_TEXTURE_2D, 0);
    bitmap.recycle();
}
```

Метод `createMipmaps()` довольно прямолинеен. Начинаем работу с привязки к текстуре, чтобы затем можно было изменять ее атрибуты. Первое, что нам нужно сделать, — отслеживать ширину и высоту бинарного изображения и установить фильтры. Обратите внимание, для минимизирующего фильтра мы применили константу `GL_LINEAR_MIPMAP_NEAREST`. Если мы не будем использовать этот фильтр, mip-текстурирование не сработает и OpenGL ES будет задействовать обычную фильтрацию, используя только базовое изображение.

Содержимое цикла `while` довольно прямолинейно. Мы загружаем текущее bitmap как изображение для текущего уровня. Мы начинаем с уровня 0, базового уровня оригинального изображения. Как только изображение для текущего уровня загружено, мы создаем меньшую его версию, разделив его ширину и высоту на 2. Если новая ширина меньше или равна нулю, мы выходим из бесконечного цикла, поскольку изображения для всех уровней **mip-текстур уже загружены (последнее изображение** имеет размеры 1×1 пиксел). Мы используем класс `Canvas` для изменения

размеров изображения и сохраняем результат в переменной newBitmap. Затем удаляем старое изображение для того, чтобы очистить всю занимаемую им память, и устанавливаем текущее изображение, равное newBitmap. Этот процесс повторяется, пока изображение не станет меньше, чем 1×1 пиксел. Наконец, отвязываем текстуру и удаляем последнее изображение, созданное в цикле.

```
    public void reload() {
        load();
        bind();
        setFilters(minFilter, magFilter);
        glGraphics.getGL().glBindTexture(GL10.GL_TEXTURE_2D, 0);
    }

    public void setFilters(int minFilter, int magFilter) {
        this.minFilter = minFilter;
        this.magFilter = magFilter;
        GL10 gl = glGraphics.getGL();
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER,
                minFilter);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MAG_FILTER,
                magFilter);
    }

    public void bind() {
        GL10 gl = glGraphics.getGL();
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
    }

    public void dispose() {
        GL10 gl = glGraphics.getGL();
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
        int[] textureIds = { textureId };
        gl.glDeleteTextures(1, textureIds, 0);
    }
}
```

Остальная часть класса не изменяется по сравнению с предыдущей версией. Единственное различие в использовании — способ вызова конструктора. И поскольку этот класс очень прост, мы не будем писать пример для иллюстрации работы только лишь mip-текстурирования. Мы используем mip-текстурирование на всех текстурах, применяемых для 3D-объектов. В двух измерениях mip-текстурирование задействуется реже. Несколько заключительных замечаний, касающихся mip-текстурирования.

- Mip-текстурирование может увеличить производительность, если рисуемые с помощью mip-текстуры объекты малы. Причина заключается в том, что GPU приходится получать меньшее количество текселов из меньших изображений пирамиды mip-текстурир. Поэтому мудрым решением является постоянное использование mip-текстур для тех объектов, которые могут уменьшиться.

- Mip-текстуры занимают до 33 % больше памяти, чем аналогичные не mip-текстурированные версии. Этот обмен зачастую справедлив.
- Mip-текстурирование работает только для квадратных текстур, если вы используете OpenGL ES версии 1.x. Это важно помнить. Если ваши объекты остаются белыми, даже несмотря на то, что для них использована текстура с отличным изображением, вы можете быть уверены, что забыли об этом ограничении.

ПРИМЕЧАНИЕ

Повторю еще раз, поскольку это действительно важно. Mip-текстурирование будет работать только с квадратными текстурами! Изображение, имеющее размер 512 × 256 пикселов, не подойдет.

Простые камеры

В предыдущей главе мы говорили о двух способах создания камеры. Первый из них, Эйлерова камера, был похож на тот, который используется в шутерах от первого лица. Второй, камера с видом от третьего лица, применяется для работы кинематической камеры или для следования за объектом. Создадим два вспомогательных класса, которые в дальнейшем можно будет задействовать в наших играх.

Камера с видом от первого лица (Эйлерова камера)

Камера с видом от первого лица (Эйлерова камера) определяется следующими атрибутами:

- поле обзора в градусах;
- соотношение сторон области просмотра;
- ближняя и дальняя плоскости отсечения;
- позиция в 3D-пространстве;
- угол поворота вокруг оси *y*;
- угол поворота вокруг оси *x*. Он ограничен и лежит в промежутке между –90 и 90°. Подумайте о том, как далеко вы сможете наклонить вашу голову, и попробуйте выйти за эти пределы! За любые возможные травмы я ответственности не несу.

Первые три атрибута используются для определения матрицы перспективной проекции. Мы уже делали это с помощью вызова метода `gluPerspective()` во всех примерах с 3D.

Остальные три атрибута определяют позицию и ориентацию камеры в нашем мире.

Создадим матрицу из заданных параметров так же, как мы это делали в предыдущей главе. Поместим все это в простой класс. Его код показан в листинге 11.9.

В дополнение мы хотим передвигать камеру в том направлении, в котором она направлена. Для этого нам понадобится единичный вектор, которым мы сможем добавить к вектору позиции камеры. Мы можем создать такой вектор с помощью класса `Matrix`, который предоставляется в Android API. Немного подумаем об этом.

В своей конфигурации по умолчанию наша камера будет смотреть вдоль отрицательной части оси *z*. Поэтому вектор ее направления равен (0; 0; –1). Когда мы определим углы поворота вокруг осей *x* или *y*, этот вектор будет соответственно повернут. Для определения вектора направления нам лишь нужно умножить его на матрицу, которая повернет стандартный вектор так же, как OpenGL ES повернет вершины наших моделей.

Взглянем на то, как все это работает в коде. В листинге 11.9 показан код класса `EulerCamera`.

Листинг 11.9. Класс EulerCamera.java, простая камера с видом от первого лица, основанная на Эйлеровых углах вокруг осей x и y

```
package com.badlogic.androidgames.framework.gl;

import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLU;
import android.opengl.Matrix;

import com.badlogic.androidgames.framework.math.Vector3;

public class EulerCamera {
    final Vector3 position = new Vector3();
    float yaw;
    float pitch;
    float fieldOfView;
    float aspectRatio;
    float near;
    float far;
```

Первые три члена хранят позицию и углы поворота камеры. Остальные четыре члена используются для расчета матрицы перспективной проекции. По умолчанию камера располагается в начале координат и смотрит вдоль отрицательной части оси *z*.

```
    public EulerCamera(float fieldOfView, float aspectRatio, float near,
                       float far){
        this.fieldOfView = fieldOfView;
        this.aspectRatio = aspectRatio;
        this.near = near;
        this.far = far;
    }
```

Конструктор принимает четыре параметра, определяющих перспективную проекцию. Мы оставляем значения по умолчанию для позиции камеры и углов ее поворота.

```
public Vector3 getPosition() {
    return position;
}
public float getYaw() {
    return yaw;
}

public float getPitch() {
    return pitch;
}
```

Эти методы просто возвращают ориентацию камеры и ее позицию.

```
public void setAngles(float yaw, float pitch) {
    if (pitch < -90)
        pitch = -90;
    if (pitch > 90)
        pitch = 90;
    this.yaw = yaw;
    this.pitch = pitch;
}

public void rotate(float yawInc, float pitchInc) {
    this.yaw += yawInc;
    this.pitch += pitchInc;
    if (pitch < -90)
        pitch = -90;
    if (pitch > 90)
        pitch = 90;
}
```

Метод `setAngles()` позволяет нам напрямую устанавливать значения угла поворота камеры. Обратите внимание на то, что мы ограничиваем значение угла поворота по оси x промежутком от –90 до 90. Мы не можем повернуть собственную голову больше, чем на эти градусы, поэтому камера тоже не должна этого делать.

Метод `rotate()` практически идентичен методу `setAngles()`. Вместо установки градусов он увеличивает их на значение параметра. Это будет полезно, когда мы в следующем примере реализуем небольшую схему управления, основанную на прикосновениях к экрану.

```
public void setMatrices(GL10 gl) {
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    GLU.gluPerspective(gl, fieldOfView, aspectRatio, near, far);
    gl.glMatrixMode(GL10.GL_MODELVIEW);
    gl.glLoadIdentity();
    gl.glRotatef(-pitch, 1, 0, 0);
    gl.glRotatef(-yaw, 0, 1, 0);
    gl.glTranslatef(-position.x, -position.y, -position.z);
}
```

Метод setMatrices() просто инициализирует проекционную и модельно-видовую матрицы так, как мы говорили ранее. Проекционная матрица устанавливается с помощью метода gluPerspective(), основываясь на параметрах, переданных камере в конструкторе. Модельно-видовая матрица выполняет прием «Магомет — гора», применяя поворот и параллельный перенос на оси *x* и *y*. Все факторы, включенные в это действие, меняют свой знак на минус, чтобы достичь того, чтобы камера оставалась в начале координат и направлялась вдоль отрицательной части оси *z*. Именно поэтому мы поворачиваем и параллельно переносим объекты вокруг камеры. Других способов нет.

```
final float[] matrix = new float[16];
    final float[] inVec = { 0, 0, -1, 1 };
    final float[] outVec = new float[4];
    final Vector3 direction = new Vector3();

    public Vector3 getDirection() {
        Matrix.setIdentityM(matrix, 0);
        Matrix.rotateM(matrix, 0, yaw, 0, 1, 0);
        Matrix.rotateM(matrix, 0, pitch, 1, 0, 0);
        Matrix.multiplyMV(outVec, 0, matrix, 0, inVec, 0);
        direction.set(outVec[0], outVec[1], outVec[2]);
        return direction;
    }
}
```

Наконец, мы видим таинственный метод getDirection(). Он содержит несколько членов, которые используются для расчетов внутри метода. Благодаря этому мы не создаем новые массивы чисел с плавающей точкой и экземпляры класса Vector3 каждый раз, когда вызывается этот метод.

Рассматривайте эти члены класса как временные рабочие переменные. Внутри метода мы сначала устанавливаем матрицу преобразований, содержащую информацию о том, что необходимо выполнить поворот вокруг осей *x* и *y*. Нам не нужно включать туда параллельный перенос, поскольку нам необходимо создать вектор направления, а не позиции. Направление камеры не зависит от ее позиции в мире. Методы класса Matrix говорят сами за себя. Единственный странный аспект — мы вызываем эти методы в обратном порядке, не изменяя знак их аргументов. Противоположное происходит при вызове метода setMatrices(), это достигается за счет того, что теперь мы применяем преобразования к точке точно так же, как и к виртуальной камере, которая не обязательно должна размещаться в начале координат и направлена в сторону отрицательной части оси *z*. Поворачиваемый вектор имеет координаты (0; 0; –1) и хранится в переменной inVec. Это направление камеры по умолчанию, если она не была повернута. Все производимые нами манипуляции с матрицами — это поворот вектора направления на углы pitch и roll, поэтому она указывает в том же направлении, что и камера. Последнее, что нам нужно сделать, — установить значение экземпляра класса Vector3, основываясь на результате перемножения матрицы и вектора и вернуть его вызывающей функции.

Мы можем использовать этот единичный вектор направления и в дальнейшем для того, чтобы переместить камеру в том направлении, куда она повернута.

С помощью небольшого вспомогательного класса мы напишем маленькую тестовую программу, позволяющую нам перемещаться в мире ящиков.

Пример работы с Эйлеровой камерой

Испытаем класс `EulerCamera`, написав небольшую программу. Мы хотим поворачивать камеру вверх, вниз, влево и вправо, основываясь на прикосновениях пальца к экрану.

Мы также желаем перемещать ее вперед, если нажата кнопка. Наш мир следует наполнить ящиками. На рис. 11.10 изображена картина, которая предстанет перед глазами пользователя в начале работы.

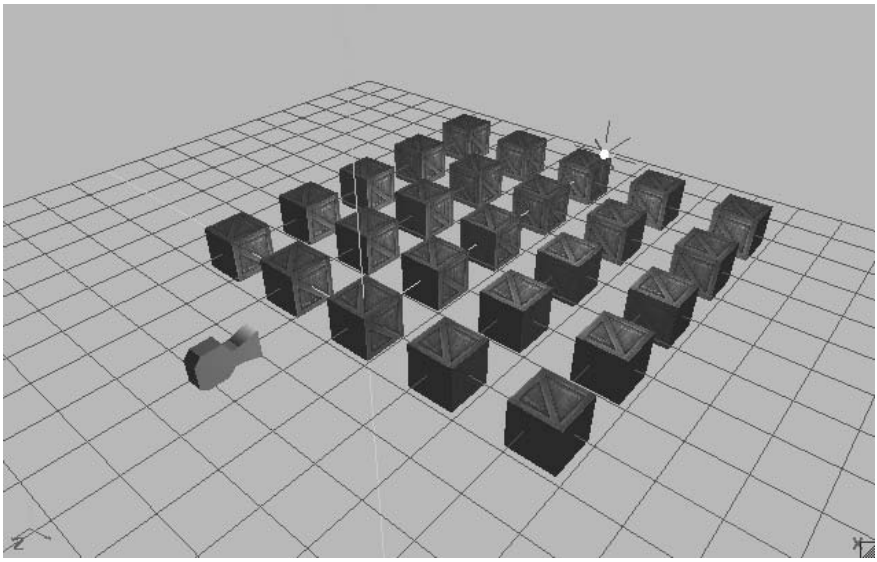

Рис. 11.10. Простая сцена, содержащая 25 ящиков, точечный источник света и Эйлерову камеру с параметрами, заданными по умолчанию

Камера будет находиться в точке (0; 1; 3). В мире также будет располагаться белый точечный источник света в точке (3; 3; –3). Ящики размещаются в сетке с координатами (–4; 4) по оси x и (0; –8) по оси z, расстояние между ящиками равно 2.

Как камера будет реагировать на касания экрана пользователем? Мы хотим, чтобы камера поворачивалась вокруг оси y, если пользователь проведет пальцем по экрану горизонтально. Это эквивалентно повороту головы влево и вправо. Мы также желаем, чтобы камера поворачивалась вокруг оси x после того, как пользователь проведет по экрану вертикально. Это эквивалентно движению головы вверх и вниз. В дополнение мы бы хотели объединить эти два движения. Наиболее

прямолинейный способ добиться этого — проверять, касается ли палец экрана, и, если это так, сравнивать координаты касания с последним сохраненным значением. Далее можем реализовать изменение угла поворота вокруг обеих осей, используя разницу по оси x для поворота по оси y и наоборот.

Мы также хотим, чтобы камера перемещалась вперед при нажатии кнопки, расположенной на экране. Это довольно просто. Мы просто вызываем метод `EulerCamera.getDirection()` и умножаем полученный результат на скорость, с которой должна перемещаться камера, и на изменение времени. Как видите, мы снова реализовываем перемещение, основанное на времени. Единственное, что нам остается сделать, — нарисовать кнопку (я решил нарисовать кнопку размером 64×64 пиксела в нижнем левом углу экрана) и проверять, нажимает ли на нее пользователь.

Чтобы упростить реализацию, мы разрешим пользователю выполнять только одно действие за один раз — либо поворачивать камеру, либо перемещать ее.

Мы могли бы добавить поддержку мультитача, но это усложнило бы нашу реализацию.

Теперь, когда мы описали будущую программу, взглянем на класс `EulerCameraScreen`, реализацию класса `GLScreen`, которая находится внутри реализации класса `GLGame`, называющейся `EulerCameraTest` (обыкновенной тестовой структуры). Код показан в листинге 11.10.

Листинг 11.10. Фрагменты теста EulerCameraTest.java, класс EulerCameraScreen

```
class EulerCameraScreen extends GLScreen {
    Texture crateTexture;
    Vertices3 cube;
    PointLight light;
    EulerCamera camera;
    Texture buttonTexture;
    SpriteBatcher batcher;
    Camera2D guiCamera;
    TextureRegion buttonRegion;
    Vector2 touchPos;
    float lastX = -1;
    float lastY = -1;
```

Мы начинаем с описания нескольких членов класса. Первые два хранят текстуры ящика и вершины текстуры куба. Мы будем генерировать эти вершины с помощью метода `createCube()`, описанного в прошлом примере.

Следующий член класса — это `PointLight`, с которым мы уже знакомы, как и с экземпляром нашего нового класса `EulerCamera`.

Следующие несколько членов нужны для отрисовки кнопки. Для нее мы используем отдельное изображение размером 64×64 пиксела. Нам также понадобятся экземпляры классов `SpriteBatcher`, `Camera2D` и `TextureRegion`. Это означает, что мы объединим 3D- и 2D-отрисовку в одном примере! Последние три члена используются для отслеживания текущей позиции прикосновения к экрану (`touchPos`) в координатах пользовательского интерфейса, а также последние известные точки прикосновений. Мы используем значения –1 для параметров `lastX` и `lastY`, чтобы сообщить, что пока не было ни одного прикосновения.

```
    public EulerCameraScreen(Game game) {
        super(game);

        crateTexture = new Texture(glGame, "crate.png", true);
        cube = createCube();
        light = new PointLight();
        light.setPosition(3, 3, -3);
        camera = new EulerCamera(67, glGraphics.getWidth() /
(float)glGraphics.getHeight(), 1, 100);
        camera.getPosition().set(0, 1, 3);

        buttonTexture = new Texture(glGame, "button.png");
        batcher = new SpriteBatcher(glGraphics, 1);
        guiCamera = new Camera2D(glGraphics, 480, 320);
        buttonRegion = new TextureRegion(buttonTexture, 0, 0, 64, 64);
        touchPos = new Vector2();
    }
```

В конструкторе загружаем текстуру ящика и создаем вершины куба точно так же, как и в прошлом примере. Мы также создаем точечный источник освещения `PointLight` и устанавливаем его позицию равной (3; 3; –3). Экземпляр класса `EulerCamera` создается со стандартными параметрами — поле обзора 67°, соотношение сторон соответствует текущему разрешению экрана, расстояние ближней плоскости отсечения равно 1, а дальней 100. Наконец, перемещаем камеру в позицию (0; 1; 3), что показано на рис. 11.10.

В остальной части конструктора просто загружаем текстуру кнопки, а также создаем экземпляры классов `SpriteBatcher`, `Camera2D` и `TextureRegion`, необходимые для отрисовки кнопки. Наконец, создаем экземпляр класса `Vector2`, благодаря чему можем преобразовывать координаты прикосновения к экрану к координатной системе экземпляра класса `Camera2D`, который используется для отрисовки элементов пользовательского интерфейса, точно так же, как и в игре «Большой прыгун», рассмотренной в главе 9.

```
    private Vertices3 createCube() {
       // аналогично предыдущему примеру
    }

    @Override
    public void resume() {
        crateTexture.reload();
    }
```

Методы `createCube()` и `resume()` совершенно не изменились с предыдущего примера, поэтому я не повторяю здесь весь их код.

```
    @Override
    public void update(float deltaTime) {
        game.getInput().getTouchEvents();
        float x = game.getInput().getTouchX(0);
        float y = game.getInput().getTouchY(0);
```

```
        guiCamera.touchToWorld(touchPos.set(x, y));

        if(game.getInput().isTouchDown(0)) {
            if(touchPos.x < 64 && touchPos.y < 64) {
                Vector3 direction = camera.getDirection();
                camera.getPosition().add(direction.mul(deltaTime));
            } else {
                if(lastX == -1) {
                    lastX = x;
                    lastY = y;
                } else {
                    camera.rotate((x – lastX) / 10, (y – lastY) / 10);
                    lastX = x;
                    lastY = y;
                }
            }
        } else {
            lastX = -1;
            lastY = -1;
        }
    }
```

В методе `update()` происходят поворот камеры и ее перемещение, которые основываются на событиях прикосновений к экрану. Первое, что мы делаем, — очищаем буфер этих событий с помощью вызова метода `Input.getTouchEvents()`. Далее получаем координаты текущего прикосновения первого пальца к экрану. Обратите внимание, если палец не прикасается к экрану, вызываемые методы просто вернут последнюю известную позицию экрана с индексом 0. Мы также преобразовываем координаты прикосновения к координатной системе пользовательского интерфейса. Это делается для того, чтобы проверить, нажал ли пользователь кнопку в левом нижнем углу.

После того как мы получим все эти значения, выполняется проверка на то, прикасается ли палец к экрану в данный момент. Если да, сначала проверяем, не нажата ли кнопка, которая заполняет все пространство от точки (0; 0) до точки (64; 64) двухмерной системы координат пользовательского интерфейса. Если это так, то мы получаем текущее направление камеры и, умножив на изменение времени, добавляем его к позиции камеры.

Поскольку мы работаем с единичным вектором, камера будет перемещаться на одну единицу расстояния в секунду.

Если же пользователь не коснулся кнопки, программа интерпретирует это прикосновение так, будто он провел по экрану пальцем. Чтобы работать в этом направлении дальше, нам нужно знать последнюю координату прикосновения к экрану. Во время первого прикосновения пользователя к экрану члены `lastX` и `lastY` будут равны –1. Это говорит о том, что в этот раз мы не можем вычислить разность между последней и текущей координатами прикосновения к экрану, поскольку у нас есть всего лишь один отсчет данных. Поэтому мы просто сохраняем текущие координаты и возвращаемся из метода `update()`. Если координаты прикосновения уже сохранялись, просто находим разность между значениями координат по осям x и y

для предыдущего и текущего прикосновений. Затем напрямую преобразовываем их к инкрементам углов поворота.

Чтобы несколько замедлить поворот камеры, полученную разность придется делить на 10. Единственное, что остается сделать, — вызвать метод `EulerCamera.rotate()`, который соответственно изменит углы поворота камеры.

Наконец, если палец больше не касается экрана, устанавливаем значения параметров `lastX` и `lastY` равными –1, сообщая, что нам снова приходится ждать первого прикосновения к экрану перед тем, как осуществлять обработку события прикосновения.

```
    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        gl.glViewport(0, 0, glGraphics.getWidth(), glGraphics.getHeight());

        camera.setMatrices(gl);

        gl.glEnable(GL10.GL_DEPTH_TEST);
        gl.glEnable(GL10.GL_TEXTURE_2D);
        gl.glEnable(GL10.GL_LIGHTING);

        crateTexture.bind();
        cube.bind();
        light.enable(gl, GL10.GL_LIGHT0);

        for(int z = 0; z >= -8; z-=2) {
            for(int x = -4; x <=4; x+=2 ) {
                gl.glPushMatrix();
                gl.glTranslatef(x, 0, z);
                cube.draw(GL10.GL_TRIANGLES, 0, 6 * 2 * 3);
                gl.glPopMatrix();
            }
        }

        cube.unbind();

        gl.glDisable(GL10.GL_LIGHTING);
        gl.glDisable(GL10.GL_DEPTH_TEST);

        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);

        guiCamera.setViewportAndMatrices();
        batcher.beginBatch(buttonTexture);
        batcher.drawSprite(32, 32, 64, 64, buttonRegion);
        batcher.endBatch();

        gl.glDisable(GL10.GL_BLEND);
        gl.glDisable(GL10.GL_TEXTURE_2D);
    }
```

Метод present() удивительно прост благодаря тому, что он содержит вызовы всех вспомогательных классов. Мы начинаем с привычных действий — очистки экрана и установки окна просмотра. Далее указываем классу EulerCamera установить проекционную и видовую матрицы.

С этого момента мы можем отрисовывать на экране все трехмерные элементы. До того как мы сможем сделать это, следует разрешить тестирование глубины, текстурирование и освещение. Далее связываем текстуру ящика и вершины куба, а также разрешаем использование точечного источника освещения. Обратите внимание, мы связываем текстуру и вершины куба только однажды, поскольку хотим повторно их использовать для всех отрисовываемых ящиков. Тот же самый прием мы применяли в примере BobTest (глава 8) для ускорения отрисовки с помощью сокращения количества изменений состояния.

Следующий фрагмент кода просто рисует 25 кубов, расположенных в форме сетки, с помощью простого вложенного цикла for. Поскольку нам приходится умножать модельно-видовую матрицу на матрицу преобразования для того, чтобы поместить вершины куба на определенные позиции, мы также должны использовать методы glPushMatrix() и glPopMatrix(), чтобы не уничтожать матрицу камеры, которая также является модельно-видовой.

Как только мы отрисуем все кубы, необходимо отвязать вершины куба, а также отключить освещение и тестирование глубины. Это критично, поскольку теперь мы собираемся отрисовать двухмерный элемент пользовательского интерфейса, содержащий кнопку. Поскольку кнопка является круглой, мы также включаем смешивание, чтобы сделать границы текстуры прозрачными.

Отрисовка кнопки работает точно так же, как и отрисовка элементов пользовательского интерфейса в игре «Большой прыгун». Мы приказываем экземпляру класса Camera2D установить окно просмотра и матрицы (на самом деле нам не понадобится повторно устанавливать окно просмотра, вы можете свободно «отпимизировать» этот метод) и указываем экземпляру класса SpriteBatcher, что мы собираемся отрисовать спрайт. Мы отрисовываем всю текстуру кнопки в точке (32; 32) в нашей системе координат размером (480 × 320), установленной с помощью guiCamera.

Наконец, мы просто отключаем несколько предыдущих состояний, которые подключили ранее, а также смешивание и текстурирование.

```
    @Override
    public void pause() {

    }

    @Override
    public void dispose() {
    }
}
```

Остальная часть класса — это просто пустые методы pause() и dispose(). На рис. 11.11 показан результат работы этой небольшой программы.

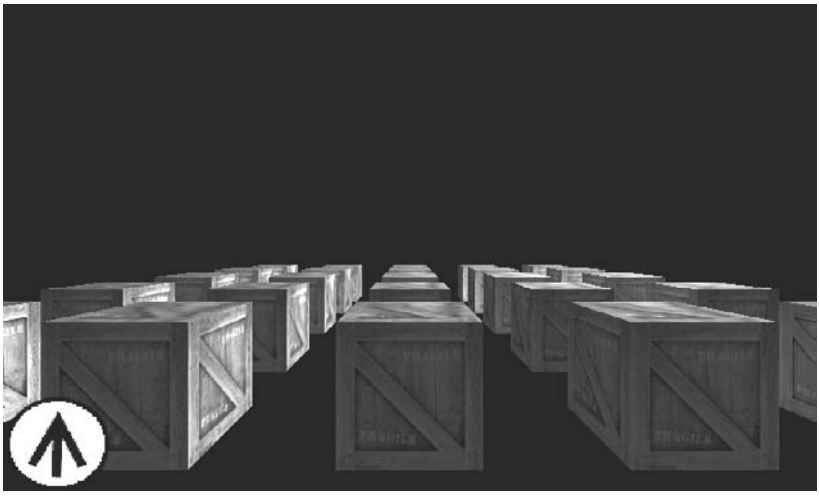

Рис. 11.11. Простой пример, иллюстрирующий работу камеры для первого лица, мультитач для простоты не поддерживается[1]

Довольно мило, не правда ли? Программа имеет небольшой объем благодаря нашим вспомогательным классам. Теперь неплохо было бы добавить поддержку мультитача. Вот небольшая подсказка: вместо использования опроса, как в предыдущем примере, применяйте реальные события прикосновения. При прикосновении пальца к экрану выполняйте проверку нажатия кнопки. Если это случилось, отмечайте ID указателя, связанный с ней так, что он не будет способен обрабатывать проведение пальцем по экрану, пока первый палец не оторвется от экрана. События прикосновения к экрану от всех остальных ID могут быть интерпретированы как проведение пальцем по экрану!

Камера с видом от третьего лица

Второй тип камеры, который часто можно встретить в играх, — это камера с видом от третьего лица. Она определяется следующими параметрами:

- позиция в пространстве;
- верхний вектор; рассматривайте его как стрелку, выходящую из вершины вашего черепа и указывающую в том же направлении, что и она;
- позиция взгляда в пространстве или в качестве альтернативы — вектор направления; мы будем использовать первый вариант;
- поле видимости в градусах;
- соотношение сторон окна просмотра;
- расстояния ближней и дальней плоскостей отсечения.

[1] Изображение на самом деле менее контрастное. При подготовке издания к печати контрастность рисунка была немного увеличена. — *Примеч. ред.*

Единственное отличие от Эйлеровой камеры заключается в способе представления ориентации камеры. В этом случае мы определяем ее как верхний вектор и позиция взгляда.

Напишем вспомогательный класс для этого типа камеры (листинг 11.11).

Листинг 11.11. Класс LookAtCamera.java, простая камера с видом от третьего лица без особых изысков

```
package com.badlogic.androidgames.framework.gl;

import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLU;

import com.badlogic.androidgames.framework.math.Vector3;

public class LookAtCamera {
    final Vector3 position;
    final Vector3 up;
    final Vector3 lookAt;
    float fieldOfView;
    float aspectRatio;
    float near;
    float far;

    public LookAtCamera(float fieldOfView, float aspectRatio, float near,
                        float far) {
        this.fieldOfView = fieldOfView;
        this.aspectRatio = aspectRatio;
        this.near = near;
        this.far = far;

        position = new Vector3();
        up = new Vector3(0, 1, 0);
        lookAt = new Vector3(0,0,-1);
    }

    public Vector3 getPosition() {
        return position;
    }

    public Vector3 getUp() {
        return up;
    }

    public Vector3 getLookAt() {
        return lookAt;
    }

    public void setMatrices(GL10 gl) {
        gl.glMatrixMode(GL10.GL_PROJECTION);
```

```
        gl.glLoadIdentity();
        GLU.gluPerspective(gl, fieldOfView, aspectRatio, near, far);
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();
       GLU.gluLookAt(gl, position.x, position.y, position.z, lookAt.x, lookAt.y,
lookAt.z, up.x, up.y, up.z);
    }
}
```

Ничего удивительного. Мы просто сохраняем значения `position`, `up` и `lookAt` как экземпляры класса `Vector3` наряду с параметрами перспективной проекции, которые также использовались и для `EulerCamera`. В дополнение мы предоставляем несколько методов, возвращающих значения параметров, поэтому можем модифицировать атрибуты камеры. Единственный стоящий нашего внимания метод — это `setMatrices()`. Но даже он нам уже знаком. Мы сначала устанавливаем матрицу проекции, основываясь на поле видимости, соотношении сторон и расстояний до ближней и дальней плоскостей отсечения. Далее устанавливаем модельно-видовую матрицу таким образом, чтобы она содержала позицию камеры и матрицу ориентации с помощью метода `gluLookAt()`, как мы обсуждали в предыдущей главе. Это сделает матрицу, очень похожую на ту, которую мы создали собственноручно в примере `EulerCamera`. Это также повернет объекты вокруг камеры. Однако замечательный интерфейс метода `gluLookAt()` позволяет нам не делать такие глупые вещи, как инвертирование позиций или углов.

Фактически использование этой камеры похоже на применение камеры Эйлера. Все, что нам нужно, — создать вектор направления, вычтя позицию камеры от точки, на которую она направлена, и нормализовав полученный результат. Далее мы просто поворачиваем этот вектор на углы поворота вокруг осей x и y.

Наконец, мы устанавливаем новую позицию взгляда на позицию камеры и добавляем вектор направления.

Оба способа создадут одинаковую матрицу трансформации. Эти способы отличаются лишь обработкой ориентации камеры.

Мы воздержимся от написания прямого примера с использованием класса `LookAtCamera`, поскольку его интерфейс идеально прост. Мы будем применять его в последней игре, которую рассмотрим в книге, где камера будет следовать за небольшим милым космическим кораблем! Если же вы хотите узнать, на что способен этот класс, добавьте его в пример `LightTest`, который мы написали ранее, или модифицируйте пример `EulerCameraTest` таким образом, чтобы класс `LookAtCamera` использовался как камера от первого лица, что мы обозначили в предыдущем разделе.

Загрузка моделей

Определение моделей вроде нашего куба в коде выглядит довольно громоздко. Более удачным способом создания подобных моделей является использование специального программного обеспечения, позволяющего довольно просто создавать

сложные фигуры и формы. Существует множество приложений, которые можно применять для решения этой задачи.

- Blender — проект с открытым кодом, использующийся во многих играх и фильмах. Очень мощный и гибкий, но в то же время несколько устрашающий.
- Wings3D — мой выбор, также является проектом с открытым кодом. Я использую его для моделирования низкополигональных (имеющих в своем составе не так много треугольников) статических объектов. Он очень прост, но с поставленной задачей справляется хорошо.
- 3ds Max — один из стандартов де-факто в этой области. Это коммерческий продукт, однако доступно множество версий для студентов.
- Maya — еще один фаворит. Также коммерческий продукт, но на него предоставляются некоторые скидки, поэтому он доступен более широкому кругу пользователей.

Это лишь небольшая подборка из множества подобных программ. Обучение работе с этими программами не является целью данной книги. Неважно, какую программу вы выберете, в какой-то момент вы сохраните свои результаты в какомлибо формате. Одним из таких форматов является Wavefront OBJ, очень старый формат, использующий неформатированный текст, который с легкостью может быть преобразован в формат, используемый экземпляром класса `Vertices3`.

Формат Wavefront OBJ

Мы реализуем загрузчик для подмножества, представленного в таком формате. Он будет поддерживать модели, состоящие только из треугольников и, возможно, содержащие координаты текстур и нормали. Формат OBJ также поддерживает хранение произвольных выпуклых полигонов, но мы не будем рассматривать этот параметр. Если вы найдете или создадите модель OBJ, просто убедитесь в том, что она состоит исключительно из треугольников.

Формат OBJ основан на строках. Рассмотрим следующие элементы синтаксиса.

- `v x y z` — `v` говорит о том, что строка содержит позицию вершины, а координаты *x*, *y* и *z* представлены как числа с плавающей точкой.
- `vn i j k` — `n` говорит о том, что строка содержит нормаль вершины, а `i`, `j` и `k` — это компоненты нормали вершины по осям *x*, *y* и *z*.
- `vt u v` — `vt` говорит о том, что строка содержит пару координат текстуры, а `u` и `v` являются координатами текстуры.
- `f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3` — `f` говорит о том, что строка содержит треугольник. Каждый блок `v/vt/vn` содержит индексы позиции, координаты текстуры и нормали вершины для одной вершины треугольника. Индексы относительны позициям вершин, координатам текстуры и нормалям вершины, определенным ранее в предыдущих форматах. Индексы `vt` и `vn` могут быть опущены, что говорит о том, что для вершины треугольника не определены координаты текстуры или нормаль.

Мы будем игнорировать любые строки, не начинающиеся с `v`, `vn`, `vt` или `f`; мы также будем выводить сообщение об ошибке, если разрешенные строки будут отформатированы отличным от нашего способом. Элементы одной строки разделяются пустым пространством, которое может состоять из пробелов, табуляций и т. д.

ПРИМЕЧАНИЕ

Формат OBJ может хранить гораздо больше информации. Мы можем преобразовывать лишь описанные строки и отбрасывать все остальные только в том случае, если модели состоят из треугольников и имеют нормали, а также координаты текстур.

Рассмотрим очень простой пример, текстурный треугольник, имеющий нормали, представленный в формате OBJ:

```
v -0.5 -0.5 0
v 0.5 -0.5 0
v 0 0.5 0
vn 0 0 1
vn 0 0 1
vn 0 0 1
vt 0 1
vt 1 1
vt 0.5 0
f 1/1/1 2/2/2 3/3/3
```

Обратите внимание на то, что позиции вершин, координаты текстуры и нормали не всегда будут определены в таком приятном глазу порядке. Они могут переплестись, если программа, сохранившая файл, решит так сделать.

Индексы, заданные в строке `f`, имеют в качестве базы единицу, а не ноль (как это было в Java). Некоторые программы иногда предоставляют даже отрицательные индексы. Это разрешается спецификацией формата OBJ, но является серьезной помехой. Нам следует отслеживать, сколько позиций вершин, координат текстуры или нормалей вершин мы уже загрузили, а затем добавлять эти отрицательные индексы к соответствующим номерам позиций, координат вершин или нормалей в зависимости от атрибута, на который указывает индекс.

Реализация загрузчика файлов формата OBJ

Наш план действий будет таким — загрузить файл в память полностью и создать строку для каждой его строки. Мы также создадим временные массивы чисел с плавающей точкой для всех позиций вершин, координат текстур и нормалей, которые мы собираемся загрузить. Их размер равен количеству строк OBJ-файла, умноженному на количество компонент атрибута, — два для координат текстур или три для нормалей. Благодаря этому мы выделим памяти больше, чем необходимо, но это лучше, чем выделять память для новых массивов всякий раз, когда их понадобится пополнить.

То же самое мы сделаем для индексов, определяющих каждый треугольник. Поскольку формат OBJ является индексированным, мы не можем передать эти индексы непосредственно в класс Vertices3.

Причиной этого является тот факт, что атрибуты вершины могут быть использованы повторно несколькими вершинами, поэтому складываются отношения «один ко многим», что не разрешается OpenGL ES. Поэтому мы будем использовать неиндексированный экземпляр класса Vertices3 и просто дублировать вершины. Такой подход подойдет для наших нужд.

Посмотрим, как мы можем реализовать все это. Код этого класса содержится в листинге 11.12.

Листинг 11.12. Класс ObjLoader.java, простой класс, предназначенный для загрузки подмножества формата OBJ

```
package com.badlogic.androidgames.framework.gl;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

import com.badlogic.androidgames.framework.impl.GLGame;

public class ObjLoader {
    public static Vertices3 load(GLGame game, String file) {
        InputStream in = null;
        try {
            in = game.getFileIO().readAsset(file);
            List<String> lines = readLines(in);

            float[] vertices = new float[lines.size() * 3];
            float[] normals = new float[lines.size() * 3];
            float[] uv = new float[lines.size() * 2];

            int numVertices = 0;
            int numNormals = 0;
            int numUV = 0;
            int numFaces = 0;
```

Первое, что мы делаем, — открываем в потоке InputStream файл с моделью, определяемый параметром file. Потом в методе readLines() (определенном далее) мы считываем все строки этого файла. Основываясь на количестве строк, мы выделяем память для массивов чисел с плавающей точкой, которые будут хранить компоненты осей x, y и z каждой нормали вершин, а также u- и v-координаты текстур каждой вершины. Поскольку мы не знаем, сколько именно вершин мы увидим в файле, просто выделяем больше памяти, чем требуется. Каждый атрибут вершины хранится в последовательных элементах трех массивов. Позиция первой считанной вершины представлена элементами массива vertices[0], vertices[1]

и vertices[2] и т. д. Мы также отслеживаем индексы при определении треугольников для каждого из трех атрибутов вершины. В дополнение у нас есть несколько счетчиков для отслеживания того, как много всего мы уже загрузили.

```
for (int i = 0; i < lines.size(); i++) {
    String line = lines.get(i);
```

Далее у нас есть цикл, которых проходит по всем строкам файлов.

```
if (line.startsWith("v ")) {
    String[] tokens = line.split("[ ]+");
    vertices[vertexIndex] = Float.parseFloat(tokens[1]);
    vertices[vertexIndex + 1] = Float.parseFloat(tokens[2]);
    vertices[vertexIndex + 2] = Float.parseFloat(tokens[3]);
    vertexIndex += 3;
    numVertices++;
    continue;
}
```

Если текущая строка является позицией вершины, мы считываем ее части, разделенные пробелами, такие как координаты по осям *x*, *y* и *z*, и сохраняем их в массиве вершин.

```
if (line.startsWith("vn ")) {
    String[] tokens = line.split("[ ]+");
    normals[normalIndex] = Float.parseFloat(tokens[1]);
    normals[normalIndex + 1] = Float.parseFloat(tokens[2]);
    normals[normalIndex + 2] = Float.parseFloat(tokens[3]);
    normalIndex += 3;
    numNormals++;
    continue;
}

if (line.startsWith("vt")) {
    String[] tokens = line.split("[ ]+");
    uv[uvIndex] = Float.parseFloat(tokens[1]);
    uv[uvIndex + 1] = Float.parseFloat(tokens[2]);
    uvIndex += 2;
    numUV++;
    continue;
}
```

Мы делаем то же самое для нормалей и координат текстур:

```
if (line.startsWith("f ")) {
    String[] tokens = line.split("[ ]+");

    String[] parts = tokens[1].split("/");
    facesVerts[faceIndex] = getIndex(parts[0], numVertices);
    if (parts.length > 2)
        facesNormals[faceIndex] = getIndex(parts[2],
                                           numNormals);
    if (parts.length > 1)
```

```
                facesUV[faceIndex] = getIndex(parts[1], numUV);
            faceIndex++;

            parts = tokens[2].split("/");
            facesVerts[faceIndex] = getIndex(parts[0], numVertices);
            if (parts.length > 2)
                facesNormals[faceIndex] = getIndex(parts[2],
                                                   numNormals);
            if (parts.length > 1)
                facesUV[faceIndex] = getIndex(parts[1], numUV);
            faceIndex++;

            parts = tokens[3].split("/");
            facesVerts[faceIndex] = getIndex(parts[0], numVertices);
            if (parts.length > 2)
                facesNormals[faceIndex] = getIndex(parts[2],
                                                   numNormals);
            if (parts.length > 1)
                facesUV[faceIndex] = getIndex(parts[1], numUV);
            faceIndex++;
            numFaces++;
            continue;
        }
    }
```

В этом коде каждая вершина треугольника (названного здесь `face` (ребро) из-за терминологии формата OBJ) определяется тройкой индексов в массивах позиции вершин, координат текстур и нормалей. Индексы координат текстур и нормалей могут быть опущены, поэтому нам нужно следить за ними. Индексы также могут быть отрицательными, в этом случае нам придется добавлять их к количеству позиций/координат текстур/нормалей, загруженных ранее. Этим занимается метод `getIndex()`.

```
float[] verts = new float[(numFaces * 3)
        * (3 + (numNormals > 0 ? 3 : 0) + (numUV > 0 ? 2 : 0))];
```

Как только мы загрузим все позиции вершин, нормали и треугольники, мы можем начать создавать массив чисел с плавающей точкой, содержащий вершины в формате, который использует экземпляр класса `Vertices3`. Количество чисел с плавающей точкой, необходимых для хранения этих вершин, может быть с легкостью получено как число загруженных треугольников, а также с помощью информации о том, заданы ли нормали и координаты текстур.

```
for (int i = 0, vi = 0; i < numFaces * 3; i++) {
    int vertexIdx = facesVerts[i] * 3;
    verts[vi++] = vertices[vertexIdx];
    verts[vi++] = vertices[vertexIdx + 1];
    verts[vi++] = vertices[vertexIdx + 2];

    if (numUV > 0) {
```

```
                int uvIdx = facesUV[i] * 2;
                verts[vi++] = uv[uvIdx];
                verts[vi++] = 1 - uv[uvIdx + 1];
            }

            if (numNormals > 0) {
                int normalIdx = facesNormals[i] * 3;
                verts[vi++] = normals[normalIdx];
                verts[vi++] = normals[normalIdx + 1];
                verts[vi++] = normals[normalIdx + 2];
            }
        }
```

Чтобы заполнить массив `verts`, мы просто проходим по всем треугольникам, получаем атрибуты каждой вершины треугольника и помещаем их в массив `verts` по шаблону, который используется экземпляром класса `Vertices3`.

```
        Vertices3 model = new Vertices3(game.getGLGraphics(), numFaces *
                          3, 0, false, numUV > 0, numNormals > 0);
        model.setVertices(verts, 0, verts.length);
        return model;
```

Последнее, что мы делаем, — создаем экземпляр класса `Vertices3` и устанавливаем вершины.

```
    } catch (Exception ex) {
        throw new RuntimeException("couldn't load '" + file + "'", ex);
    } finally {
        if (in != null)
            try {
                in.close();
            } catch (Exception ex) {

            }
    }
}
```

Остальная часть метода посвящена обработке исключений и закрытию потока `InputStream`.

```
static int getIndex(String index, int size) {
    int idx = Integer.parseInt(index);
    if (idx < 0)
        return size + idx;
    else
        return idx - 1;
}
```

Метод `getIndex()` принимает один из индексов, заданных для атрибута вершины определяемого треугольника, а также количество ранее загруженных атрибутов,

и возвращает индекс, подходящий для ссылки на атрибут, который находится в одном из наших рабочих массивов.

```
    static List<String> readLines(InputStream in) throws IOException {
        List<String> lines = new ArrayList<String>();

        BufferedReader reader = new BufferedReader(new
                                                InputStreamReader(in));
        String line = null;
        while ((line = reader.readLine()) != null)
            lines.add(line);
        return lines;
    }
}
```

Наконец, рассмотрим метод `readLines()`, который просто считывает каждую строку файла и возвращает их все как список строк.

Чтобы загрузить OBJ-файл из ресурса, мы можем использовать класс `ObjLoader` следующим образом:

```
Vertices3 model = ObjLoader.load(game, "mymodel.obj");
```

Довольно прямолинейно после всей этой чехарды с индексами, не правда ли? Для отрисовки этого экземпляра класса `Vertices3` нам необходимо знать, сколько он имеет вершин. Расширим класс `Vertices3` еще раз, добавив два метода, возвращающих количество вершин и количество индексов, которые определены в объекте в данный момент. Код этих методов содержится в листинге 11.13.

Листинг 11.13. Фрагменты класса `Vertices3.java`, получение количества вершин и индексов

```
public int getNumIndices() {
    return indices.limit();
}

public int getNumVertices() {
    return vertices.limit() / (vertexSize / 4);
}
```

Для количества индексов мы просто возвращаем границу `ShortBuffer`, хранящего индексы. Для количества вершин мы делаем то же самое. Однако, поскольку граница сообщается в виде количества чисел с плавающей точкой, определенных в буфере `FloatBuffer`, нам придется делить его на размер вершины.

Поскольку мы храним это количество байтов в параметре `vertexSize`, мы делим этот член класса на 4.

Использование класса OBJ Loader

Чтобы продемонстрировать работу загрузчика файлов OBJ, я переписал последний пример и создал новый тест, который называется `ObjTest`, а также экран `ObjScreen`. Я скопировал весь код из предыдущего примера и изменил только одну строку конструктора класса `ObjScreen`:

```
cube = ObjLoader.load(glGame, "cube.obj");
```

Поэтому вместо использования метода `createCube()` (который я убрал) теперь мы напрямую загружаем модель из файла с расширением OBJ, который называется `cube.obj`. В программе Wings3D я создал копию куба, которую мы ранее создавали программно в методе `createCube()`. Она имеет те же позиции вершин, текстурные координаты и нормали, что и созданная вручную версия. Вас не должно удивлять то, что результат работы программы `ObjTest` выглядит точно так же, как и результат работы `EulerCameraTest`. Поэтому я не буду приводить скриншот.

Несколько замечаний по загрузке моделей

Для игры, которую мы напишем в следующей главе, загрузчик имеет большое значение, но он далеко не надежен. Есть несколько подводных камней.

- Обработка строк в ОС Android по определению медленная. OBJ — это формат, использующий неформатированный текст, поэтому он требует больших затрат времени на свое преобразование. Это отрицательно повлияет на время загрузки. Вы можете обойти эту проблему, преобразуя ваши OBJ-модели в пользовательский бинарный формат. Вы можете, например, сериализовать массив `verts` таким образом, что мы заполняем его в методе `ObjLoader.load()`.
- Формат OBJ имеет гораздо больше особенностей, чем мы рассмотрели. Если вы хотите расширить наш простой загрузчик, посмотрите спецификацию формата в Интернете. Добавить новую функциональность должно быть легко.
- OBJ-файл часто предоставляется вместе с файлом материала. Этот файл определяет цвета и текстуры, используемые группами вершин, которые описаны в OBJ-файле. Нам не понадобится эта функциональность, поскольку мы знаем, какую текстуру использовать для конкретного OBJ-файла. Для создания более надежного загрузчика вам потребуется заглянуть в спецификацию файлов материала.

Немного физики в 3D

В главе 8 мы разработали очень простую физическую модель в 2D. Хорошие новости — в 3D все работает точно так же!

- Позиции теперь являются 3D-векторами вместо 2D-векторов. Мы просто добавили координату *z*.
- Скорости по-прежнему представляются метрами в секунду для каждой оси. Мы просто добавляем еще один компонент для оси *z*!
- Ускорения все так же представляются метрами на секунду в квадрате для каждой оси. Опять же, мы просто добавляем еще одну координату.

Псевдокод, описывающий эмуляцию физики, который мы рассматривали в главе 8, выглядит следующим образом:

```
Vector2 position = new Vector2();
Vector2 velocity = new Vector2();
Vector2 acceleration = new Vector2(0, -10);
```

```
while(simulationRuns) {
   float deltaTime = getDeltaTime();
   velocity.add(acceleration.x * deltaTime, acceleration.y * deltaTime);
   position.add(velocity.x * deltaTime, velocity.y * deltaTime);
}
```

Мы можем преобразовать его так, чтобы он мог использоваться в трехмерном пространстве, просто заменив экземпляры класса `Vector2` на экземпляры класса `Vector3`:

```
Vector3 position = new Vector3();
Vector3 velocity = new Vector3();
Vector3 acceleration = new Vector3(0, -10, 0);
while(simulationRuns) {
   float deltaTime = getDeltaTime();
   velocity.add(acceleration.x * deltaTime, acceleration.y * deltaTime,
                acceleration.z * deltaTime);
   position.add(velocity.x * deltaTime, velocity.y * deltaTime, velocity.z *
                deltaTime);
}
```

Больше ничего делать не следует. Эта простая физическая модель подходит для множества трехмерных игр. В последней игре этой книги мы даже не будем использовать ускорение из-за природы объектов этой игры.

Более сложную физику в 3D (и 2D), безусловно, сложнее реализовать. Для этой цели вы можете использовать сторонние библиотеки вместо того, чтобы заново изобретать велосипед самостоятельно.

Проблема ОС Android заключается в том, что решения, основанные на Java, будут слишком медленными из-за огромного количества вычислений, которые они производят. Существуют некоторые решения для физики в 2D на ОС Android, которые используют C++ библиотеки вроде `Box2D` с помощью Java Native Interface (JNI), предоставляя API для приложения на Java. Для физики в 3D есть библиотека, которая называется `Bullet`. Однако для этой библиотеки пока не существует готовых к использованию привязок JNI. Мы не будем рассматривать эти темы в книге, поскольку они достаточно далеки от ее цели, во многих случаях нам не понадобится такая сложная физическая модель.

Определение столкновений и представление объектов в 3D

В главе 8 мы обсудили отношение между представлением объектов и определением столкновений. Мы постараемся сделать объекты нашего игрового мира как можно более независимыми от их графического представления. Вместо этого мы хотели бы определить их, описав их ограничивающую фигуру, позицию и ориентацию. Позиция и ориентация не являются серьезной проблемой: мы можем представить позицию как экземпляр класса `Vector3`, а ориентацию — как поворот

вокруг осей *x*, *y* и *z* (не забывая при этом потенциальную проблему шарнирного замка, упомянутую в предыдущей главе). Рассмотрим ограничивающие фигуры подробнее.

Ограничивающие фигуры в 3D

Для работы с ограничивающими фигурами у нас есть множество вариантов. Рисунок 11.12 демонстрирует наиболее популярные ограничивающие фигуры в 3D-программировании.

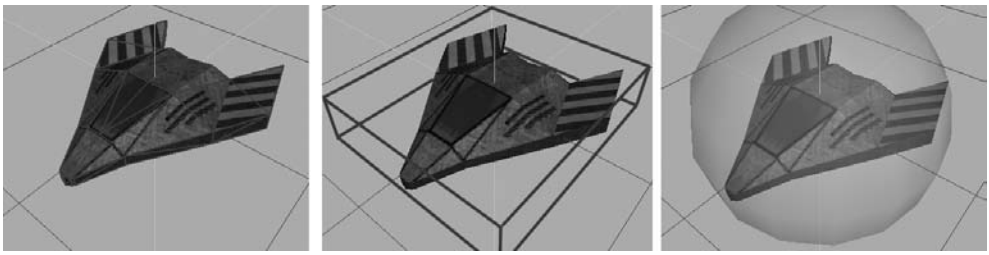

Рис. 11.12. Различные ограничивающие фигуры: сеть треугольников *(слева)*, ограничивающий параллелепипед *(в центре)*, выровненный по координатным осям, ограничивающая сфера *(справа)*

- *Сеть треугольников*. Наиболее близко прилегает к объекту. Однако столкновение двух объектов, использующих сети треугольников, довольно долго рассчитывается.
- *Ограничивающий параллелепипед, выровненный по координатным осям*. Объект располагается в фигуре более свободно. Столкновение таких объектов рассчитывается быстрее, чем столкновение объектов, основанных на сети треугольников.
- *Ограничивающая сфера*. Объект в ней чувствует себя еще более свободно. Это наиболее быстрый способ определить столкновения.

Еще одной проблемой, связанной с сетью треугольников и ограничивающими параллелепипедами, является тот факт, что их требуется переориентировать, если необходимо повернуть или масштабировать объект, так же как и в 2D. Ограничивающие сферы не претерпевают никаких изменений, если мы поворачиваем объект. Если мы масштабируем объект, нам нужно просто соответственно изменить радиус сферы с помощью простого умножения.

Проверка пересечения ограничивающих сфер

Математика расчетов столкновений сетей треугольников или ограничивающих параллелепипедов может быть довольно запутанной. Для нашей следующей игры отлично подойдут ограничивающие сферы. Существует простой прием, который

мы уже применили в игре «Большой прыгун»: чтобы сферы более удачно вписывались, мы сделаем их меньше, чем графическое представление определенных объектов. На рис. 11.13 показана сфера космического корабля.

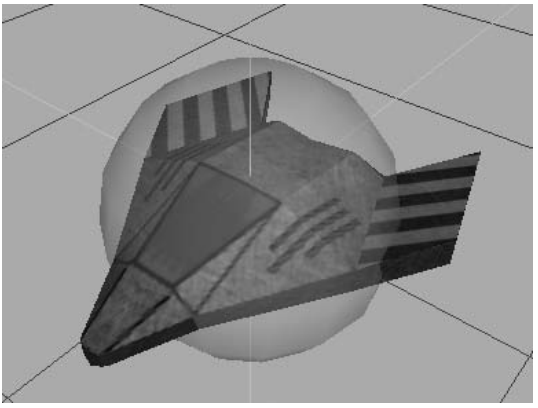

Рис. 11.13. Уменьшаем ограничивающую сферу для того, чтобы она лучше вписывалась в объект

Это, конечно, довольно дешевый трюк, но оказывается, что во многих ситуациях этого более чем достаточно, чтобы поддерживать иллюзию корректного определения столкновений.

Как же мы столкнем две сферы друг с другом? Говоря другими словами, как мы проверим, что сферы пересекаются? Это возможно определить точно так же, как и в случае с кругами! Все, что нам нужно, — измерить расстояние от центра одной сферы до центра другой сферы. Если оно меньше, чем сумма радиусов этих сфер — значит, эти сферы столкнулись. Создадим простой класс `Sphere` (листинг 11.14).

Листинг 11.14. Класс Sphere.java, простая ограничивающая сфера

```
package com.badlogic.androidgames.framework.math;

public class Sphere {
    public final Vector3 center = new Vector3();
    public float radius;

    public Sphere(float x, float y, float z, float radius) {
        this.center.set(x,y,z);
        this.radius = radius;
    }
}
```

Этот код точно такой же, что и в классе `Circle`. Мы изменили лишь вектор, хранящий координаты центра. Теперь он является экземпляром класса `Vector3`, а не `Vector2`.

Расширим также класс `OverlapTester` методами, проверяющими пересечение двух сфер и наличие точки внутри сферы. Его код содержится в листинге 11.15.

Листинг 11.15. Фрагменты класса OverlapTester.java, добавляем методы для тестирования сфер

```
public static boolean overlapSpheres(Sphere s1, Sphere s2) {
      float distance = s1.center.distSquared(s2.center);
      float radiusSum = s1.radius + s2.radius;
      return distance <= radiusSum * radiusSum;
}

public static boolean pointInSphere(Sphere c, Vector3 p) {
    return c.center.distSquared(p) < c.radius * c.radius;
}

public static boolean pointInSphere(Sphere c, float x, float y, float z) {
    return c.center.distSquared(x, y, z) < c.radius * c.radius;
}
```

Опять же этот код в точности повторяет аналогичный код класса `Circle`. Мы просто изменили тип центра сфер на `Vector3`.

ПРИМЕЧАНИЕ

Целые книги посвящены определению столкновений трехмерных объектов. Если вы хотите окунуться в этот интересный мир, я предложу вам книгу «Определение столкновений в реальном времени» (Real-time Collision Detection), написанную Кристером Эриксоном (Christer Ericson) и выпущенную издательством Morgan Kaufmann в 2005 году. Она должна быть на книжной полке любого уважающего себя разработчика игр!

Классы GameObject3D и DynamicGameObject3D

Теперь, когда мы определили ограничивающую фигуру для 3D-объектов, мы с легкостью можем написать эквиваленты классов `GameObject` и `DynamicGameObject`, использованных в 2D. Мы просто заменим все экземпляры класса `Vector2` на экземпляры класса `Vector3` и будем применять класс `Sphere` вместо класса `Rectangle`. В листинге 11.16 показан код класса `GameObject3D`.

Листинг 11.16. Класс GameObject3D, представление простого объекта, имеющего позицию и границы

```
package com.badlogic.androidgames.framework;

import com.badlogic.androidgames.framework.math.Sphere;
import com.badlogic.androidgames.framework.math.Vector3;

public class GameObject3D {
    public final Vector3 position;
    public final Sphere bounds;

    public GameObject3D(float x, float y, float z, float radius) {
        this.position = new Vector3(x,y,z);
        this.bounds = new Sphere(x, y, z, radius);
    }
}
```

Этот код очень тривиален, возможно, вам вообще не понадобится его объяснять. Единственный недостаток заключается в том, что нам приходится хранить одинаковую позицию дважды: один раз как член `position` класса `GameObject3D` и второй — внутри члена `position` экземпляра класса `Sphere`, содержащегося в классе `GameObject3D`. Это несколько неэлегантно, но для ясности мы будем придерживаться этого поведения.

Унаследовать класс `DynamicGameObject3D` от этого класса довольно просто. Код представлен в листинге 11.17.

Листинг 11.17. Класс DynamicGameObject3D.java, динамический аналог класса GameObject3D

```
package com.badlogic.androidgames.framework;

import com.badlogic.androidgames.framework.math.Vector3;

public class DynamicGameObject3D extends GameObject {
    public final Vector3 velocity;
    public final Vector3 accel;

    public DynamicGameObject3D(float x, float y, float z, float radius) {
        super(x, y, z, radius);
        velocity = new Vector3();
        accel = new Vector3();
    }
}
```

Опять же заменяем экземпляры класса `Vector2` на экземпляры класса `Vector3` и счастливо улыбаемся.

В 2D нам приходилось думать об отношениях между графическим представлением наших объектов (заданным в пикселах) и единицами измерения внутри модели нашего мира. В 3D мы свободны от этого! Вершины наших 3D-моделей, которые мы загружаем из, например, OBJ-файла, могут быть определены в любой измерительной системе. Нам более не нужно преобразовывать пикселы в единицы измерения мира и наоборот. Это упрощает работу в 3D. Нам нужно лишь обучить нашего художника рисовать такие модели, которые хорошо масштабируются к измерительной системе нашего мира.

Подводя итог

Снова мы рассмотрели множество тайн из мира программирования игр. Мы немного поговорили о векторах в 3D, которые, как оказалось, так же просты, как и их собратья в 2D. Общая идея: мы просто добавили координату по оси *z*! Мы также рассмотрели освещение в OpenGL ES. Написали вспомогательные классы, необходимые для представления материалов и источников освещения, с их помощью довольно просто можно установить освещение на сцене. Для лучшей производительности и сокращения количества графических артефактов реализовали

mip-текстурирование как часть класса `Texture`. Изучили реализацию простых камер с видом от первого и от третьего лица, написав совсем немного кода и воспользовавшись классом `Matrix`. Поскольку создание 3D-сетей вручную в коде утомительно, мы также рассмотрели один из наиболее простых и популярных форматов 3D-файлов: Wavefront OBJ. Мы снова рассмотрели старую физическую модель и перенесли ее в трехмерную реальность, это оказалось так же просто, как и создание 3D-векторов. Последним пунктом нашего плана была работа с ограничивающими фигурами и представлением объектов в 3D. Учитывая наши скромные потребности, мы выбрали очень простые решения для обеих проблем, которые практически идентичны тем, что мы использовали в 2D.

Хотя я мог бы представить еще многие аспекты, связанные с написанием игр в 3D, теперь вы имеете представление о том, как написать трехмерную игру. Секрет реализации заключается в том, что нет особой разницы между 2D- и 3D-игрой (конечно, если не сильно все усложнять). Более нам не стоит бояться 3D! В главе 12 мы используем полученные знания для написания последней игры этой книги — Droid Invaders.

12 Droid Invaders: большой финал

Наконец, мы готовы создать последнюю игру этой книги. В этот раз мы разработаем простую игру жанра Action/Arcade. Адаптируем старые приемы и придадим им новый вид в 3D, а также воспользуемся знаниями, полученными в последних двух главах.

Основы игровой механики

Как вы могли догадаться из названия этой главы, мы собираемся реализовать вариацию игры Space Invaders, которая в оригинале была двухмерной (рис. 12.1).

Рис. 12.1. Оригинальная игра Space Invaders

Вот небольшой сюрприз — большая часть игры останется в 2D. Все наши объекты будут иметь трехмерные границы в виде ограничивающих сфер, а также позиции в трехмерном пространстве. Однако перемещение будет происходить только в плоскости *xz*, что упростит многое при создании игры. На рис. 12.2 показан наш адаптированный к 3D мир игры Space Invaders. Макет был создан в программе Wings3D.

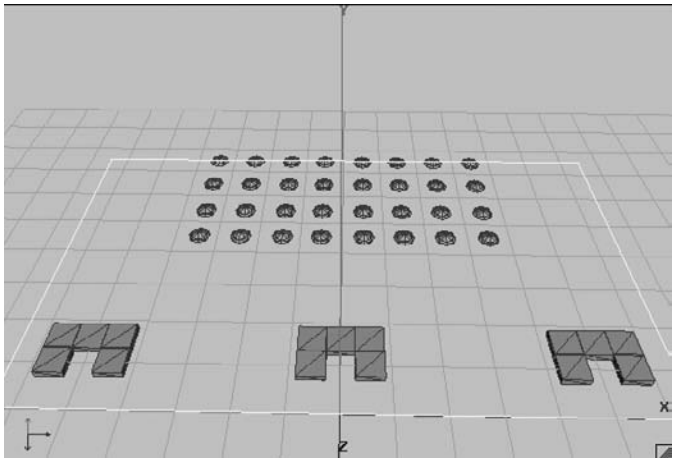

Рис. 12.2. Макет трехмерного игрового поля

Определим игровую механику.

- У нас есть корабль, летающий в нижней части игрового поля, способный перемещаться только вдоль оси *x*.
- Передвижение ограничено границами игрового мира. Когда корабль достигнет левой или правой границы игрового поля, он просто перестанет двигаться.
- Мы хотим предоставить игроку возможность выбрать способ управления кораблем — для перемещения влево и вправо может использоваться либо акселерометр, либо кнопки, расположенные на экране.
- Корабль может стрелять один раз в секунду. Игрок стреляет путем нажатия кнопки, расположенной на экране.
- В нижней части игрового поля размещаются три щита, каждый из которых состоит из пяти кубов.
- Захватчики появляются в конфигурации, показанной на рис. 12.2, затем перемещаются влево на некоторое расстояние, затем на некоторое расстояние вдоль оси *z*, а потом на некоторое расстояние вправо. Всего будет 32 захватчика, они выстроены в 4 ряда по 8 захватчиков.
- Захватчики будут случайным образом стрелять.
- Когда выстрел попадает в корабль, тот взрывается и теряет одну жизнь.
- Когда выстрел попадает в щит, тот пропадает навсегда.
- Когда выстрел попадает в захватчика, тот взрывается. Счет увеличивается на 10 очков.
- Когда все захватчики будут уничтожены, появится их новая волна. Они будут передвигаться чуть быстрее, чем предыдущая волна.
- Если захватчик напрямую столкнется с кораблем, игра закончится.
- Если корабль потеряет все свои жизни, игра закончится.

Не такой уж и большой список, не так ли? Все операции могут быть выполнены и в 2D (однако в плоскости *xz* вместо *xy*). Мы по-прежнему будем использовать трехмерные ограничивающие сферы. Возможно, вам захочется перенести игру полностью в 3D после того, как мы закончим первую фазу. Перейдем к сюжету игры.

Сюжет и художественное оформление

Мы назовем игру Droid Invaders, что будет отсылкой как и игре Space Invaders, так и к ОС Android. Это довольно дешевый трюк, но мы пока не планируем создать гениальное название прямо сейчас. По традиции классических шутеров вроде Doom сюжет игры будет очень коротким. Например, таким: пришельцы из космоса атакуют Землю; вы являетесь единственным человеком, способным противостоять силам врага. Такой сюжет вполне подошел бы для игр Doom и Quake, поэтому он подойдет и для нашей игры. Художественное оформление пользовательского интерфейса будет иметь оттенок ретро, в частности, будет использоваться тот же старомодный шрифт, который мы применяли в главе 9 для игры «Большой прыгун». Сам по себе игровой мир будет отображаться в 3D, он будет содержать текстурированные и освещенные 3D-модели. Рисунок 12.3 показывает, на что будет похож игровой экран.

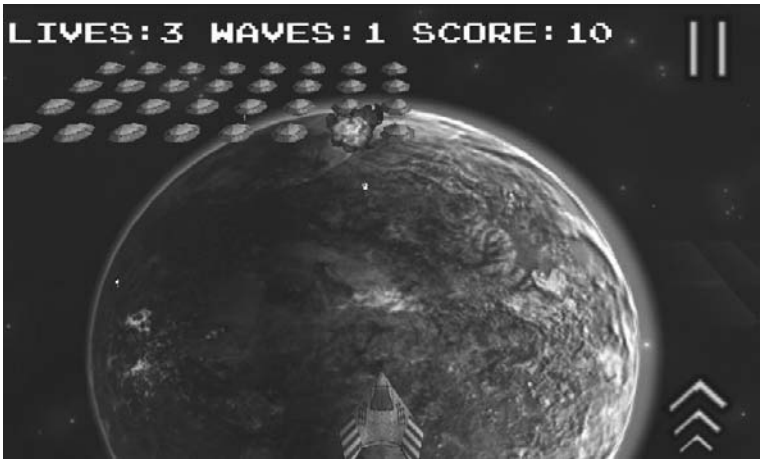

Рис. 12.3. Макет экрана игры Droid Invaders

В качестве музыкальной темы будет использоваться композиция в стиле рок/металл, а звуковые эффекты будут соответствовать сценарию.

Экраны и переходы

Поскольку мы уже дважды реализовали экраны помощи и лучших результатов, в главе 6 для игры «Мистер Ном» и в главе 9 для игры «Большой прыгун», не будем делать это и для игры Droid Invaders; всегда используется одинаковый принцип,

и игрок должен моментально понимать, что именно нужно делать, как только он увидит игровой экран. Вместо этого мы добавим экран настроек, который позволит игроку выбрать режим управления кораблем (мультитач или акселерометр) и отключить или включить звук. Вот список экранов игры Droid Invaders.

- Основной экран, на котором присутствуют логотип игры, а также пункты меню Play (Играть) и Settings (Настройки).
- Игровой экран, на котором моментально будет начинаться игра (больше никаких сигналов о готовности!), обрабатываться состояние паузы и отображаться фраза Game Over (Игра окончена), если у корабля более не останется жизней.
- Экран настроек, который отображает три значка, представляющие конфигурацию игры (мультитач, акселерометр и звук).

Этот список очень похож на те, которые мы составляли для предыдущих двух игр. На рис. 12.4 показаны все экраны и переходы.

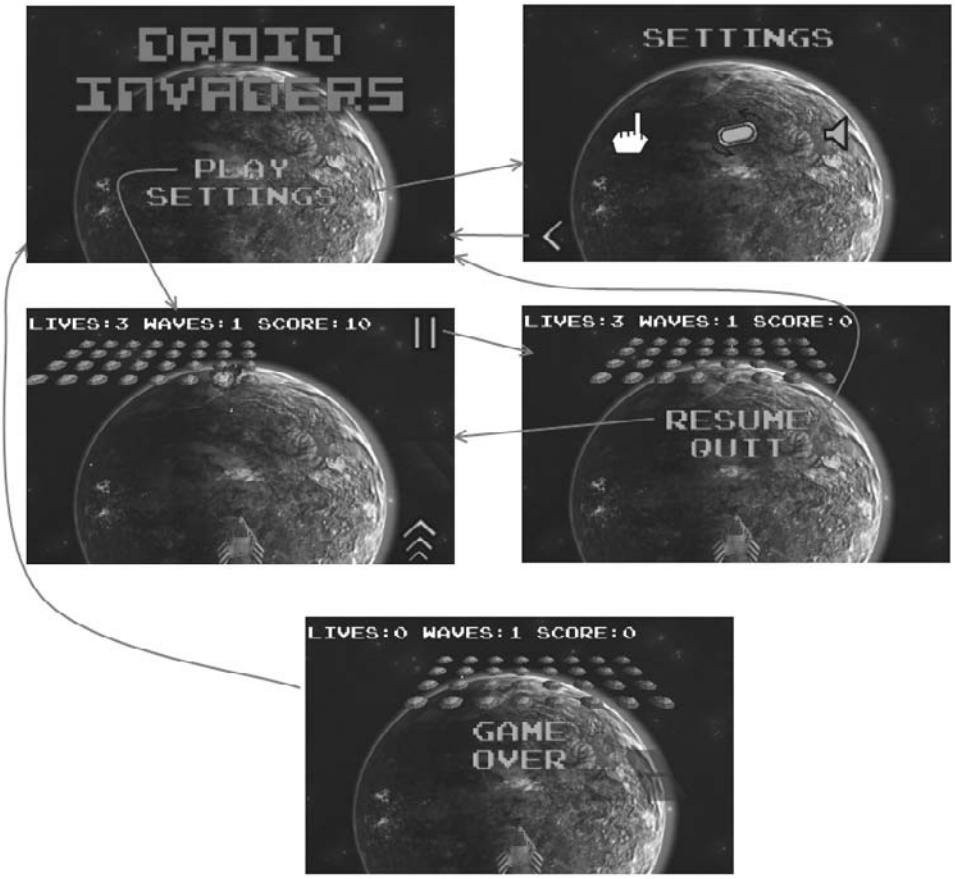

Рис. 12.4. Экраны и переходы игры Droid Invaders

Определение игрового мира

Одна из прелестей работы в 3D заключается в том, что мы освобождаемся от оков, в которые нас «заковали» пикселы. Мы можем определить для нашего мира любую единицу измерения. Наша игровая механика гласит о том, что игровое поле будет ограничено, поэтому для начала определим его размеры. На рис. 12.5 показана площадь поля нашего игрового мира.

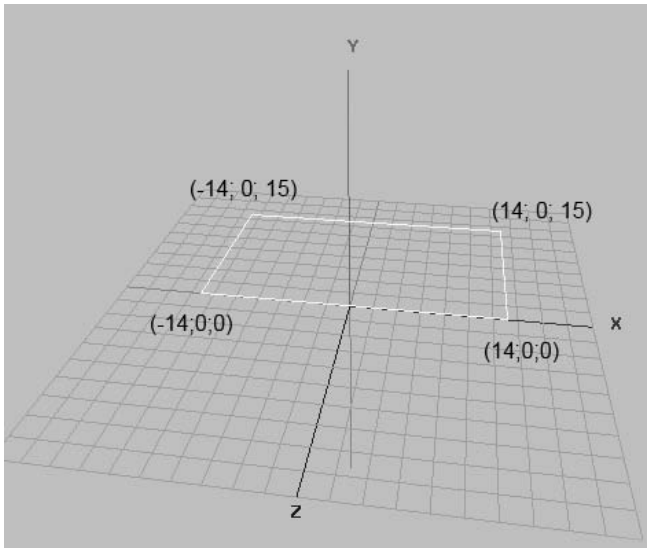

Рис. 12.5. Игровое поле

Все события нашего игрового мира будут происходить внутри этих границ в плоскости *xz*. Координаты по оси *x* будут ограничены значениями от –14 до 14, а по оси *z* — от 0 до –15. Корабль будет способен перемещаться вдоль нижней границы игрового поля, от точки (–14; 0; 0) до (14; 0; 0).

Далее нам следует определить размеры всех объектов нашего мира:

- Корабль будет иметь радиус 0,5 единицы.
- Захватчики будут иметь несколько больший радиус — 0,75 единицы. Так в них будет легче попасть.
- Каждый блок щита будет иметь радиус 0,5 единицы.
- Каждый выстрел будет иметь радиус, равный 0,1 единицы.

Как я получил эти значения? Я просто разделил игровой мир на клетки размером 1 × 1 единица и подумал о том, насколько большим должен быть каждый элемент по отношению к размеру игрового мира. Обычно вы получаете эти значения после небольших экспериментов или же используя единицы измерения реаль-

ного мира вроде метров. В игре Droid Invaders мы будем применять не метры, а безымянные единицы измерения.

Радиусы, которые мы только что определили, могут быть напрямую преобразованы в ограничивающие сферы. В случае блоков щита и корабля мы немного сжульничаем, поскольку они являются не совсем сферическими. Благодаря двухмерным свойствам нашего мира мы можем себе позволить этот небольшой трюк. В случае с пришельцами сфера — это очень хорошая аппроксимация их формы.

Нам также нужно определить скорости перемещающихся объектов.

- Корабль может перемещаться с максимальной скоростью, равной 20 единиц в секунду. Как и в игре «Большой прыгун», обычно он будет иметь меньшую скорость, ее значение будет зависеть от наклона телефона.
- Захватчики поначалу будут перемещаться со скоростью, равной 1 единице в секунду. Каждая волна будет двигаться чуть быстрее.
- Выстрелы будут иметь скорость, равную 10 единицам в секунду.

Используя эти определения, мы уже можем начать реализовывать логику нашего игрового мира. Оказывается, создание ресурсов непосредственно связано с единицами измерения, которые мы здесь определили.

Создание ресурсов

Как и в предыдущих играх, у нас есть два типа графических активов: элементы пользовательского интерфейса (например, логотипы или кнопки) и модели различных объектов нашей игры.

Активы пользовательского интерфейса

Мы снова будем создавать активы пользовательского интерфейса относительно целевого разрешения экрана. Наша игра будет работать в пейзажном режиме, поэтому мы просто выберем целевое разрешение равным 480 × 320 пикселов. Экраны на рис. 12.4 уже содержат все элементы, которые присутствуют в нашем пользовательском интерфейсе: логотип, различные пункты меню, несколько кнопок и пара строк текста. Для текста мы повторно используем шрифт, который уже применяли для игры «Большой прыгун». Мы уже собирали воедино все эти элементы в предыдущих играх, и вы уже знаете, что помещение их в атлас текстур положительно скажется на производительности. Поэтому рассмотрим атлас текстур, который будем использовать для игры Droid Invaders, содержащий все элементы пользовательского интерфейса, а также шрифт, который будет применяться на всех экранах игры. Атлас текстур показан на рис. 12.6.

Такую же концепцию мы использовали в игре «Большой прыгун». У нас также есть фоновое изображение, которое будет отрисовываться на всех экранах. Оно показано на рис. 12.7.

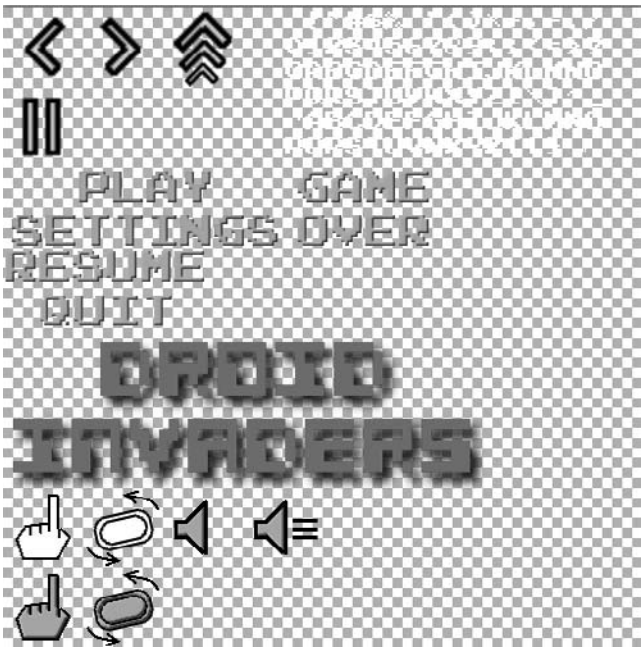

Рис. 12.6. Атлас, содержащий элементы пользовательского интерфейса, такие как кнопки, логотип и шрифт. Он хранится в файле items.png размером 512 × 512 пикселов

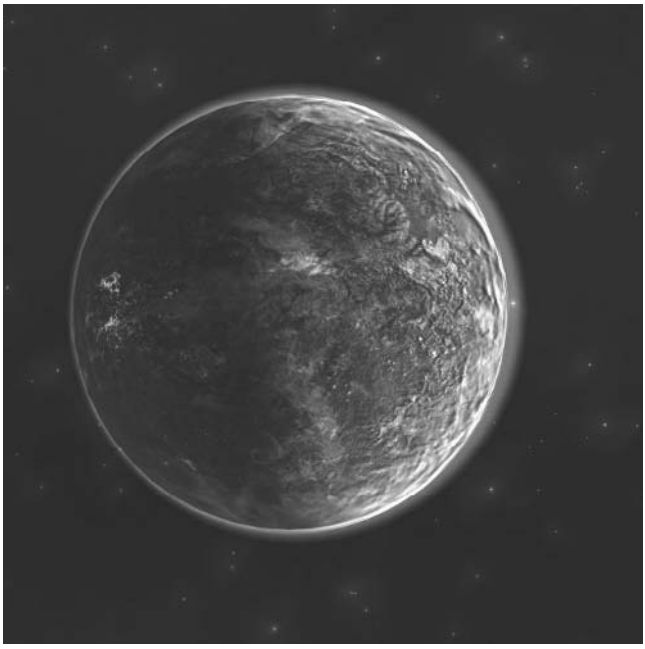

Рис. 12.7. Фоновое изображение, хранящееся в файле background.jpg размером 512 × 512 пикселов

Как вы можете увидеть на рис. 12.4, мы используем лишь верхний левый регион этого изображения для отрисовки полной картинки (480 × 320 пикселов).

Это все элементы пользовательского интерфейса, которые нам понадобятся. Рассмотрим 3D-модели и их текстуры.

Игровые ресурсы

Как я говорил в главе 11, в этой книге не будут рассматриваться способы создания 3D-моделей с помощью программ вроде Wings3D. Если вы хотите создать собственные модели, вам следует выбрать приложение, решающее подобные задачи, и пройти курс обучения работы с ним, который часто оказывается бесплатным и свободно доступным в Сети. Для создания моделей игры Droid Invaders я использовал программу Wings3D и просто экспортировал их в формате OBJ, который мы можем загрузить с помощью нашего фреймворка. Все модели состоят из треугольников и имеют координаты текстур и нормали. Для некоторых из них координаты текстуры не являются обязательным параметром, но им не повредит их иметь.

Модель корабля и его текстура показаны на рис. 12.8.

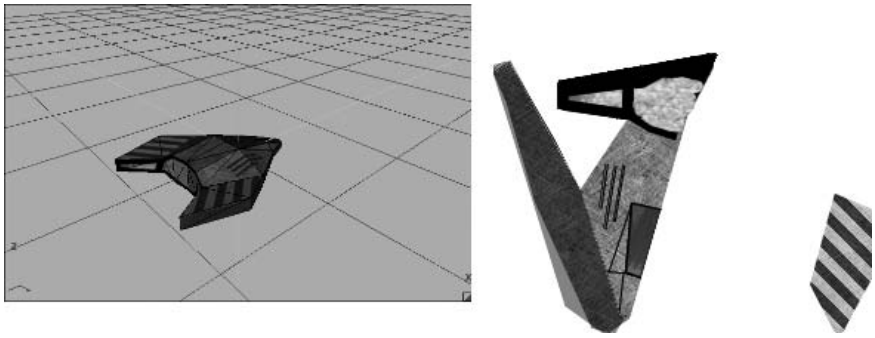

Рис. 12.8. Модель корабля, созданная в программе Wings3D (ship.obj), и ее текстура (ship.png, 256 × 256 пикселов)

Важным моментом является то, что корабль, изображенный на рис. 12.8, имеет лишь приблизительный «радиус», обозначенный нами в предыдущем разделе. Нам не нужно ничего масштабировать или преобразовывать размеры и позиции из одной системы координат в другую. Модель корабля определяется в тех же единицах измерения, что и его ограничивающая сфера.

На рис. 12.9 показана модель захватчика и его текстура.

Модель захватчика следует тем же принципам, что и модель корабля. У нас есть один OBJ-файл, хранящий позиции вершин, координаты текстуры, нормали и изображение текстуры.

Блоки щитов и выстрелы моделируются как кубы и хранятся в файлах `shield.obj` и `shot.obj`. Хотя к ним прикреплены координаты текстуры, мы на самом деле не накладываем текстуры при их отрисовке. Мы просто рисуем их как (полупрозрачные) объекты определенного цвета (синий в случае блоков, желтый в случае выстрелов).

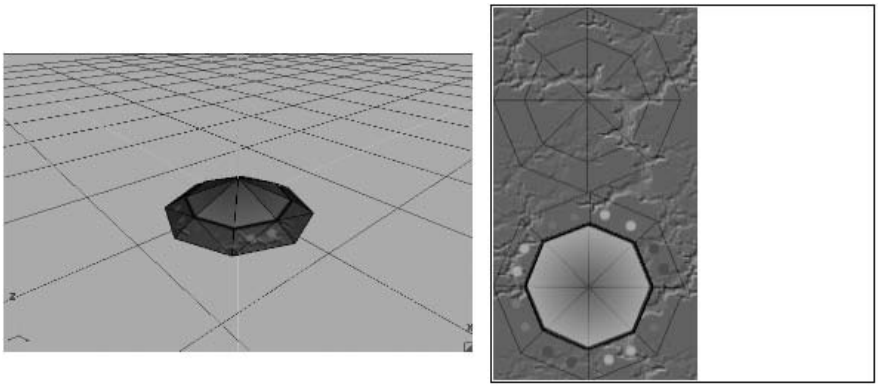

Рис. 12.9. Модель захватчика (invader.obj) и его текстура (invader.png, 256 × 256 пикселов)

Наконец, рассмотрим взрывы (см. рис. 12.3). Как нам их смоделировать? Нам это не нужно. Мы сделаем то же самое, что делали в 2D, и просто нарисуем прямоугольник с подходящей позицией по оси *z* в нашем 3D-мире, наложив на него текстуру одного кадра из текстурного изображения, содержащего анимацию взрыва. Этот же принцип использовался для создания анимированных объектов в игре «Большой прыгун». Единственное различие заключается в том, что мы будем рисовать прямоугольник в позиции по оси *z*, меньшей 0 (где находится взрывающийся объект). Мы даже можем заставить делать это класс `SpriteBatcher`. Ура OpenGL ES! На рис. 12.10 показана текстура.

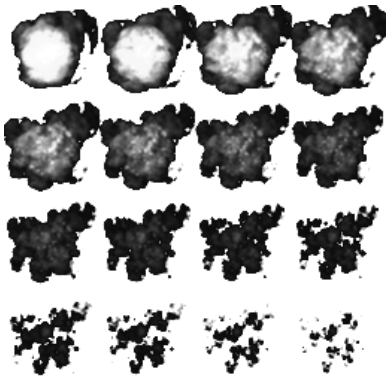

Рис. 12.10. Текстура, содержащая анимацию взрыва (explode.png, 256 × 256 пикселов)

Каждый кадр анимации имеет размер 64 × 64 пиксела. Все, что нам нужно, — сгенерировать текстурные регионы для каждого кадра и поместить их в экземпляр класса `Animation`, который мы можем использовать для получения корректного кадра для заданного времени анимации, точно так же, как мы делали это для игры «Большой прыгун».

Звук и музыка

Для создания звуковых эффектов я снова использовал программу sfxr. Звуковой эффект взрыва я нашел в Интернете. Это общедоступный звуковой эффект, поэтому мы можем использовать его в игре Droid Invaders. Музыку для игры я написал сам. С помощью настоящих инструментов. Да, я настолько старомоден. Вот список всех аудиофайлов игры Droid Invaders:

- `click.ogg` — звук щелчка, необходимый для пунктов меню и кнопок;
- `shot.ogg` — звук выстрела;
- `explosion.ogg` — взрыв;
- `music.mp3` — композиция в стиле рок/металл, которую я написал для игры Droid Invaders.

План разработки

После того как мы определили игровую механику, дизайн и ресурсы, мы можем начать программировать. Как обычно, создаем новый проект, копируем в него весь код нашего фреймворка, убеждаемся, что у нас есть подходящий манифест, значки и т. д. Весь код игры Droid Invaders будет помещен в пакет `com.badlogic.androidgames.droidinvaders`. Ресурсы будут храниться в директории `assets` проекта для ОС Android. Мы используем точно такую же структуру, которую применяли для игры «Большой прыгун»: стандартная активность будет наследовать от класса `GLGame`, несколько экземпляров класса `GLScreen` будут представлять собой различные экраны и переходы, как показано на рис. 12.4, классы для загрузки ресурсов и хранения настроек, а также классы для наших игровых объектов и класс, отрисовывающий наш игровой мир в 3D. Начнем с класса `Assets`.

Класс Assets

Мы уже делали это раньше, поэтому не ожидайте ничего удивительного. Код класса показан в листинге 12.1.

Листинг 12.1. Класс Assets.java, загружаем и сохраняем активы, как и обычно

```
package com.badlogic.androidgames.droidinvaders;

import com.badlogic.androidgames.framework.Music;
import com.badlogic.androidgames.framework.Sound;
import com.badlogic.androidgames.framework.gl.Animation;
import com.badlogic.androidgames.framework.gl.Font;
import com.badlogic.androidgames.framework.gl.ObjLoader;
import com.badlogic.androidgames.framework.gl.Texture;
```

```
import com.badlogic.androidgames.framework.gl.TextureRegion;
import com.badlogic.androidgames.framework.gl.Vertices3;
import com.badlogic.androidgames.framework.impl.GLGame;

public class Assets {
    public static Texture background;
    public static TextureRegion backgroundRegion;
    public static Texture items;
    public static TextureRegion logoRegion;
    public static TextureRegion menuRegion;
    public static TextureRegion gameOverRegion;
    public static TextureRegion pauseRegion;
    public static TextureRegion settingsRegion;
    public static TextureRegion touchRegion;
    public static TextureRegion accelRegion;
    public static TextureRegion touchEnabledRegion;
    public static TextureRegion accelEnabledRegion;
    public static TextureRegion soundRegion;
    public static TextureRegion soundEnabledRegion;
    public static TextureRegion leftRegion;
    public static TextureRegion rightRegion;
    public static TextureRegion fireRegion;
    public static TextureRegion pauseButtonRegion;
    public static Font font;
```

Здесь есть несколько членов, хранящих текстуры элементов пользовательского интерфейса и фонового изображения. Мы также храним несколько экземпляров класса `TextureRegions` и `Font`. Этого хватит для всех наших нужд, касающихся пользовательского интерфейса.

```
    public static Texture explosionTexture;
    public static Animation explosionAnim;
    public static Vertices3 shipModel;
    public static Texture shipTexture;
    public static Vertices3 invaderModel;
    public static Texture invaderTexture;
    public static Vertices3 shotModel;
    public static Vertices3 shieldModel;
```

У нас также есть текстуры и экземпляры класса `Vertices3`, хранящие модели и текстуры игровых объектов. Кроме того, мы имеем экземпляр класса `Animation`, в котором хранятся кадры анимации взрыва.

```
    public static Music music;
    public static Sound clickSound;
    public static Sound explosionSound;
    public static Sound shotSound;
```

Наконец, в классе есть несколько экземпляров классов `Music` и `Sound`, хранящих звуки игры.

```
    public static void load(GLGame game) {
        background = new Texture(game, "background.jpg", true);
```

```
backgroundRegion = new TextureRegion(background, 0, 0, 480, 320);
items = new Texture(game, "items.png", true);
logoRegion = new TextureRegion(items, 0, 256, 384, 128);
menuRegion = new TextureRegion(items, 0, 128, 224, 64);
gameOverRegion = new TextureRegion(items, 224, 128, 128, 64);
pauseRegion = new TextureRegion(items, 0, 192, 160, 64);
settingsRegion = new TextureRegion(items, 0, 160, 224, 32);
touchRegion = new TextureRegion(items, 0, 384, 64, 64);
accelRegion = new TextureRegion(items, 64, 384, 64, 64);
touchEnabledRegion = new TextureRegion(items, 0, 448, 64, 64);
accelEnabledRegion = new TextureRegion(items, 64, 448, 64, 64);
soundRegion = new TextureRegion(items, 128, 384, 64, 64);
soundEnabledRegion = new TextureRegion(items, 190, 384, 64, 64);
leftRegion = new TextureRegion(items, 0, 0, 64, 64);
rightRegion = new TextureRegion(items, 64, 0, 64, 64);
fireRegion = new TextureRegion(items, 128, 0, 64, 64);
pauseButtonRegion = new TextureRegion(items, 0, 64, 64, 64);
font = new Font(items, 224, 0, 16, 16, 20);
```

Метод `load()` начинается с создания всего необходимого для пользовательского интерфейса. Как и обычно, мы просто загружаем несколько текстур и создаем регионы.

```
explosionTexture = new Texture(game, "explode.png", true);
TextureRegion[] keyFrames = new TextureRegion[16];
int frame = 0;
for (int y = 0; y < 256; y += 64) {
    for (int x = 0; x < 256; x += 64) {
        keyFrames[frame++] = new TextureRegion(explosionTexture, x,
                                               y, 64, 64);
    }
}
explosionAnim = new Animation(0.1f, keyFrames);
```

Далее создаем экземпляр класса `Texture` для анимации взрыва, а также текстурные регионы для каждого кадра и экземпляр класса `Animation`. Мы просто проходим в цикле от верхнего левого угла до нижнего правого с шагом в 64 пиксела и создаем один экземпляр класса `TextureRegion` на каждый кадр. Все регионы далее передаются экземпляру класса `Animation`, каждый кадр которого длится 0,1 секунды.

```
shipTexture = new Texture(game, "ship.png", true);
shipModel = ObjLoader.load(game, "ship.obj");
invaderTexture = new Texture(game, "invader.png", true);
invaderModel = ObjLoader.load(game, "invader.obj");
shieldModel = ObjLoader.load(game, "shield.obj");
shotModel = ObjLoader.load(game, "shot.obj");
```

Далее загружаем модели и текстуры для корабля, захватчиков, щитов и выстрелов. Это довольно легко сделать с помощью могучего класса `ObjLoader`, не правда ли? Обратите внимание, для текстур мы используем mip-текстурирование.

```
music = game.getAudio().newMusic("music.mp3");
music.setLooping(true);
```

```
        music.setVolume(0.5f);
        if (Settings.soundEnabled)
            music.play();

        clickSound = game.getAudio().newSound("click.ogg");
        explosionSound = game.getAudio().newSound("explosion.ogg");
        shotSound = game.getAudio().newSound("shot.ogg");
    }
```

Наконец, мы загружаем музыку и звуковые эффекты игры. Вы можете видеть ссылку на класс `Settings`, являющийся практически аналогичным одноименным классам игр «Большой прыгун» и «Мистер Ном». Этот метод будет вызываться всего один раз, когда наша игра начнется, в классе `DroidInvaders`, который мы реализуем через мгновение. Как только все активы будут загружены, мы можем забыть о большинстве из них, кроме текстур, которые необходимо перезагружать в том случае, если игра приостановлена или возобновлена.

```
    public static void reload() {
        background.reload();
        items.reload();
        explosionTexture.reload();
        shipTexture.reload();
        invaderTexture.reload();
        if (Settings.soundEnabled)
            music.play();
    }
```

А это метод `reload()`. Мы будем вызывать его в методе `DroidInvaders.onResume()`, поэтому текстуры будут перезагружаться, а воспроизведение музыки будет возобновлено.

```
    public static void playSound(Sound sound) {
        if (Settings.soundEnabled)
            sound.play(1);
    }
}
```

Наконец, рассмотрим вспомогательный метод, несколько упрощающий воспроизведение звуковых эффектов, который присутствовал также в игре «Большой прыгун». Когда пользователь отключает звук в настройках, мы просто ничего не проигрываем в этом методе.

ПРИМЕЧАНИЕ

Хотя этот метод, загружающий активы и управляющий ими, довольно легко реализовать, он может разрастись до невероятных размеров, если у вас будет много активов. Необходимо также знать, что иногда не все активы могут быть загружены в память. Для простых игр вроде тех, что мы разработали ранее в этой книге, данный метод работает довольно хорошо. Я часто использую его в своих играх. Для больших игр вам необходимо разработать более эффективную стратегию управления активами.

Класс Settings

Как и в случае с классом Assets, мы можем использовать повторно тот код, который был написан для предыдущих игр. Теперь мы можем хранить дополнительную булеву переменную, которая сообщает нам о том, как именно пользователь хочет управлять кораблем, с помощью акселерометра или тачскрина. Не будет также реализована поддержка определения лучших результатов, поскольку мы их не отслеживаем. В качестве упражнения мы можете создать собственный экран лучших результатов и сохранение этих результатов на карту памяти. В листинге 12.2 показан код нашего класса.

Листинг 12.2. Класс Settings.java, наш старый знакомый

```
package com.badlogic.androidgames.droidinvaders;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

import com.badlogic.androidgames.framework.FileIO;

public class Settings {
    public static boolean soundEnabled = true;
    public static boolean touchEnabled = true;
    public final static String file = ".droidinvaders";
```

Мы храним информацию о том, включен ли звук, а также о способе управления кораблем. Настройки будут храниться в файле с расширением .droidinvaders на карте памяти.

```
    public static void load(FileIO files) {
        BufferedReader in = null;
        try {
            in = new BufferedReader(new
                        InputStreamReader(files.readFile(file)));
            soundEnabled = Boolean.parseBoolean(in.readLine());
            touchEnabled = Boolean.parseBoolean(in.readLine());
        } catch (IOException e) {
            // :( Все в порядке, у нас есть значения по умолчанию
        } catch (NumberFormatException e) {
            // :/ Все в порядке, у нас есть значения по умолчанию
        } finally {
            try {
                if (in != null)
                    in.close();
            } catch (IOException e) {
            }
        }
    }
```

В этом разделе нам не нужно ничего рассматривать, мы уже делали это раньше. Мы пытаемся считать два булевых значения из файла, расположенного на карте памяти. Если это сделать не удается, используем значения по умолчанию.

```
    public static void save(FileIO files) {
        BufferedWriter out = null;
        try {
            out = new BufferedWriter(new OutputStreamWriter(
                    files.writeFile(file)));
            out.write(Boolean.toString(soundEnabled));
            out.write("\n");
            out.write(Boolean.toString(touchEnabled));
        } catch (IOException e) {
        } finally {
            try {
                if (out != null)
                    out.close();
            } catch (IOException e) {
            }
        }
    }
}
```

Сохранение опять же довольно скучное. Мы просто сохраняем все, а если происходит ошибка, то молча ее игнорируем. Это также неплохой участок для улучшения — возможно, вы захотите дать пользователю понять, что что-то пошло не так.

Основная активность

Как обычно, у нас есть основная активность, которая наследует от класса `GLGame`. Она отвечает за загрузку активов с помощью вызова метода `Assets.load()` при запуске, а также за приостановку и возобновление воспроизведения музыки в том случае, если приостановлена/возобновлена активность. В качестве стартового экрана мы просто возвращаем `MainMenuScreen`, который вскоре также реализуем. Единственное, о чем следует помнить, — это определение активности в файле манифеста. Убедитесь в том, что установлена пейзажная ориентация экрана! Код класса приведен в листинге 12.3.

Листинг 12.3. Класс DroidInvaders.java, основная активность

```
package com.badlogic.androidgames.droidinvaders;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Screen;
```

```
import com.badlogic.androidgames.framework.impl.GLGame;

public class DroidInvaders extends GLGame {
    boolean firstTimeCreate = true;

    @Override
    public Screen getStartScreen() {
        return new MainMenuScreen(this);
    }

    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        super.onSurfaceCreated(gl, config);
        if (firstTimeCreate) {
            Settings.load(getFileIO());
            Assets.load(this);
            firstTimeCreate = false;
        } else {
            Assets.reload();
        }
    }

    @Override
    public void onPause() {
        super.onPause();
        if (Settings.soundEnabled)
            Assets.music.pause();
    }
}
```

Точно то же самое было в игре «Большой прыгун». В вызове метода `getStartScreen()` мы возвращаем новый экземпляр класса `MainMenuScreen`, который напишем далее. В методе `onSurfaceCreated()` мы убеждаемся, что активы перезагружены, а в методе `onPause()` мы приостанавливаем музыку, если она проигрывается.

Как вы можете видеть, большая часть кода повторяется, поскольку у нас есть хороший план того, как реализовать простую игру. Подумайте о том, насколько возможно уменьшить количество шаблонного кода, если переместить некоторые его фрагменты во фреймворк!

Экран главного меню

Мы уже написали множество простых экранов для предыдущих игр. В игре Droid Invaders также есть несколько подобных экранов. Принцип всегда одинаков: предоставляйте несколько элементов пользовательского интерфейса, на которых можно щелкнуть и запустить переход с одного экрана на другой, и отображайте важную информацию. На экране главного меню находятся логотип игры, а также

кнопки Play (Играть) и Settings (Настройки) (что показано на рис. 12.4). При прикосновении к одной из этих кнопок срабатывает переход на экран GameScreen или SettingsScreen. Код этого класса приведен в листинге 12.4.

Листинг 12.4. Класс MainMenuScreen.java, экран главного меню

```
package com.badlogic.androidgames.droidinvaders;

import java.util.List;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.gl.Camera2D;
import com.badlogic.androidgames.framework.gl.SpriteBatcher;
import com.badlogic.androidgames.framework.impl.GLScreen;
import com.badlogic.androidgames.framework.math.OverlapTester;
import com.badlogic.androidgames.framework.math.Rectangle;
import com.badlogic.androidgames.framework.math.Vector2;

public class MainMenuScreen extends GLScreen {
    Camera2D guiCam;
    SpriteBatcher batcher;
    Vector2 touchPoint;
    Rectangle playBounds;
    Rectangle settingsBounds;
```

Как обычно, нам необходима камера для того, чтобы установить окно просмотра и виртуальное целевое разрешение, равное 480×320 пикселов. Мы используем экземпляр класса SpriteBatcher для отрисовки элементов пользовательского интерфейса и фонового изображения. Экземпляры классов Vector2 и Rectangle помогут нам определить, было ли касание кнопки.

```
    public MainMenuScreen(Game game) {
        super(game);

        guiCam = new Camera2D(glGraphics, 480, 320);
        batcher = new SpriteBatcher(glGraphics, 10);
        touchPoint = new Vector2();
        playBounds = new Rectangle(240 - 112, 100, 224, 32);
        settingsBounds = new Rectangle(240 - 112, 100 - 32, 224, 32);
    }
```

В конструкторе устанавливаем камеру и экземпляр класса SpriteBatcher, ничего необычного. Создаем также экземпляры классов Vector2 и Rectangles, используя заданные позиции и размеры двух элементов на экране, имеющем целевое разрешение 480×320.

```
    @Override
    public void update(float deltaTime) {
        List<TouchEvent> events = game.getInput().getTouchEvents();
```

```
        int len = events.size();
        for(int i = 0; i < len; i++) {
            TouchEvent event = events.get(i);
            if(event.type != TouchEvent.TOUCH_UP)
                continue;

            guiCam.touchToWorld(touchPoint.set(event.x, event.y));
            if(OverlapTester.pointInRectangle(playBounds, touchPoint)) {
                Assets.playSound(Assets.clickSound);
                game.setScreen(new GameScreen(game));
            }
            if(OverlapTester.pointInRectangle(settingsBounds, touchPoint)) {
                Assets.playSound(Assets.clickSound);
                game.setScreen(new SettingsScreen(game));
            }
        }
    }
```

В методе `update()` получаем события прикосновения к экрану и проверяем их тип. Если появилось событие типа `touch-up`, преобразовываем его реальные координаты в координатную систему нашей камеры. Все, что нам остается сделать, — проверить, произошло ли прикосновение к одному из двух прямоугольников, ограничивающих пункты меню. Если один из них был задет, воспроизводим звук щелчка и переходим на соответствующий экран.

```
    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        guiCam.setViewportAndMatrices();

        gl.glEnable(GL10.GL_TEXTURE_2D);

        batcher.beginBatch(Assets.background);
        batcher.drawSprite(240, 160, 480, 320, Assets.backgroundRegion);
        batcher.endBatch();

        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);

        batcher.beginBatch(Assets.items);
        batcher.drawSprite(240, 240, 384, 128, Assets.logoRegion);
        batcher.drawSprite(240, 100, 224, 64, Assets.menuRegion);
        batcher.endBatch();

        gl.glDisable(GL10.GL_BLEND);
        gl.glDisable(GL10.GL_TEXTURE_2D);
    }
```

Метод `present()` выполняет те же самые операции, что применялись практически для всех экранов игры «Большой прыгун».

Мы очищаем экран и устанавливаем проекционную матрицу с помощью нашей камеры. Включаем текстурирование, а затем мгновенно отрисовываем фоновое изображение с помощью экземпляров классов SpriteBatcher и TextureRegion, определенных в классе Assets. Элементы меню полупрозрачны, поэтому мы включаем смешивание перед их отрисовкой.

```
    @Override
    public void pause() {
    }

    @Override
    public void resume() {
    }

    @Override
    public void dispose() {
    }
}
```

Остальная часть класса состоит из шаблонных методов, которые не делают ничего. Загрузка текстур происходит в активности DroidInvaders, поэтому классу MainMenuScreen заботиться более не о чем.

Экран настроек

Экран настроек предлагает игроку сменить способ управления кораблем, а также включить или отключить звук. На этом экране есть три различных значка (см. рис. 12.4). Прикосновение к значку, на котором изображена рука или наклоненный телефон, включит соответствующий способ управления кораблем. Значок выбранного способа будет иметь золотой цвет. Для значка, управляющего звуком, мы будем делать то же самое, что и в предыдущих играх.

Выбор пользователя будет отражаться установкой соответствующих значений булевых переменных в классе Settings. Мы также убеждаемся, что эти настройки мгновенно сохраняются на карту памяти при каждом их изменении, с помощью вызова метода Settings.save(). Код класса приведен в листинге 12.5.

Листинг 12.5. Класс SettingsScreen.java, экран настроек

```
package com.badlogic.androidgames.droidinvaders;

import java.util.List;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.gl.Camera2D;
import com.badlogic.androidgames.framework.gl.SpriteBatcher;
```

```
import com.badlogic.androidgames.framework.impl.GLScreen;
import com.badlogic.androidgames.framework.math.OverlapTester;
import com.badlogic.androidgames.framework.math.Rectangle;
import com.badlogic.androidgames.framework.math.Vector2;

public class SettingsScreen extends GLScreen {
    Camera2D guiCam;
    SpriteBatcher batcher;
    Vector2 touchPoint;
    Rectangle touchBounds;
    Rectangle accelBounds;
    Rectangle soundBounds;
    Rectangle backBounds;
```

Как обычно, у нас есть камера и экземпляр класса `SpriteBatcher`, необходимый для отрисовки элементов пользовательского интерфейса и фонового изображения. Для проверки того, производилось ли прикосновение к кнопке, мы также храним вектор и прямоугольники для трех кнопок, расположенных на экране.

```
    public SettingsScreen(Game game) {
        super(game);
        guiCam = new Camera2D(glGraphics, 480, 320);
        batcher = new SpriteBatcher(glGraphics, 10);
        touchPoint = new Vector2();
        touchBounds = new Rectangle(120 - 32, 160 - 32, 64, 64);
        accelBounds = new Rectangle(240 - 32, 160 - 32, 64, 64);
        soundBounds = new Rectangle(360 - 32, 160 - 32, 64, 64);
        backBounds = new Rectangle(32, 32, 64, 64);
    }
```

В конструкторе просто устанавливаем значения всех членов класса. Ничего необычного.

```
    @Override
    public void update(float deltaTime) {
        List<TouchEvent> events = game.getInput().getTouchEvents();
        int len = events.size();
        for (int i = 0; i < len; i++) {
            TouchEvent event = events.get(i);
            if (event.type != TouchEvent.TOUCH_UP)
                continue;

            guiCam.touchToWorld(touchPoint.set(event.x, event.y));
            if (OverlapTester.pointInRectangle(touchBounds, touchPoint)) {
                Assets.playSound(Assets.clickSound);
                Settings.touchEnabled = true;
                Settings.save(game.getFileIO());
            }
            if (OverlapTester.pointInRectangle(accelBounds, touchPoint)) {
                Assets.playSound(Assets.clickSound);
                Settings.touchEnabled = false;
```

```
                Settings.save(game.getFileIO());
            }
            if (OverlapTester.pointInRectangle(soundBounds, touchPoint)) {
                Assets.playSound(Assets.clickSound);
                Settings.soundEnabled = !Settings.soundEnabled;
                if (Settings.soundEnabled) {
                    Assets.music.play();
                } else {
                    Assets.music.pause();
                }
                Settings.save(game.getFileIO());
            }
            if (OverlapTester.pointInRectangle(backBounds, touchPoint)) {
                Assets.playSound(Assets.clickSound);
                game.setScreen(new MainMenuScreen(game));
            }
        }
    }
```

Метод `update()` получает события прикосновения к экрану и проверяет их тип. Если произошло событие `touch-up`, его координаты преобразовываются в координатную систему камеры. Затем проверяется вхождение этих координат в прямоугольники, ограничивающие кнопки, и определяется, какое именно действие следует предпринять.

```
    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        guiCam.setViewportAndMatrices();

        gl.glEnable(GL10.GL_TEXTURE_2D);

        batcher.beginBatch(Assets.background);
        batcher.drawSprite(240, 160, 480, 320, Assets.backgroundRegion);
        batcher.endBatch();

        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);

        batcher.beginBatch(Assets.items);
        batcher.drawSprite(240, 280, 224, 32, Assets.settingsRegion);
        batcher.drawSprite(120, 160, 64, 64,
                Settings.touchEnabled ? Assets.touchEnabledRegion
                        : Assets.touchRegion);
        batcher.drawSprite(240, 160, 64, 64,
                Settings.touchEnabled ? Assets.accelRegion
                        : Assets.accelEnabledRegion);
        batcher.drawSprite(360, 160, 64, 64,
                Settings.soundEnabled ? Assets.soundEnabledRegion
                        : Assets.soundRegion);
        batcher.drawSprite(32, 32, 64, 64, Assets.leftRegion);
```

```
        batcher.endBatch();

        gl.glDisable(GL10.GL_BLEND);
        gl.glDisable(GL10.GL_TEXTURE_2D);
    }
```

Метод `render()` выполняет те же самые действия, что и метод `MainMenuScreen.render()`. Мы отрисовываем фоновое изображение и кнопки, применяя по необходимости текстурирование и смешивание. Основываясь на текущих настройках, определяем, какой текстурный регион будет использоваться для отрисовки трех кнопок настроек.

```
    @Override
    public void pause() {
    }

    @Override
    public void resume() {
    }

    @Override
    public void dispose() {
    }
}
```

Остальная часть класса состоит из нескольких шаблонных методов, не имеющих никакой функциональности.

Перед тем как создавать класс `GameScreen`, нам следует сначала реализовать логику и отрисовку нашего мира. Шаблон MVC придет к нам на помощь.

Классы эмуляции

Как обычно, создаем классы для каждого объекта нашего мира. У нас в игре есть следующие объекты:

- корабль;
- захватчики;
- выстрелы;
- щиты.

Управление осуществляется всезнающим классом `World`. Как вы видели в предыдущей главе, при представлении объектов нет особой разницы между 2D и 3D. Вместо классов `GameObject` и `DynamicObject` мы теперь будем использовать классы `GameObject3D` и `DynamicObject3D`. Единственное различие заключается в том, что мы теперь для хранения позиций, скоростей и ускорений применяем экземпляры класса `Vector3` вместо экземпляров класса `Vector2`, а форму объектов представляют ограничивающие сферы вместо ограничивающих прямоугольников. Все, что нам осталось сделать, — реализовать поведение различных объектов нашего мира.

Класс Shield

Из определения игровой механики мы узнали размер и поведение блоков щита. Они просто неподвижно стоят на одной позиции, ожидая своей кончины от выстрела корабля или захватчика. Они не требуют серьезной разработки логики, поэтому их код довольно краток. В листинге 12.6 содержатся внутренности блока щита.

Листинг 12.6. Класс Shield.java, представляющий блок щита

```
package com.badlogic.androidgames.droidinvaders;

import com.badlogic.androidgames.framework.GameObject3D;

public class Shield extends GameObject3D {
    static float SHIELD_RADIUS = 0.5f;

    public Shield(float x, float y, float z) {
        super(x, y, z, SHIELD_RADIUS);
    }
}
```

Мы определили радиус щита и инициализировали его позицию и ограничивающую сферу соответственно параметрам конструктора. На этом все!

Класс Shot

Класс выстрела также довольно прост. Он наследует от класса `DynamicGameObject3D`, поскольку он на самом деле перемещается. Код класса `Shot` содержится в листинге 12.7.

Листинг 12.7. Класс Shot.java, представляющий выстрел

```
package com.badlogic.androidgames.droidinvaders;

import com.badlogic.androidgames.framework.DynamicGameObject3D;

public class Shot extends DynamicGameObject3D {
    static float SHOT_VELOCITY = 10f;
    static float SHOT_RADIUS = 0.1f;

    public Shot(float x, float y, float z, float velocityZ) {
        super(x, y, z, SHOT_RADIUS);
        velocity.z = velocityZ;
    }

    public void update(float deltaTime) {
        position.z += velocity.z * deltaTime;
        bounds.center.set(position);
    }
}
```

Мы снова определяем несколько констант, представляющих собой скорость выстрела и его радиус. Конструктор принимает в качестве параметра исходную позицию выстрела, а также его скорость по оси *z*. Но подождите, разве мы не определили скорость как константу? Да, но это позволило бы перемещаться снарядам только вдоль положительной части оси *z*. Это годится для выстрелов захватчиков, но выстрелы нашего корабля должны передвигаться в противоположном направлении. Когда мы создаем выстрел (вне этого класса), мы знаем, в каком направлении должен двигаться выстрел. Поэтому его скорость задается его создателем.

Метод `update()` просто выполняет привычные нам физические задачи. Для выстрела не используется ускорение, и поэтому нам нужно добавить к позиции выстрела константную скорость, умноженную на прошедшее время. Главный момент заключается в том, что мы также обновляем позицию центра ограничивающей сферы в соответствии с позицией выстрела. В противном случае ограничивающая сфера не будет перемещаться вместе с выстрелом.

Класс Ship

Класс `Ship` отвечает за обновление позиции корабля, ограничивая ее границами игрового поля. Кроме того, он отслеживает текущее состояние корабля. Он может быть либо целым, либо взрывающимся. В обоих случаях мы отслеживаем промежуток времени, в течение которого корабль находился в этом состоянии. Впоследствии он будет использован для создания анимации, в качестве примера можно взять игру «Большой прыгун» и ее класс `WorldRenderer`. Корабль будет получать текущее значение скорости извне, основываясь на данных, поступивших от пользователя. Это будут или показания акселерометра (в случае игры «Большой прыгун») или константа в зависимости от кнопки, нажатой пользователем.

В дополнение корабль будет отслеживать количество жизней и предлагать нам способ сообщать ему о том, что он был убит. Код класса корабля содержится в листинге 12.8.

Листинг 12.8. Класс Ship.java, представляющий корабль

```
package com.badlogic.androidgames.droidinvaders;

import com.badlogic.androidgames.framework.DynamicGameObject3D;

public class Ship extends DynamicGameObject3D {
    static float SHIP_VELOCITY = 20f;
    static int SHIP_ALIVE = 0;
    static int SHIP_EXPLODING = 1;
    static float SHIP_EXPLOSION_TIME = 1.6f;
    static float SHIP_RADIUS = 0.5f;
```

Мы начинаем с описания нескольких констант, определяющих максимальную скорость корабля, два состояния (целый и взрывающийся), время, необходимое кораблю для того, чтобы полностью взорваться, а также радиус ограничивающей сферы. Этот класс также наследует от класса `DynamicGameObject3D`, поскольку у него

есть позиция, ограничивающая сфера и скорость. Вектор ускорения, хранящийся в классе DynamicGameObject3D, снова не используется.

```
int lives;
int state;
float stateTime = 0;
```

Далее следуют две переменные типа int, необходимые для отслеживания количества жизней и состояния корабля (либо SHIP_ALIVE, либо SHIP_EXPLODING). Последний член класса отслеживает количество секунд, которое корабль провел в текущем состоянии.

```
public Ship(float x, float y, float z) {
    super(x, y, z, SHIP_RADIUS);
    lives = 3;
    state = SHIP_ALIVE;
}
```

В конструкторе осуществляется уже привычный вызов конструктора суперкласса, а также инициализируются некоторые члены класса. Всего корабль будет иметь три жизни.

```
public void update(float deltaTime, float accelY) {
    if (state == SHIP_ALIVE) {
        velocity.set(accelY / 10 * SHIP_VELOCITY, 0, 0);
        position.add(velocity.x * deltaTime, 0, 0);
        if (position.x < World.WORLD_MIN_X)
            position.x = World.WORLD_MIN_X;
        if (position.x > World.WORLD_MAX_X)
            position.x = World.WORLD_MAX_X;
        bounds.center.set(position);
    } else {
        if (stateTime >= SHIP_EXPLOSION_TIME) {
            lives--;
            stateTime = 0;
            state = SHIP_ALIVE;
        }
    }
    stateTime += deltaTime;
}
```

Метод update() довольно прост. Он принимает промежуток времени, а также показания акселерометра по оси y (помните, игра работает в пейзажном режиме, поэтому ось y акселерометра является нашей осью x). Если корабль цел, мы устанавливаем его скорость соответственно показаниям акселерометра (которые будут лежать в промежутке от –10 до 10) так же, как мы поступали и в случае с игрой «Большой прыгун». В дополнение обновляем позицию корабля, основываясь на его текущей скорости. Далее проверяем, покинул ли корабль границы игрового поля, используя две константы, которые определим далее в классе World. Когда

позиция корабля установлена, мы, наконец, можем обновить позицию ограничивающей сферы корабля.

Если корабль взрывается, мы проверяем, как долго он находится в этом состоянии. После 1,6 секунды, проведенных в этом состоянии, корабль заканчивает взрываться, теряет одну жизнь и переходит обратно в «целое» состояние.

Наконец, обновляем параметр `stateTime`, основываясь на заданном промежутке времени.

```
    public void kill() {
        state = SHIP_EXPLODING;
        stateTime = 0;
        velocity.x = 0;
    }
}
```

Последний метод `kill()` может быть вызван классом `World`, если он определит, что произошло и столкновение между кораблем и выстрелом или захватчиком. Он установит состояние «взрывающийся», сбросит время состояния, а также убедится, что текущая скорость корабля равна нулю по всем осям (мы никогда не устанавливаем y- и z-компоненты вектора скорости, поскольку корабль перемещается только по оси x).

Класс Invader

Согласно шаблону, определенному ранее, захватчики просто парят в космосе. Этот шаблон приведен на рис. 12.11.

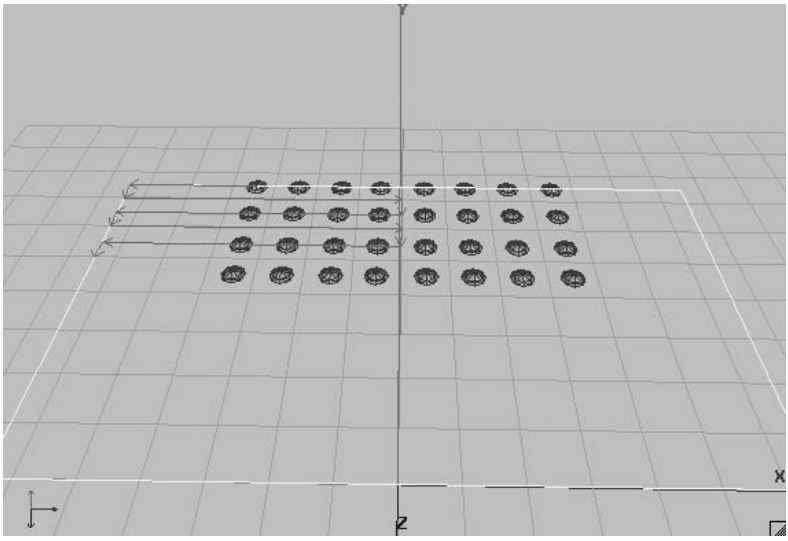

Рис. 12.11. Передвижение захватчиков: влево, вниз, вправо, вниз, влево, вниз, вправо, вниз…

Захватчик следует очень примитивному шаблону передвижения. Из своей исходной позиции он сначала перемещается на некоторое расстояние влево. Далее он спускается вниз (что означает направление в сторону положительной части оси *z* на игровом поле), опять же на определенное расстояние. После того как это закончится, он начнет перемещаться вправо, возвращаясь в позицию с той же координатой по оси *x*, с которой он начинал движение влево.

Расстояния, на которые захватчик перемещается вправо и влево, всегда одинаковы, кроме начала.

Рис. 12.11 иллюстрирует передвижение левого верхнего захватчика. Расстояние, на которое он переместится влево в первый раз, меньше, чем то, на которое он будет двигаться в стороны каждый последующий раз. Расстояние, на которое захватчик перемещается по горизонтали, равно половине игрового поля, в нашем случае — 14 единиц. При первом перемещении по горизонтали захватчик переместится на половину этого расстояния — 7 единиц.

Нам нужно отслеживать направление перемещения захватчика, а также то, насколько он уже переместился в этом направлении. Если он передвигается на максимальное расстояние (14 единиц для горизонтального перемещения, 1 единица для вертикального перемещения), он переходит в следующий режим перемещения. Расстояние, на которое передвигаются все захватчики, равно половине игрового поля. Взгляните снова на рис. 12.11, чтобы убедиться, почему это работает! В результате захватчики будут отскакивать от левой и правой границ игрового поля.

Захватчики также будут иметь постоянную скорость. Вообще, на самом деле скорость будет увеличиваться всякий раз, когда будет генерироваться новая волна захватчиков (это происходит в том случае, если все захватчики текущей волны уже уничтожены). Мы можем реализовать это с помощью простого умножения данной стандартной скорости на некоторую константу, значение которой устанавливается классом `World`, ответственным на обновление всех захватчиков.

Наконец, мы также отслеживаем текущее состояние захватчика, который также может быть как целым, так и взрывающимся. Мы используем тот механизм, который применяли для корабля (мы задействовали состояние и время, которое он в нем находился). Код класса захватчика содержится в листинге 12.9.

Листинг 12.9. Класс `Invader.java`, представляющий захватчика

```
package com.badlogic.androidgames.droidinvaders;

import com.badlogic.androidgames.framework.DynamicGameObject3D;

public class Invader extends DynamicGameObject3D {
    static final int INVADER_ALIVE = 0;
    static final int INVADER_DEAD = 1;
    static final float INVADER_EXPLOSION_TIME = 1.6f;
    static final float INVADER_RADIUS = 0.75f;
    static final float INVADER_VELOCITY = 1;
    static final int MOVE_LEFT = 0;
    static final int MOVE_DOWN = 1;
    static final int MOVE_RIGHT = 2;
```

Начнем с описания нескольких констант, определяющих состояние захватчика, продолжительность взрыва, его радиус и скорость по умолчанию, а также три константы, позволяющие нам отслеживать, в каком направлении захватчик перемещается в данный момент.

```
int state = INVADER_ALIVE;
float stateTime = 0;
int move = MOVE_LEFT;
boolean wasLastStateLeft = true;
float movedDistance = World.WORLD_MAX_X / 2;
```

Мы отслеживаем состояние захватчика, время состояния, направление передвижения и расстояние, на которое переместился захватчик, равное сначала половине ширины игрового поля. Мы также отслеживаем, перемещался ли захватчик в последний раз влево. Это позволяет нам решить, в каком направлении будет двигаться захватчик после того, как завершит перемещение по оси *z*.

```
public Invader(float x, float y, float z) {
    super(x, y, z, INVADER_RADIUS);
}
```

В конструкторе происходит обычная установка позиции захватчика, а также задание его границ при помощи конструктора суперкласса.

```
public void update(float deltaTime, float speedMultiplier) {
    if (state == INVADER_ALIVE) {
        movedDistance += deltaTime * INVADER_VELOCITY * speedMultiplier;
        if (move == MOVE_LEFT) {
            position.x -= deltaTime * INVADER_VELOCITY *
                                      speedMultiplier;
            if (movedDistance > World.WORLD_MAX_X) {
                move = MOVE_DOWN;
                movedDistance = 0;
                wasLastStateLeft = true;
            }
        }
        if (move == MOVE_RIGHT) {
            position.x += deltaTime * INVADER_VELOCITY *
                                      speedMultiplier;
            if (movedDistance > World.WORLD_MAX_X) {
                move = MOVE_DOWN;
                movedDistance = 0;
                wasLastStateLeft = false;
            }
        }
        if (move == MOVE_DOWN) {
            position.z += deltaTime * INVADER_VELOCITY *
                                      speedMultiplier;
            if (movedDistance > 1) {
                if (wasLastStateLeft)
                    move = MOVE_RIGHT;
                else
```

```
                    move = MOVE_LEFT;
                movedDistance = 0;
            }
        }

        bounds.center.set(position);
    }

    stateTime += deltaTime;
}
```

Метод `update()` принимает текущий промежуток времени и множитель скорости, который используется для ускорения новых волн захватчиков. Перемещение осуществляется только в том случае, если захватчик еще жив.

Мы начинаем с расчета того, на сколько единиц захватчик переместится в этом обновлении, и соответственно увеличиваем значение переменной `movedDistance`. Если он перемещается влево, мы обновляем позицию, вычитая скорость из координаты по оси *x*, умноженную на изменение времени и множитель скорости. Если захватчик переместился достаточно далеко, мы указываем ему переместиться по вертикали, устанавливая значение переменной `move` равным `MOVE_DOWN`. Мы также устанавливаем значение переменной `wasLastStateLeft` равным `true` для того, чтобы знать, что при следующем перемещении по горизонтали необходимо будет двигаться вправо.

То же самое мы делаем и при обработке движения вправо. Единственное различие заключается в том, что мы добавляем скорость перемещения к координате позиции по оси *x* и устанавливаем значение переменной `wasLastStateLeft` равным `false` в тот момент, когда перемещение закончено.

Если захватчик перемещается вниз, мы работаем с *z*-координатой его позиции и опять же проверяем, как далеко он уже передвинулся. Если он переместился на максимальное расстояние, мы переключаем состояние на `MOVE_LEFT` или `MOVE_RIGHT` в зависимости от направления последнего перемещения по горизонтали, хранящегося в переменной `wasLastStateLeft`. Как только мы обновили позиции захватчиков, мы можем установить позицию ограничивающей сферы так же, как мы делали это для корабля. Наконец, обновляем текущее время состояния и считаем, что обновление завершено.

```
    public void kill() {
        state = INVADER_DEAD;
        stateTime = 0;
    }
}
```

Метод `kill()` нужен для того же, для чего служит одноименный метод класса `Ship`. Он позволяет сообщить захватчику, что он должен начать умирать. Мы устанавливаем его состояние равным `INVADER_DEAD` и сбрасываем время состояния. Захватчик больше не будет перемещаться и будет только обновлять свое состояние, основываясь на текущем изменении времени.

Класс World

Класс World является самым главным классом. Он хранит экземпляры классов корабля, захватчиков и выстрелов, а также отвечает за их обновление и проверку столкновений. Он делает то же самое, что и в игре «Большой прыгун», с небольшими различиями. Исходное размещение блоков щита и захватчиков — это также обязанность этого класса. Мы также создаем экземпляр класса WorldListener для того, чтобы информировать всех слушателей о событиях, произошедших внутри мира, таких как выстрел или взрыв. Он будет воспроизводить звуковые эффекты, как и в игре «Большой прыгун». Рассмотрим каждый метод этого класса подробнее. Код содержится в листинге 12.10.

Листинг 12.10. Класс World.java, представляющий собой игровой мир. Он связывает все воедино

```
package com.badlogic.androidgames.droidinvaders;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import com.badlogic.androidgames.framework.math.OverlapTester;

public class World {
    public interface WorldListener {
        public void explosion();

        public void shot();
    }
```

Мы хотим, чтобы внешние слушатели класса знали о событиях, произошедших внутри него, таких как взрыв или выстрел. Для этого мы определяем интерфейс слушателей, который можем реализовать и зарегистрировать с экземпляром World, который будет вызываться всякий раз, когда выполняется соответствующее событие. Точно то же самое происходило и в игре «Большой прыгун», только события были другими.

```
    final static float WORLD_MIN_X = -14;
    final static float WORLD_MAX_X = 14;
    final static float WORLD_MIN_Z = -15;
```

В классе также имеется несколько констант, определяющих границы нашего мира. Их обсудили мы в разделе «Определение игрового мира».

```
    WorldListener listener;
    int waves = 1;
    int score = 0;
    float speedMultiplier = 1;
    final List<Shot> shots = new ArrayList<Shot>();
    final List<Invader> invaders = new ArrayList<Invader>();
```

```
final List<Shield> shields = new ArrayList<Shield>();
final Ship ship;
long lastShotTime;
Random random;
```

Этот класс отслеживает множество вещей. В нем есть слушатель, который будет вызываться всякий раз, когда происходит взрыв или выстрел. Он также отслеживает, сколько волн пришельцев игрок уже уничтожил. Переменная `score` отвечает за текущий счет, а параметр `speedMultiplier` позволяет ускорить движение захватчиков (вспомните метод `Invaders.update()`). Кроме того, здесь хранятся списки выстрелов, захватчиков и блоков щита, которые в данный момент существуют в игровом мире. Наконец, здесь есть экземпляр класса `Ship`, а также время, в которое был произведен последний выстрел. Это время представляется в наносекундах, поскольку метод `System.nanoTime()` возвращает время в таком формате, и хранится в переменной типа `long`. Экземпляр класса `Random` пригодится, когда нам понадобится определить, выстрелит ли захватчик.

```
public World() {
    ship = new Ship(0, 0, 0);
    generateInvaders();
    generateShields();
    lastShotTime = System.nanoTime();
    random = new Random();
}
```

В конструкторе создается корабль на своей исходной позиции, генерируются захватчики и щиты, а также инициализируются остальные члены класса.

```
private void generateInvaders() {
    for (int row = 0; row < 4; row++) {
        for (int column = 0; column < 8; column++) {
            Invader invader = new Invader(-WORLD_MAX_X / 2 + column *
                    2f, 0, WORLD_MIN_Z + row * 2f);
            invaders.add(invader);
        }
    }
}
```

Метод `generateInvaders()` просто создает сетку размером 8×4, которая состоит из пришельцев, выстроенных так, как показывает рис. 12.11.

```
private void generateShields() {
    for (int shield = 0; shield < 3; shield++) {
        shields.add(new Shield(-10 + shield * 10 – 1, 0, -3));
        shields.add(new Shield(-10 + shield * 10 + 0, 0, -3));
        shields.add(new Shield(-10 + shield * 10 + 1, 0, -3));
        shields.add(new Shield(-10 + shield * 10 – 1, 0, -2));
        shields.add(new Shield(-10 + shield * 10 + 1, 0, -2));
    }
}
```

Метод generateShields() выполняет те же самые задачи — он создает 3 щита, каждый из которых состоит из 5 блоков, выстроенных так, как показано на рис. 12.2.

```
public void setWorldListener(WorldListener worldListener) {
    this.listener = worldListener;
}
```

В классе World есть также метод, устанавливающий слушателя.

```
public void update(float deltaTime, float accelX) {
    ship.update(deltaTime, accelX);
    updateInvaders(deltaTime);
    updateShots(deltaTime);

    checkShotCollisions();
    checkInvaderCollisions();

    if (invaders.size() == 0) {
        generateInvaders();
        waves++;
        speedMultiplier += 0.5f;
    }
}
```

Метод update() удивительно прост. Он принимает текущее изменение времени, а также данные с акселерометра по оси *y*, которые мы передаем методу Ship.update(). Как только обновится корабль, вызываются методы updateInvaders() и updateShots(), которые ответственны за обновление захватчиков и выстрелов. После того как все объекты мира обновились, мы можем начать проверку того, произошло ли столкновение. Метод checkShotCollision() проверит на столкновения все выстрелы, корабль и захватчиков. Наконец, мы проверяем, уничтожены ли все захватчики. Если да, то генерируется их новая волна. К радости сборщика мусора мы могли бы повторно использовать старые экземпляры класса Invader, например с помощью экземпляра класса Pool. Однако для простоты мы создаем новые экземпляры класса Invader. Кстати, то же самое верно и для выстрелов. Поскольку за одну игру может быть создано не так уж много объектов, сборщик мусора не будет часто запускаться. Но если вы хотите предотвратить все возможные проблемы, просто используйте класс Pool, чтобы повторно применять погибших захватчиков и выстрелы. Обратите также внимание на то, что в этом методе мы увеличиваем множитель скорости!

```
private void updateInvaders(float deltaTime) {
    int len = invaders.size();
    for (int i = 0; i < len; i++) {
        Invader invader = invaders.get(i);
        invader.update(deltaTime, speedMultiplier);

        if (invader.state == Invader.INVADER_ALIVE) {
            if (random.nextFloat() < 0.001f) {
                Shot shot = new Shot(invader.position.x,
```

```
                                invader.position.y,
                                invader.position.z,
                                Shot.SHOT_VELOCITY);
                        shots.add(shot);
                        listener.shot();
                    }
                }

                if (invader.state == Invader.INVADER_DEAD &&
                       invader.stateTime > Invader.INVADER_EXPLOSION_TIME) {
                    invaders.remove(i);
                    i--;
                    len--;
                }
            }
        }
```

У метода `updateInvaders()` есть несколько обязанностей. Он в цикле проходит по всем захватчикам и вызывает их методы `update()`. Как только экземпляр класса `Invader` обновился, мы проверяем, жив ли он к этому моменту. Если да, то генерируется случайное число, представляющее собой шанс на выстрел. Если оно меньше 0,001, то захватчик производит выстрел. Это означает, что каждый захватчик имеет 0,1 % шанса на выстрел каждый кадр. Если шанс срабатывает, создается новый экземпляр класса `Shot`, его скорость устанавливается таким образом, что он начинает перемещаться вдоль положительной части оси *z*. Об этом событии информируется слушатель. Если захватчик умирает и заканчивает взрываться, мы просто удаляем его из списка захватчиков.

```
    private void updateShots(float deltaTime) {
        int len = shots.size();
        for (int i = 0; i < len; i++) {
            Shot shot = shots.get(i);
            shot.update(deltaTime);
            if (shot.position.z < WORLD_MIN_Z ||
                shot.position.z > 0) {
                shots.remove(i);
                i--;
                len--;
            }
        }
    }
```

Метод `updateShots()` также очень прост. Он проходит в цикле по всем выстрелам, обновляет их и проверяет, покинул ли каждый из них игровое поле. В этом случае он просто удаляется из списка выстрелов.

```
    private void checkInvaderCollisions() {
        if (ship.state == Ship.SHIP_EXPLODING)
            return;

        int len = invaders.size();
```

```
        for (int i = 0; i < len; i++) {
            Invader invader = invaders.get(i);
            if (OverlapTester.overlapSpheres(ship.bounds, invader.bounds)) {
                ship.lives = 1;
                ship.kill();
                return;
            }
        }
    }
```

В методе `checkInvaderCollisions()` происходит проверка того, столкнулся ли какой-нибудь захватчик с кораблем. Это довольно просто, поскольку все, что нужно сделать, — пройти в цикле по всем захватчикам и проверить, пересекается ли их ограничивающая сфера с ограничивающей сферой корабля. В соответствии с игровой механикой, когда это происходит, игра заканчивается. Поэтому мы устанавливаем количество жизней корабля равным 1 перед тем, как вызвать метод `Ship.kill()`. После этого вызова параметр `live` корабля устанавливается равным 0, что будет использовано в другом методе для проверки наступления состояния конца игры.

```
    private void checkShotCollisions() {
        int len = shots.size();
        for (int i = 0; i < len; i++) {
            Shot shot = shots.get(i);
            boolean shotRemoved = false;

            int len2 = shields.size();
            for (int j = 0; j < len2; j++) {
                Shield shield = shields.get(j);
                if (OverlapTester.overlapSpheres(shield.bounds,
                                                 shot.bounds)) {
                    shields.remove(j);
                    shots.remove(i);
                    i--;
                    len--;
                    shotRemoved = true;
                    break;
                }
            }
            if (shotRemoved)
                continue;

            if (shot.velocity.z < 0) {
                len2 = invaders.size();
                for (int j = 0; j < len2; j++) {
                    Invader invader = invaders.get(j);
                    if (OverlapTester.overlapSpheres(invader.bounds,
                            shot.bounds)
                            && invader.state == Invader.INVADER_ALIVE) {
                        invader.kill();
                        listener.explosion();
```

```
                    score += 10;
                    shots.remove(i);
                    i--;
                    len--;
                    break;
                }
            }
        } else {
            if (OverlapTester.overlapSpheres(shot.bounds, ship.bounds)
                    && ship.state == Ship.SHIP_ALIVE) {
                ship.kill();
                listener.explosion();
                shots.remove(i);
                i--;
                len--;
            }
        }
    }
}
```

Метод `checkShotCollisions()` несколько более сложен. Он проходит в цикле по всем экземплярам класса `Shot` и проверяет их пересечение с блоком щита, захватчиком или кораблем. Блоки щита могут быть задеты как в случае выстрела захватчика, так и в случае выстрела корабля. Захватчик может быть сбит только выстрелом корабля, а корабль — только выстрелом захватчика. Чтобы определить, кто именно стрелял, необходимо лишь посмотреть *z*-компоненту скорости стрелявшего. Если она положительна, то выстрел произвел захватчик, в противном случае — корабль.

```
public boolean isGameOver() {
    return ship.lives == 0;
}
```

Метод `isGameOver()` сообщает сторонним слушателям о том, что корабль потерял все жизни.

```
public void shoot() {
    if (ship.state == Ship.SHIP_EXPLODING)
        return;

    int friendlyShots = 0;
    int len = shots.size();
    for (int i = 0; i < len; i++) {
        if (shots.get(i).velocity.z < 0)
            friendlyShots++;
    }

    if (System.nanoTime() - lastShotTime > 1000000000 || friendlyShots
                                                    == 0) {
        shots.add(new Shot(ship.position.x, ship.position.y,
```

```
                    ship.position.z, -Shot.SHOT_VELOCITY));
            lastShotTime = System.nanoTime();
            listener.shot();
        }
    }
}
```

Наконец, рассмотрим метод `shoot()`. Он будет вызываться извне всякий раз, когда будет нажиматься кнопка выстрела. В разделе описания игровой механики мы говорили, что корабль может стрелять каждую секунду или если на игровом поле нет выстрела корабля. Конечно же, корабль не сможет стрелять, если он взрывается, поэтому сначала выполняем именно эту проверку. Далее проходим в цикле по всем выстрелам и проверяем, является хотя бы один из них выстрелом корабля. Если такового не находится, корабль может мгновенно выстрелить. В противном случае проверяется последний выстрел, произведенный кораблем. Если со времени его появления прошло больше секунды, корабль может выстрелить снова. В этот раз мы устанавливаем значение скорости равным `-Shot.SHOT_VELOCITY`, что заставит выстрел перемещаться вдоль отрицательной части оси *z*, в сторону захватчиков. Как обычно, мы информируем об этом событии слушателя.

И это все классы игрового мира! Сравните их с классами игры «Большой прыгун». Принципы практически одинаковы, и код выглядит похоже.

Droid Invaders — это, конечно, очень простая игра, поэтому мы можем использовать самые простые решения вроде применения в качестве ограничивающих фигур сфер. Это все, что нужно, для многих простых 3D-игр. Перейдем к рассмотрению двух заключительных фрагментов нашей игры, классам `GameScreen` и `WorldRenderer`!

Класс GameScreen

Как только игра переходит к классу `GameScreen`, игрок может мгновенно начать играть, при этом не используется никакая проверка его готовности. Единственные состояния, о которых надо заботиться, следующие.

- Запущенная игра — в этом случае необходимо отрисовывать фоновое изображение, игровой мир и элементы пользовательского интерфейса так, как это показано на рис. 12.4.
- Приостановленная игра — в данном случае нужно отрисовывать фоновое изображение, мир и меню паузы, опять же как это показано на рис. 12.4.
- Законченная игра — в этом случае отрисовывается практически то же самое.

Мы опять используем прием, который уже применяли в игре «Большой прыгун», где у нас были методы `update()` и `present()` для каждого из трех состояний.

Единственная интересная часть этого класса — обработка пользовательского ввода.

Мы хотим, чтобы игрок мог управлять кораблем с помощью кнопок, расположенных на экране, а также с помощью акселерометра. Мы можем считать поле `Settings.touchEnabled`, чтобы определить, как именно хочет это делать пользователь. В зависимости от выбранного метода ввода информации мы отрисовываем на экране кнопки, также, возможно, появляется необходимость передавать подходящие значения акселерометра методу `World.update()`.

Если отображаются кнопки, нам, конечно же, не нужно использовать показания акселерометра, вместо этого в метод `World.update()` будет передаваться постоянное искусственное значение ускорения. Оно должно находиться в промежутке от –10 (влево) до 10 (вправо). После недолгих экспериментов я остановился на значении –5 для движения влево и 5 для движения вправо с помощью экранных кнопок.

Последний интересный фрагмент класса — это способ объединения отрисовки трехмерного игрового мира и двухмерных элементов пользовательского интерфейса. Взглянем на код класса `GameScreen`, приведенный в листинге 12.11.

Листинг 12.11. Класс GameScreen.java, экран игры

```
package com.badlogic.androidgames.droidinvaders;

import java.util.List;

import javax.microedition.khronos.opengles.GL10;

import com.badlogic.androidgames.droidinvaders.World.WorldListener;
import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.gl.Camera2D;
import com.badlogic.androidgames.framework.gl.FPSCounter;
import com.badlogic.androidgames.framework.gl.SpriteBatcher;
import com.badlogic.androidgames.framework.impl.GLScreen;
import com.badlogic.androidgames.framework.math.OverlapTester;
import com.badlogic.androidgames.framework.math.Rectangle;
import com.badlogic.androidgames.framework.math.Vector2;

public class GameScreen extends GLScreen {
    static final int GAME_RUNNING = 0;
    static final int GAME_PAUSED = 1;
    static final int GAME_OVER = 2;
```

Как обычно, у нас есть несколько констант, кодирующих текущее состояние экрана.

```
    int state;
    Camera2D guiCam;
    Vector2 touchPoint;
    SpriteBatcher batcher;
    World world;
    WorldListener worldListener;
    WorldRenderer renderer;
    Rectangle pauseBounds;
    Rectangle resumeBounds;
```

```
Rectangle quitBounds;
Rectangle leftBounds;
Rectangle rightBounds;
Rectangle shotBounds;
int lastScore;
int lastLives;
int lastWaves;
String scoreString;
FPSCounter fpsCounter;
```

Члены класса GameScreen также относятся к логике игры. У нас есть член, необходимый для отслеживания текущего состояния, камера, вектор для хранения точки прикосновения, экземпляр класса SpriteBatcher, необходимый для отрисовки двухмерных элементов пользовательского интерфейса, экземпляры классов World, WorldListener и WorldRenderer (которые мы очень скоро напишем), а также несколько экземпляров класса Rectangle, необходимых для проверки того, происходило ли прикосновение к элементам пользовательского интерфейса. В дополнение три переменных типа int отслеживают оставшееся количество жизней, волны захватчиков и счет, поэтому нам не нужно каждый раз обновлять параметр scoreString, а это позволяет снизить активность сборщика мусора. Наконец, есть параметр FPSCounter, необходимый для того, чтобы определять производительность игры.

```
public GameScreen(Game game) {
    super(game);

    state = GAME_RUNNING;
    guiCam = new Camera2D(glGraphics, 480, 320);
    touchPoint = new Vector2();
    batcher = new SpriteBatcher(glGraphics, 100);
    world = new World();
    worldListener = new WorldListener() {
        @Override
        public void shot() {
            Assets.playSound(Assets.shotSound);
        }

        @Override
        public void explosion() {
            Assets.playSound(Assets.explosionSound);
        }
    };
    world.setWorldListener(worldListener);
    renderer = new WorldRenderer(glGraphics);
    pauseBounds = new Rectangle(480 - 64, 320 - 64, 64, 64);
    resumeBounds = new Rectangle(240 - 80, 160, 160, 32);
    quitBounds = new Rectangle(240 - 80, 160 - 32, 160, 32);
    shotBounds = new Rectangle(480 - 64, 0, 64, 64);
    leftBounds = new Rectangle(0, 0, 64, 64);
    rightBounds = new Rectangle(64, 0, 64, 64);
    lastScore = 0;
```

```
        lastLives = world.ship.lives;
        lastWaves = world.waves;
        scoreString = "lives:" + lastLives + " waves:" + lastWaves +
                      " score:" + lastScore;
        fpsCounter = new FPSCounter();
    }
```

В конструкторе просто устанавливаем значения всех членов класса. Экземпляр класса `WorldListener` отвечает за воспроизведение корректного звука в случае появления события в игровом мире. Остальная часть конструктора идентична аналогичному методу игры «Большой прыгун», она лишь несколько адаптирована для различающихся местами элементов пользовательского интерфейса.

```
    @Override
    public void update(float deltaTime) {
        switch (state) {
        case GAME_PAUSED:
            updatePaused();
            break;
        case GAME_RUNNING:
            updateRunning(deltaTime);
            break;
        case GAME_OVER:
            updateGameOver();
            break;
        }
    }
```

Метод `update()` передает управление обновлением одному из трех методов в зависимости от текущего состояния экрана.

```
    private void updatePaused() {
        List<TouchEvent> events = game.getInput().getTouchEvents();
        int len = events.size();
        for (int i = 0; i < len; i++) {
            TouchEvent event = events.get(i);
            if (event.type != TouchEvent.TOUCH_UP)
                continue;

            guiCam.touchToWorld(touchPoint.set(event.x, event.y));
            if (OverlapTester.pointInRectangle(resumeBounds, touchPoint)) {
                Assets.playSound(Assets.clickSound);
                state = GAME_RUNNING;
            }

            if (OverlapTester.pointInRectangle(quitBounds, touchPoint)) {
                Assets.playSound(Assets.clickSound);
                game.setScreen(new MainMenuScreen(game));
            }
        }
    }
```

Метод updatePaused() проходит в цикле по всем доступным событиям прикосновения к экрану и проверяет, было ли осуществлено нажатие одного из двух пунктов меню (Resume (Продолжить) или Quit (Выйти)). В каждом случае воспроизводим звук щелчка. Ничего нового.

```
private void updateRunning(float deltaTime) {
    List<TouchEvent> events = game.getInput().getTouchEvents();
    int len = events.size();
    for (int i = 0; i < len; i++) {
        TouchEvent event = events.get(i);
        if (event.type != TouchEvent.TOUCH_DOWN)
            continue;

        guiCam.touchToWorld(touchPoint.set(event.x, event.y));

        if (OverlapTester.pointInRectangle(pauseBounds, touchPoint)) {
            Assets.playSound(Assets.clickSound);
            state = GAME_PAUSED;
        }
        if (OverlapTester.pointInRectangle(shotBounds, touchPoint)) {
            world.shot();
        }
    }

    world.update(deltaTime, calculateInputAcceleration());
    if (world.ship.lives != lastLives || world.score != lastScore
            || world.waves != lastWaves) {
        lastLives = world.ship.lives;
        lastScore = world.score;
        lastWaves = world.waves;
        scoreString = "lives:" + lastLives + " waves:" + lastWaves
                + " score:" + lastScore;
    }
    if (world.isGameOver()) {
        state = GAME_OVER;
    }
}
```

Метод updateRunning() отвечает за два действия: проверка нажатия кнопки паузы и соответственная реакция, а также обновление мира на основании действий пользователя. Первая часть этого пазла довольно тривиальна, поэтому рассмотрим механизм обновления мира. Как вы можете видеть, мы передаем расчет значения ускорения методу, который называется calculateInputAcceleration(). Как только мир обновился, мы проверяем, изменились ли три состояния (жизни, волны и счет), и соответственно обновляем параметр scoreString. Наконец, проверяем, закончилась ли игра, если это так, то выполняется переход в состояние GameOver.

```
private float calculateInputAcceleration() {
    float accelX = 0;
    if (Settings.touchEnabled) {
```

```
            for (int i = 0; i < 2; i++) {
                if (game.getInput().isTouchDown(i)) {
                    guiCam.touchToWorld(touchPoint.set(game.getInput()
                            .getTouchX(i), game.getInput().getTouchY(i)));
                    if (OverlapTester.pointInRectangle(leftBounds,
                                                       touchPoint)) {
                        accelX = -Ship.SHIP_VELOCITY / 5;
                    }
                    if (OverlapTester.pointInRectangle(rightBounds,
                                                       touchPoint)) {
                        accelX = Ship.SHIP_VELOCITY / 5;
                    }
                }
            }
        } else {
            accelX = game.getInput().getAccelY();
        }
        return accelX;
    }
```

Метод `calculateInputAcceleration()` — это то место, где на самом деле обрабатывается ввод данных пользователем. Если был включен режим управления кораблем с помощью прикосновений, мы проверяем, были ли нажаты кнопки перемещения влево или вправо и соответственно устанавливаем значение ускорения равным от –5 до 5. Если используется акселерометр, мы просто возвращаем его текущее значение по оси *y* (помните, игра работает только в пейзажном режиме).

```
    private void updateGameOver() {
        List<TouchEvent> events = game.getInput().getTouchEvents();
        int len = events.size();
        for (int i = 0; i < len; i++) {
            TouchEvent event = events.get(i);
            if (event.type == TouchEvent.TOUCH_UP) {
                Assets.playSound(Assets.clickSound);
                game.setScreen(new MainMenuScreen(game));
            }
        }
    }
```

Метод `updateGameOver()` также тривиален и просто проверяет, произошло ли прикосновение к экрану. Если да, то просто переходим на экран `MainMenuScreen`.

```
    @Override
    public void present(float deltaTime) {
        GL10 gl = glGraphics.getGL();
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        guiCam.setViewportAndMatrices();

        gl.glEnable(GL10.GL_TEXTURE_2D);
        batcher.beginBatch(Assets.background);
```

```
        batcher.drawSprite(240, 160, 480, 320, Assets.backgroundRegion);
        batcher.endBatch();
        gl.glDisable(GL10.GL_TEXTURE_2D);

        renderer.render(world, deltaTime);

        switch (state) {
        case GAME_RUNNING:
            presentRunning();
            break;
        case GAME_PAUSED:
            presentPaused();
            break;
        case GAME_OVER:
            presentGameOver();
        }

        fpsCounter.logFrame();
    }
```

Метод `present()` довольно прост. Как обычно, начинаем с очистки цветового буфера. Мы также очищаем *z*-буфер, поскольку собираемся отрисовать несколько 3D-объектов, для чего нам понадобится *z*-тестирование. Далее устанавливаем матрицу проекции, благодаря чему сможем отрисовывать двухмерное фоновое изображение так же, как и в классах `MainMenuScreen` или `SettingsScreen`. Как только все будет готово, мы прикажем классу `WorldRenderer` отрисовать игровой мир. Наконец, передаем отрисовку элементов пользовательского интерфейса в зависимости от текущего состояния. Обратите внимание, что метод `WorldRenderer.render()` отвечает за установку всего необходимого для отрисовки трехмерного мира!

```
    private void presentPaused() {
        GL10 gl = glGraphics.getGL();
        guiCam.setViewportAndMatrices();
        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
        gl.glEnable(GL10.GL_TEXTURE_2D);

        batcher.beginBatch(Assets.items);
        Assets.font.drawText(batcher, scoreString, 10, 320-20);
        batcher.drawSprite(240, 160, 160, 64, Assets.pauseRegion);
        batcher.endBatch();

        gl.glDisable(GL10.GL_TEXTURE_2D);
        gl.glDisable(GL10.GL_BLEND);
    }
```

Метод `presentPaused()` просто отрисовывает строку `scoreString` с помощью экземпляра класса `Font`, хранящегося в классе `Assets`, а также меню паузы. Обратите внимание, что в этот момент мы уже отрисовали фоновое изображение, как

и трехмерный игровой мир. Все элементы пользовательского интерфейса будут наложены на трехмерный мир.

```
private void presentRunning() {
    GL10 gl = glGraphics.getGL();
    guiCam.setViewportAndMatrices();
    gl.glEnable(GL10.GL_BLEND);
    gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
    gl.glEnable(GL10.GL_TEXTURE_2D);

    batcher.beginBatch(Assets.items);
    batcher.drawSprite(480- 32, 320 – 32, 64, 64,
                       Assets.pauseButtonRegion);
    Assets.font.drawText(batcher, scoreString, 10, 320-20);
    if(Settings.touchEnabled) {
        batcher.drawSprite(32, 32, 64, 64, Assets.leftRegion);
        batcher.drawSprite(96, 32, 64, 64, Assets.rightRegion);
    }
    batcher.drawSprite(480 – 40, 32, 64, 64, Assets.fireRegion);
    batcher.endBatch();

    gl.glDisable(GL10.GL_TEXTURE_2D);
    gl.glDisable(GL10.GL_BLEND);
}
```

Метод `presentRunning()` также довольно прямолинеен. Сначала мы отрисовываем строку `scoreString`. Если был включен режим ввода данных с помощью прикосновений к экрану, то далее отрисовываем кнопки перемещения влево и вправо. В конце метода отрисовываем кнопку стрельбы и сбрасываем все состояния OpenGL ES, которые мы изменили (текстурирование и смешивание).

```
private void presentGameOver() {
    GL10 gl = glGraphics.getGL();
    guiCam.setViewportAndMatrices();
    gl.glEnable(GL10.GL_BLEND);
    gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
    gl.glEnable(GL10.GL_TEXTURE_2D);

    batcher.beginBatch(Assets.items);
    batcher.drawSprite(240, 160, 128, 64, Assets.gameOverRegion);
    Assets.font.drawText(batcher, scoreString, 10, 320-20);
    batcher.endBatch();

    gl.glDisable(GL10.GL_TEXTURE_2D);
    gl.glDisable(GL10.GL_BLEND);
}
```

Метод `presentGameOver()` — это просто отрисовка строк и элементов пользовательского интерфейса.

```
@Override
public void pause() {
    state = GAME_PAUSED;
}
```

Наконец, рассмотрим метод pause(), который просто переводит игровой экран GameScreen в приостановленное состояние.

```
    @Override
    public void resume() {

    }

    @Override
    public void dispose() {
    }
}
```

Остальная часть класса представляет собой пустые методы. Она необходима для того, чтобы закончить описание интерфейса GLGame.

Теперь перейдем к последнему классу — WorldRenderer.

Класс WorldRenderer

Перечислим все, что нам необходимо отрисовать в 3D:

- корабль — мы сделаем это, используя модель и текстуру корабля и применив освещение;
- захватчики — это выполним, применив модель и текстуру захватчика и опять же задействовав освещение;
- выстрелы на поле боя — сделаем это, основываясь на модели выстрела; в этот раз без текстур, но с освещением;
- щиты — выполним это, основываясь на модели щита, опять же без текстурирования, но с использованием освещения и прозрачности (см. рис. 12.3);
- взрывы на месте модели корабля или захватчика — они, конечно же, не освещаются.

Мы уже знаем, как написать код для первых четырех элементов нашего списка. Что же делать со взрывами?

Оказывается, можно использовать SpriteBatcher. Основываясь на времени состояния взрывающегося корабля или захватчика, мы можем получить текстурный регион из экземпляра класса Animation, хранящего анимацию взрыва (смотрите класс Assets). Класс SpriteBatcher может отрисовывать лишь текстурированные прямоугольники в плоскости *xy*, поэтому нам придется найти способ переместить прямоугольник на произвольную позицию в пространстве (ту, где находится взрывающийся корабль игрока или захватчика). Мы можем с легкостью сделать это, используя метод glTranslatef() на видовой матрице перед отрисовкой прямоугольника с помощью экземпляра класса SpriteBatcher!

Создание начальных условий для отрисовки прочих объектов не составляет труда. У нас есть направленный источник света, исходящий из правого верхнего угла, а также подсветка, немного освещающая все объекты вне зависимости от их ориентации. Камера располагается немного выше за кораблем. Она смотрит в точку, расположенную перед кораблем. Мы будем использовать камеру LookAtCamera.

Чтобы камера следовала за кораблем, нам необходимо просто хранить координату ее текущей позиции по оси *x* и синхронизировать позицию точки, на которую направлена камера в данный момент, с координатой корабля по той же оси.

Чтобы добавить еще одну визуальную изюминку, мы будем поворачивать захватчиков вокруг оси *y*. Мы также будем поворачивать корабль вокруг оси *z*, основываясь на его текущей скорости, благодаря чему он будет наклоняться в ту сторону, в которую перемещается.

Напишем для всего этого код. В листинге 12.12 приведен последний класс игры Droid Invaders.

Листинг 12.12. Класс WorldRenderer.java, занимающийся отрисовкой игрового мира

```
package com.badlogic.androidgames.droidinvaders;

import java.util.List;

import javax.microedition.khronos.opengles.GL10;
import com.badlogic.androidgames.framework.gl.AmbientLight;
import com.badlogic.androidgames.framework.gl.Animation;
import com.badlogic.androidgames.framework.gl.DirectionalLight;
import com.badlogic.androidgames.framework.gl.LookAtCamera;
import com.badlogic.androidgames.framework.gl.SpriteBatcher;
import com.badlogic.androidgames.framework.gl.TextureRegion;
import com.badlogic.androidgames.framework.impl.GLGraphics;
import com.badlogic.androidgames.framework.math.Vector3;

public class WorldRenderer {
    GLGraphics glGraphics;
    LookAtCamera camera;
    AmbientLight ambientLight;
    DirectionalLight directionalLight;
    SpriteBatcher batcher;
    float invaderAngle = 0;
```

Класс WorldRenderer отслеживает состояние экземпляра класса GLGraphics, из которого мы получаем экземпляр класса GL10. В этом классе также хранятся экземпляры классов LookAtCamera, AmbientLight, DirectionLight и SpriteBatcher. Наконец, в нем есть член, отслеживающий текущий угол поворота, используемый для захватчиков.

```
    public WorldRenderer(GLGraphics glGraphics) {
        this.glGraphics = glGraphics;
        camera = new LookAtCamera(67, glGraphics.getWidth()
                / (float) glGraphics.getHeight(), 0.1f, 100);
        camera.getPosition().set(0, 6, 2);
        camera.getLookAt().set(0, 0, -4);
        ambientLight = new AmbientLight();
        ambientLight.setColor(0.2f, 0.2f, 0.2f, 1.0f);
        directionalLight = new DirectionalLight();
        directionalLight.setDirection(-1, -0.5f, 0);
        batcher = new SpriteBatcher(glGraphics, 10);
    }
```

В конструкторе мы, как обычно, устанавливаем значения членов класса. Камера имеет поле видимости, равное 67°, расстояние до ближней плоскости равно 0,1 единицы, а расстояние до дальней плоскости равно 100 единицам. Получившееся окно просмотра с легкостью вместит весь игровой мир. Мы поместим его позади корабля чуть выше него и повернем так, чтобы оно смотрело в точку (0; 0; –4). Подсветка имеет светло-серый цвет, а направленный источник света белый и исходит из правой верхней грани. Наконец, создаем экземпляр класса `SpriteBatcher`, который поможет нам отрисовывать прямоугольники взрывов.

```
public void render(World world, float deltaTime) {
    GL10 gl = glGraphics.getGL();
    camera.getPosition().x = world.ship.position.x;
    camera.getLookAt().x = world.ship.position.x;
    camera.setMatrices(gl);

    gl.glEnable(GL10.GL_DEPTH_TEST);
    gl.glEnable(GL10.GL_TEXTURE_2D);
    gl.glEnable(GL10.GL_LIGHTING);
    gl.glEnable(GL10.GL_COLOR_MATERIAL);
    ambientLight.enable(gl);
    directionalLight.enable(gl, GL10.GL_LIGHT0);
    renderShip(gl, world.ship);
    renderInvaders(gl, world.invaders, deltaTime);

    gl.glDisable(GL10.GL_TEXTURE_2D);

    renderShields(gl, world.shields);
    renderShots(gl, world.shots);

    gl.glDisable(GL10.GL_COLOR_MATERIAL);
    gl.glDisable(GL10.GL_LIGHTING);
    gl.glDisable(GL10.GL_DEPTH_TEST);
}
```

Метод `render()` начинается с установки координаты камеры по оси *x*, равной координате корабля по той же оси. Конечно, мы соответственно устанавливаем значение координаты по оси *x* точки, на которую смотрит камера. Таким образом, камера будет следовать за кораблем. Как только позиция и точка просмотра обновятся, мы можем установить проекционную и видовую матрицы с помощью вызова метода `LookAtCamera.setMatrices()`. Далее устанавливаем все состояния, необходимые для отрисовки. Нам понадобятся тестирование глубины, текстурирование, освещение, а также функциональность цветов материалов, поэтому нам не нужно будет определять материалы объектов с помощью вызова метода `glMaterial()`. Следующие два метода активируют подсветку и направленный источник света. После выполнения всех этих действий мы готовы отрисовывать наши объекты.

Первый объект, который мы отрисовываем, — корабль. Его отрисовка выполняется с помощью вызова метода `renderShip()`. Далее отрисовываем захватчиков с помощью метода `renderInvaders()`. Поскольку для отрисовки блоков щита и выстрелов не нужно текстурирование, мы отключаем его для того, чтобы сохранить

несколько тактов процессора. Как только это будет выполнено, отрисовываем выстрелы и щиты с помощью вызовов методов `renderShots()` и `renderShields()`.

Наконец, просто отключаем остальные состояния OpenGL ES.

```
private void renderShip(GL10 gl, Ship ship) {
    if (ship.state == Ship.SHIP_EXPLODING) {
        gl.glDisable(GL10.GL_LIGHTING);
        renderExplosion(gl, ship.position, ship.stateTime);
        gl.glEnable(GL10.GL_LIGHTING);
    } else {
        Assets.shipTexture.bind();
        Assets.shipModel.bind();
        gl.glPushMatrix();
        gl.glTranslatef(ship.position.x, ship.position.y, ship.position.z);
        gl.glRotatef(ship.velocity.x / Ship.SHIP_VELOCITY * 90, 0, 0, -1);
        Assets.shipModel.draw(GL10.GL_TRIANGLES, 0,
                Assets.shipModel.getNumVertices());
        gl.glPopMatrix();
        Assets.shipModel.unbind();
    }
}
```

Метод `renderShip()` начинается с проверки состояния корабля. Если он взрывается, то мы отключаем освещение, вызываем метод `renderExplosion()` для отрисовки взрыва на месте корабля и снова включаем освещение. Если корабль цел, мы связываем его текстуру и модель, помещаем в стек видовую матрицу, передвигаем его на свою позицию, поворачиваем его вокруг оси *z*, в зависимости от скорости и рисуем его модель. Наконец, из стека выталкивается видовая матрица (оставляя в нем только вид камеры) и отвязываются вершины модели корабля.

```
private void renderInvaders(GL10 gl, List<Invader> invaders,
                            float deltaTime) {
    invaderAngle += 45 * deltaTime;

    Assets.invaderTexture.bind();
    Assets.invaderModel.bind();
    int len = invaders.size();
    for (int i = 0; i < len; i++) {
        Invader invader = invaders.get(i);
        if (invader.state == Invader.INVADER_DEAD) {
            gl.glDisable(GL10.GL_LIGHTING);
            Assets.invaderModel.unbind();
            renderExplosion(gl, invader.position, invader.stateTime);
            Assets.invaderTexture.bind();
            Assets.invaderModel.bind();
            gl.glEnable(GL10.GL_LIGHTING);
        } else {
            gl.glPushMatrix();
            gl.glTranslatef(invader.position.x, invader.position.y,
                    invader.position.z);
            gl.glRotatef(invaderAngle, 0, 1, 0);
```

```
                Assets.invaderModel.draw(GL10.GL_TRIANGLES, 0,
                        Assets.invaderModel.getNumVertices());
                gl.glPopMatrix();
            }
        }
        Assets.invaderModel.unbind();
    }
```

Метод `renderInvaders()` очень похож на метод `renderShip()`. Единственное различие заключается в том, что мы в цикле проходим по списку захватчиков, а текстура привязывается к сети до этого. Это уменьшит количество привязок и несколько ускорит отрисовку. Далее для каждого захватчика проверяется его состояние и отрисовывается либо взрыв, либо нормальная модель захватчика. Поскольку привязка модели и текстуры выполняется вне цикла, нам приходится отвязывать их, а затем привязывать снова перед тем, как отрисовать взрыв вместо захватчика.

```
    private void renderShields(GL10 gl, List<Shield> shields) {
        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
        gl.glColor4f(0, 0, 1, 0.4f);
        Assets.shieldModel.bind();
        int len = shields.size();
        for (int i = 0; i < len; i++) {
            Shield shield = shields.get(i);
            gl.glPushMatrix();
            gl.glTranslatef(shield.position.x, shield.position.y,
                    shield.position.z);
            Assets.shieldModel.draw(GL10.GL_TRIANGLES, 0,
                    Assets.shieldModel.getNumVertices());
            gl.glPopMatrix();
        }
        Assets.shieldModel.unbind();
        gl.glColor4f(1, 1, 1, 1f);
        gl.glDisable(GL10.GL_BLEND);
    }
```

Метод `renderShields()` отрисовывает блоки щита. Мы применяем тот же принцип, что и при отрисовке захватчиков. Привязываем модель только однажды. Поскольку у блоков щита нет текстуры, нам не нужно ее привязывать. Однако нам следует включить смешивание. Нам также необходимо сделать общий цвет вершин синим, а его альфа-компоненту установить равной 0,4. Это сделает блоки щита несколько прозрачными.

```
    private void renderShots(GL10 gl, List<Shot> shots) {
        gl.glColor4f(1, 1, 0, 1);
        Assets.shotModel.bind();
        int len = shots.size();
        for (int i = 0; i < len; i++) {
            Shot shot = shots.get(i);
            gl.glPushMatrix();
```

```
            gl.glTranslatef(shot.position.x, shot.position.y,
                            shot.position.z);
            Assets.shotModel.draw(GL10.GL_TRIANGLES, 0,
                    Assets.shotModel.getNumVertices());
            gl.glPopMatrix();
        }
        Assets.shotModel.unbind();
        gl.glColor4f(1, 1, 1, 1);
    }
```

Отрисовка выстрелов с помощью метода `renderShots()` очень похожа на отрисовку щитов, за исключением того, что не используется смешивание и изменяется цвет вершин (желтый).

```
    private void renderExplosion(GL10 gl, Vector3 position, float stateTime) {
        TextureRegion frame = Assets.explosionAnim.getKeyFrame(stateTime,
                Animation.ANIMATION_NONLOOPING);

        gl.glEnable(GL10.GL_BLEND);
        gl.glPushMatrix();
        gl.glTranslatef(position.x, position.y, position.z);
        batcher.beginBatch(Assets.explosionTexture);
        batcher.drawSprite(0, 0, 2, 2, frame);
        batcher.endBatch();
        gl.glPopMatrix();
        gl.glDisable(GL10.GL_BLEND);
    }
}
```

Наконец, у нас есть таинственный метод `renderExplosion()`. Мы получаем позицию, на которой хотим отрисовать взрыв, а также время состояния взрывающегося объекта. Второй параметр используется для получения корректного текстурного региона из экземпляра класса `Animation`, так же мы поступали и в случае с Бобом в игре «Большой прыгун».

Первое, что нам нужно сделать, — получить кадр анимации взрыва, основываясь на времени состояния. Далее включаем смешивание, поскольку взрыв имеет прозрачные пикселы, которые мы не хотим отрисовывать. Помещаем текущую видовую матрицу в стек и вызываем метод `glTranslatef()`, поэтому все, что мы отрисуем после вызова этого метода, будет размещено в заданной позиции. Далее указываем экземпляру класса `SpriteBatcher`, что собираемся отрисовать прямоугольник с использованием текстуры взрыва.

Вызов следующего метода как раз и является главным. Мы указываем экземпляру класса `SpriteBatcher` отрисовать прямоугольник в точке (0; 0; 0) (координата по оси *z* не задана, но неявно равна нулю, как вы помните), ширина и высота этого прямоугольника равны 2 единицы. Поскольку мы использовали метод `glTranslatef()`, этот прямоугольник будет находиться не в центре координат, а в центре позиции, определенной методом `glTranslatef()`, которая в точности повторяет позицию взрывающегося корабля или захватчика. Наконец, видовая матрица выталкивается из стека — и мы снова отключаем смешивание.

Вот и все. Двенадцать классов, создающих полноценную трехмерную игру, пародирующую классическую игру Space Invaders.

Поиграйте в нее. Когда вы закончите, мы рассмотрим вопросы, связанные с производительностью.

Оптимизация

Перед тем как подумаем об оптимизации нашей игры, улучшим ее производительность.

Поместим экземпляр класса `FPSCounter` в класс `GameScreen` и посмотрим на его выводимые сообщения на устройствах Hero, Droid и Nexus One.

```
Hero (Android 1.5):
02-17 00:59:04.180: DEBUG/FPSCounter(457): fps: 25
02-17 00:59:05.220: DEBUG/FPSCounter(457): fps: 26
02-17 00:59:06.260: DEBUG/FPSCounter(457): fps: 26
02-17 00:59:07.280: DEBUG/FPSCounter(457): fps: 26

Nexus One (Android 2.2.1):
02-17 01:05:40.679: DEBUG/FPSCounter(577): fps: 41
02-17 01:05:41.699: DEBUG/FPSCounter(577): fps: 41
02-17 01:05:42.729: DEBUG/FPSCounter(577): fps: 41
02-17 01:05:43.729: DEBUG/FPSCounter(577): fps: 40

Droid (Android 2.1.1):
02-17 01:47:44.096: DEBUG/FPSCounter(1758): fps: 47
02-17 01:47:45.112: DEBUG/FPSCounter(1758): fps: 47
02-17 01:47:46.127: DEBUG/FPSCounter(1758): fps: 47
02-17 01:47:47.135: DEBUG/FPSCounter(1758): fps: 46
```

Hero немного не справляется, но при 25 кадрах в секунду по-прежнему можно насладиться игрой. Устройство Nexus One выдает 47 кадров в секунду, Droid от него не отстает, что положительно сказывается на игре. Что мы можем улучшить?

Если рассматривать производительность с точки зрения изменения состояний, то все не настолько плохо. Мы можем уменьшить количество избыточных изменений, например вызовы методов `glEnable()/glDisable()`. Но по опыту предыдущих оптимизаций мы уже знаем, что это не даст нам особых преимуществ.

Чтобы улучшить производительность на Hero, есть только один способ — отключить освещение. Как только мы удаляем вызовы соответствующих методов `glEnable()/glDisable()`, находящихся в `WorldRenderer.render()`, `WorldRenderer.renderShip()` и `WorldRenderer.renderInvaders()`, Hero начинает показывать следующие результаты:

```
Hero (Android 1.5):
02-17 01:14:44.580: DEBUG/FPSCounter(618): fps: 31
02-17 01:14:45.600: DEBUG/FPSCounter(618): fps: 31
02-17 01:14:46.610: DEBUG/FPSCounter(618): fps: 31
02-17 01:14:47.630: DEBUG/FPSCounter(618): fps: 31
```

Довольно неплохое улучшение, особенно учитывая то, что мы всего лишь отключили освещение. Возможно создание специального кода, отрисовывающего объекты на различных устройствах, но мы постараемся этого избежать. Есть ли еще что-нибудь, что мы можем сделать?

Способ, которым мы отрисовываем взрывы, несколько неоптимален в том случае, если взрывается захватчик. Мы меняем модель и привязки текстур в середине цикла отрисовки захватчиков, это не обрадует графический конвейер. Однако взрывы происходят не так часто и не занимают много времени (1,6 секунды). Измерения проводились в ситуациях, когда на экране не было взрывов, поэтому они неидеальны.

Правда заключается в том, что мы пытаемся отрисовать слишком много объектов за один кадр, что вызывает задержку работы графического конвейера. С нашими текущими знаниями OpenGL ES мы ничего не сможем с этим поделать. Однако основываясь на том, что игра работает довольно хорошо на всех устройствах, мы не обязаны добиваться производительности 60 кадров в секунду. Устройства Droid и Nexus One известны своим фиксированным временем отрисовки даже не очень сложных 3D-сцен при 60 кадрах в секунду. Поэтому последний урок, который мы извлечем из этого: не сходите с ума, если ваша игра не выдает 60 кадров в секунду. Если она неплохо смотрится и играбельна, то вы можете смириться и с 30 кадрами в секунду.

ПРИМЕЧАНИЕ

Типичные стратегии оптимизации включают использование отсечения, объектов-буферов вершин и прочих продвинутых приемов, которые мы не рассматривали. Я попробовал добавить их в игру Droid Invaders. Результат нулевой. Ни на одном из устройств не повысилась производительность. Это не означает, что эти приемы бесполезны. Это зависит от множества факторов, и сложно предугадать, как поведет себя та или иная конфигурация. Если вы заинтересовались, просто прочитайте об этих приемах в Интернете самостоятельно.

Подводя итог

В этой главе мы закончили нашу третью игру, полноправный трехмерный клон игры Space Invaders. Мы использовали небольшие приемы и техники, изученные на протяжении всей книги, и финальный результат был довольно удовлетворительным. Конечно, эта игра не самого высшего сорта. Фактически ни одна из наших игр не затягивает надолго. Здесь в действие вступаете вы. Будьте креативным, расширьте эти игры и сделайте их интересными! У вас под рукой множество инструментов.

13 Публикуем вашу игру

Последним шагом в становлении разработчиком игр под Android является передача вашей игры другим людям. Есть два возможных варианта:

- взять APK-файл из каталога `bin` вашего проекта, поместить его в Интернет и рассказать об этом друзьям, чтобы они скачали его и установили на своих устройствах;
- опубликовать приложение на Android Market, как поступают настоящие профессионалы.

Первый вариант — это отличный способ протестировать игру перед выходом на рынок. Все, что вам нужно, — чтобы другие люди получили APK-файл и установили его на своих устройствах. Настоящее веселье начинается тогда, когда ваша игра готова выйти на рынок.

Несколько слов о тестировании

Как мы видели в предыдущих главах, многие устройства отличаются друг от друга по различным параметрам. До того как опубликовать приложение, убедитесь, что оно работает как следует на нескольких распространенных устройствах и версиях ОС Android. К сожалению, сейчас это сделать непросто. Мне повезло заполучить несколько телефонов, представляющих разные классы и поколения. Возможно, воспользоваться таким вариантом не позволит ваш бюджет. Можете положиться на эмулятор (но не сильно рассчитывайте на него, поскольку он довольно ненадежен) или, что лучше, воспользуйтесь помощью друзей.

Еще один способ протестировать приложение — поместить его бета-версию на Android Market. Вы можете отметить приложение как бета-версию в заголовке, чтобы пользователи знали, чего ожидать.

Некоторые пользователи с удовольствием проигнорируют все предупреждения и будут жаловаться на незаконченность вашего приложения. Такова жизнь, вам придется научиться справляться с негативными и, возможно, несправедливыми комментариями. Помните, пользователи — это главное. Не злитесь на них и попробуйте понять, как можно улучшить приложение.

Вот те устройства, которые я использую для тестирования приложений:

- Samsung Galaxy Leo/I5801, экран 320 × 240 пикселов;
- HTC Hero с Android 1.5, экран 480 × 320 пикселов;
- Motorola Milestone/Droid с Android 2.1, экран 854 × 480 пикселов;
- HTC Desire HD с Android 2.2, экран 800 × 480 пикселов;
- Nexus One с Android 2.3, экран 800 × 480 пикселов.

Как вы можете видеть, я применяю большой диапазон размеров, разрешений и поколений устройств. Если вы ищете сторонних тестировщиков, убедитесь, что вы охватите большинство поколений устройств, обозначенных здесь. Более новые устройства также должны быть в вашем списке, но вы будете тестировать не производительность, а совместимость.

Еще одним классом устройств являются планшеты. В момент написания книги Samsung Galaxy Tab был практически единственным подобным устройством на рынке, а Android Xoom только анонсировался. На планшетах, само собой, большее разрешение экрана. Приемы, которые мы рассмотрели, должны хорошо масштабироваться. Если вы хотите сделать игру более красивой, то вы можете принять в расчет пиксельную плотность экрана и физические размеры устройства, что было показано в главе 5.

Наконец, вам придется смириться с тем фактом, что вы не сможете протестировать свое приложение сразу на всех устройствах. Скорее всего, вы будете получать сообщения об ошибках, которым вы не можете найти объяснение и которые могут происходить из-за того, что пользователь использует нестандартный ROM, поведение которого вы не рассматривали. В любом случае не стоит паниковать. Однако, если подобные сообщения об ошибках начинают приходить слишком часто, вам следует придумать, как же можно справиться с этой проблемой. К счастью, Android Market поможет вам в этом. Рассмотрим поближе принцип его работы.

ПРИМЕЧАНИЕ

Помимо системы оповещения об ошибках, встроенной в Android Market, есть еще одно замечательное решение, которое называется ACRA. Оно является библиотекой с открытым кодом, разработанной специально для создания сообщений об ошибке при падении приложения на ОС Android. Эту библиотеку можно скачать по адресу http://code.google.com/p/acra/. Она очень проста в использовании. Просто следуйте руководству, размещенному на странице Google Code, это поможет вам интегрировать ее в свое приложение.

Становимся зарегистрированным разработчиком

Android позволяет очень легко опубликовать приложение на официальном Android Market.

Все, что вам нужно сделать, — зарегистрироваться как разработчик, единоразово заплатив $25. В зависимости от страны, в которой вы живете, этот аккаунт по-

зволит вам размещать бесплатные и/или платные приложения (см. главу 1, в которой содержится список стран, откуда вы можете продавать свое приложение). Google усиленно работает над тем, чтобы расширить количество стран, из которых вы сможете продавать свое приложение. Чтобы зарегистрировать аккаунт, посетите сайт https://market.android.com/publish/signup[1] и следуйте инструкциям.

В дополнение к аккаунту разработчика вам также понадобится зарегистрировать торговый аккаунт в Google Checkout, если вы хотите продавать приложения. Вам предложат сделать это во время процесса регистрации. Я не адвокат, поэтому не могу давать советы в этой области. Убедитесь, что вы понимаете законы, которые управляют продажей приложений до того, как начнете их распространять. Если вы сомневаетесь, то лучше проконсультироваться с экспертом. Я не хочу вас запугать, поскольку процесс довольно прост. Однако министерство по налогам и сборам вашей страны может заинтересоваться тем, что вы делаете.

Google возьмет 30 % ваших заработанных денег за распространение приложения и предоставление инфраструктуры. Это стандартный процент, взимаемый большинством магазинов приложений на различных платформах.

Подписываем APK-файл вашей игры

После того как вы успешно зарегистрировались как официальный разработчик на Android, пришло время подготовить ваше приложение к публикации. Чтобы опубликовать приложение, вы должны подписать APK-файл. Перед тем как сделать это, убедитесь, что все находится на своих местах. Есть определенные действия, которые нужно произвести до того, как вы подпишете приложение.

- Удалите атрибут `android:debuggable` из тега `<application>` вашего файла манифеста.
- В тэге `<manifest>` вы найдете атрибуты `android:versionCode` и `android:versionName`. Если вы ранее уже публиковали более старую версию приложения, вам следует увеличить значение атрибута `versionCode` и изменить атрибут `versionName`. Атрибут `versionCode` должен являться числом, поле `versionName` вы заполняете так, как хотите.
- Если целевая платформа больше или равна SDK уровня 8 (Android 2.2), вам также следует убедиться, что в теге `<manifest>` атрибут `android:installLocation` имеет значение `preferExternal` или `auto`. Это действие гарантирует, что приложение хранится по возможности во внешнем хранилище.
- Убедитесь, что вы запрашиваете доступ только к необходимым функциям устройства. Пользователям не нравятся приложения, запрашивающие слишком много прав доступа. Проверьте теги `<uses-permission>` вашего файла манифеста.

[1] На момент подготовки русскоязычного издания этот адрес имел следующий вид: https://play.google.com/apps/publish/signup. — *Примеч. ред.*

- Дважды проверьте правильность атрибутов `android:minSdkVersion` и `android:targetSdkVersion`. Ваше приложение будет видно в Android Market только для тех телефонов, на которых версия ОС Android равна или выше той, что определена в SDK.

Затем дважды проверьте все эти пункты. Как только вы это сделаете, вы, наконец, сможете экспортировать подписанный APK-файл на рынок.

1. Для этого вам следует щелкнуть правой кнопкой мыши на проекте в обозревателе пакетов и выбрать пункт меню Android Tools ▸ Export Signed Application Package (Инструменты Android ▸ Экспортировать подписанный пакет приложения). Появится окно, показанное на рис. 13.1.

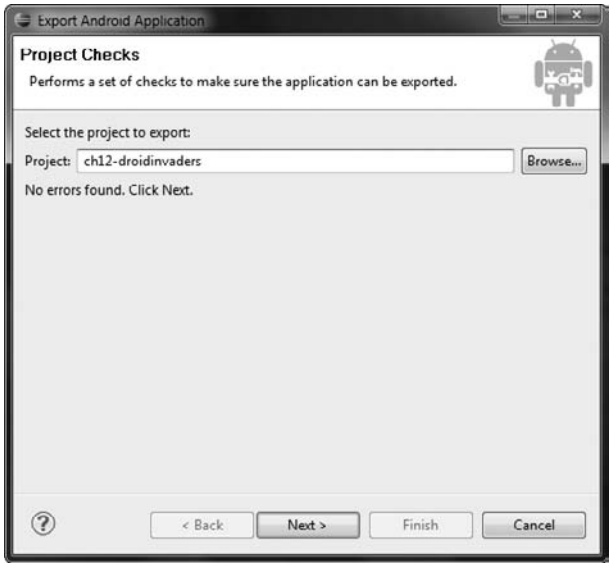

Рис. 13.1. Окно экспорта подписанного APK-файла

2. Нажмите кнопку Next (Далее), чтобы появилось окно, изображенное на рис. 13.2.
3. Хранилище ключей — это файл, защищенный паролем и хранящий ключи, которыми вы подписываете ваши APK-файлы. Поскольку вы еще пока не создали ни одного, вы можете сделать это прямо сейчас с помощью данного диалога. Просто укажите расположение хранилища ключей и пароль, который вы используете для его защиты. Если у вас уже есть хранилище ключей (например, вы публикуете вторую версию вашего приложения), вы можете установить переключатель в положение Use existing keystore (Использовать существующее хранилище ключей) и просто указать, где располагается его файл. Нажмите кнопку Next (Далее), чтобы перейти к окну, приведенному на рис. 13.3.
4. Для создания подходящего ключа вам следует указать логин, пароль и срок действия подписи в годах, а также ваши имя и фамилию. Остальное заполнять

Рис. 13.2. Окно выбора или создания хранилища ключей

Рис. 13.3. Окно создания ключа для подписи APK-файла

необязательно, но я рекомендую вам сделать это. Еще один щелчок на кнопке Next (Далее) — и перед вами появляется последнее окно (рис. 13.4).

5. Мы уже почти закончили. Просто определите, где должен находиться экспортированный APK-файл, и запомните путь. Он понадобится в дальнейшем при загрузке APK-файла на Android Market.

Рис. 13.4. Определяем место расположения файла

Когда вы хотите опубликовать новую версию изданного ранее приложения, вы можете просто повторно использовать ключ, который создали ранее. В окне, показанном на рис. 13.2, просто выберите файл хранилища ключей, созданный ранее, и предоставьте его пароль. Далее вы увидите окно, изображенное на рис. 13.5.

Рис. 13.5. Повторное использование ключа

Просто выберите ранее созданный ключ, введите его пароль и продолжайте далее, как описано выше. В обоих случаях вы получите подписанный APK-файл, который готов к загрузке на Android Market.

ПРИМЕЧАНИЕ

Как только вы загрузите подписанный APK-файл, вам придется использовать тот же ключ для подписывания всех последующих версий приложения.

Размещение игры на Android Market

Пришло время зайти в ваш аккаунт на сайте Android Market. Просто перейдите по ссылке http://market.android.com/publish и войдите в него. Вас поприветствует интерфейс, показанный на рис. 13.6.

Рис. 13.6. Добро пожаловать на Android Market, разработчик!

Вы видите то, что в Android называется консолью разработчика. О ней мы поговорим через минуту. Сейчас сконцентрируемся на публикации приложения. Это позволит сделать нам кнопка Upload Application (Загрузить приложение). Пройдем по всем этапам загрузки игры.

Загрузка ресурсов

Первое, что вам нужно сделать, — указать APK-файл, который вы только что подписали и экспортировали. Просто выберите его и нажмите кнопку Upload (Загрузить). Вы можете продолжить загружать остальные ресурсы, пока APK-файл будет загружаться в фоновом режиме. Как только файл загрузится, он будет проверен системой, и если появятся ошибки, вы будете проинформированы.

Вы также должны предоставить как минимум два скриншота вашего приложения. Они должны быть определенного формата (JPEG или PNG) и размера (320×480, 480×800 или 480×854). Они будут показываться пользователю в тот момент, когда он просматривает детали вашего приложения на Android Market (на устройстве и на официальной веб-странице http://market.android.com).

Далее вам следует загрузить значок вашего приложения в высоком разрешении размером 512×512 формата PNG или JPEG. В настоящее время он используется только веб-сайтом Android Market, когда пользователь просматривает ваше приложение там. Сделайте его красивым.

Промокартинка (180×120, PNG или JPEG) и картинка для главной страницы (1024×500) показываются на Android Market в том случае, если ваша игра присутствует на главной странице. Для игры быть на главной странице в разделе особенных приложений очень почетно, поскольку это означает, что ваше приложение пользователи увидят первым, когда откроют Android Market на своем устройстве или веб-сайте. Только Google знает, какие приложения становятся особенными.

Наконец, вы можете предоставить ссылку на видеофайл с YouTube, рассказывающий о вашем приложении. Он будет показываться на веб-сайте Android Market.

Описание деталей

В разделе Listing Details (Описание деталей) вы можете определить название (максимум 30 символов) и описание вашего приложения (максимум 4000 символов), возможно на нескольких языках. Они будут показываться пользователю на Android Market.

Дополнительные 500 символов предоставляются для того, чтобы уведомить пользователей о последних изменениях самой новой версии приложения. Промотекст будет использоваться, если ваше приложение станет особенным.

Далее вам следует определить тип и категорию вашего приложения. Для каждого типа приложений есть целый список категорий. Для игр вы можете выбрать категорию Arcade &Action (Аркады и экшн), Brain & Puzzle (Головоломки), Cards & Casino (Азартные игры), Casual (Виджеты), Racing (Гонки) и Sports Games (Спортивные игры). Эти категории несколько странные (гонки это не спорт?), видимо, они не были хорошо продуманы. Давайте надеяться на изменения в будущем.

Наконец, вам следует решить, будут ли пользователи платить за вашу игру. Это решение окончательное. Как только вы определитесь, вы не сможете изменить его,

если только не издадите игру снова с другим ключом. Вы потеряете все комментарии пользователей и слегка разозлите потенциальных игроков. Просто подумайте, чего вы хотите от игры. Я не буду давать вам советы о том, как оценить свою игру. Цена $0,99 является стандартной для большинства игр, и пользователи почему-то ожидают именно ее. Однако вас никто не удерживает от экспериментирования. Если вы будете продавать вашу игру, убедитесь, что все по закону.

Настройки публикации

Панель Publishing Options (Настройки публикации) позволяет вам определить, хотите ли вы защитить от копирования ваше приложение, а также его возрастной рейтинг и места, где вы хотите опубликовать вашу игру. Защита от копирования практически бесполезна, поскольку ее с легкостью можно обойти, просто следуя нескольким руководствам, размещенным в Интернете. Google собирается прекратить использовать этот тип защиты от копирования, что запретит пользователям копировать APK-файл с устройства и распространять его бесплатно. Вместо этого Google теперь предоставляет API, позволяющий интегрировать службу лицензий в ваше приложение. Эта служба должна усложнить пиратское распространение игры. Рассмотрение этой темы выходит за пределы этой книги, я предлагаю вам отправиться на сайт разработчиков на ОС Android, если вы слишком беспокоитесь об авторских правах. Как и в случае с другими схемами управления цифровыми правами (Digital Rights Management, DRM), у пользователей иногда возникают проблемы — они не смогут запустить приложение или даже установить его из-за службы лицензии. Следует отнестись к возможным проблемам с недоверием. Это лучший вариант DRM, который у вас есть в данный момент.

Возрастной рейтинг позволяет вам определить целевую аудиторию. Существуют руководства, помогающие оценить приложение, вы их можете найти, нажав ссылку Learn More (Узнать больше) на странице публикации приложения. Ваше приложение будет отфильтровываться рынком, основываясь на заданном возрастном рейтинге. Тщательно определяйте тот, что будет идеально подходить вашей игре.

Наконец, вы выбираете места, в которых будет доступно ваше приложение. Скорее всего, вам захочется, чтобы ваше приложение было доступно везде. Однако бывают и такие ситуации, в которых вы захотите опубликовать приложение только в некоторых странах, возможно, из-за законов. Обычно вам ничто не помешает опубликовать свою игру везде.

Публикуем!

Последнее, что вам потребуется указать на странице публикации игры, — ваша контактная информация, ваше согласие с Android Content Guidelines (на него есть ссылка на той же странице, прочтите его), а также ваше согласие с законами экспорта США, с чем вы обычно согласны всегда. После того как вы предоставили всю

необходимую информацию, пришло время нажать большую кнопку Publish (Опубликовать) в нижней части страницы!

В мгновение ока ваша игра станет доступной миллионам людей по всему миру. На Android Market ваше приложение будет находиться в категории Just In (Новинки), что позволит пользователям легко найти ее, пока она не опустится в списке из-за появления других новых приложений. Не пытайтесь обмануть механизм Just In (Новинки), это не сработает. Загрузка «новой» версии вашего приложения каждые несколько часов не приведет к тому, что оно будет подниматься в списке.

Маркетинг

Поскольку я не эксперт в этой области, приведу лишь несколько своих мыслей по поводу маркетинга. В Интернете вокруг платформы Android существует здоровая экосистема, состоящая из новостных веб-сайтов, блогов, форумов и т. д. Большая часть блогов и новостных сайтов будет рада сообщить о новых играх, поэтому постарайтесь быть на связи с как можно большим их количеством. Существует также несколько сайтов, ориентированных на игры для Android, вроде http://www.droidgamers.com, они должны быть первыми, с кем вы свяжетесь. Без продвижения вашу игру вряд ли заметят, поскольку список Just In (Новинки) заполняется очень быстро. Маркетинг — это часть успеха вашей игры. По этой теме написано много книг, но я полагаю, что волшебной формулы нет. Создайте отличную игру и сообщите людям об этом.

Консоль разработчика

Как только ваша игра попадет на рынок, вы захотите отслеживать ее статус. Сколько людей уже ее загрузили? Были ли падения приложения? Что говорят пользователи? Обо всем этом вы можете узнать из консоли разработчика (см. рис. 13.6).

Для каждого опубликованного вами приложения вы можете узнать следующую информацию:

- общий рейтинг вашей игры и количество оценок;
- комментарии пользователей (просто щелкните на ссылке Comments (Комментарии) соответствующего приложения);
- количество установок приложения;
- количество активных установленных версий вашего приложения;
- сообщения об ошибках.

Сообщения об ошибках интересуют нас больше всего. На рис. 13.7 показаны сообщения об ошибках, которые я получил для своей игры Newton.

Всего произошло 8 зависаний и 2 падения. Newton находится на рынке уже больше года, для него это неплохой показатель. Вы также можете подробно просмотреть отчеты о конкретных ошибках. Система оповещений об ошибках предоставляет детализированную информацию о падениях и зависаниях, такую как модель

устройства, на котором это произошло, полные записи о состоянии стека и т. д. Это вам очень поможет, когда вы будете пытаться понять, что же пошло не так. Комментарии на Android Market не смогут вам сильно помочь.

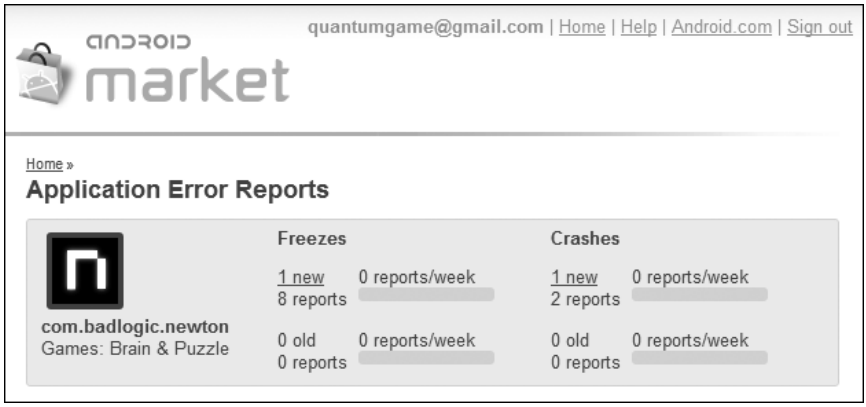

Рис. 13.7. Просмотр сообщений об ошибках

ПРИМЕЧАНИЕ

Отправка сообщений об ошибках — это особенность устройств, которая не поддерживается более старыми версиями ОС Android.

Если вы хотите быть уверенными в том, что отслеживаете ВСЕ проблемы, я предлагаю вам использовать ACRA.

Подводя итог

Публикация вашей игры на Android Market — это очень просто. Самое сложное — проинформировать других людей о том, что вы это сделали. К сожалению, я не могу вам с этим помочь. Теперь у вас есть все необходимые знания о том, как разработать, реализовать и опубликовать вашу первую игру для ОС Android. Да пребудет с вами сила!

14 Что дальше?

Мы рассмотрели большое количество материала в этой книге, но многое все еще остается нерассмотренным. Если предложенный в этой книге материал дался вам легко, вы, возможно, захотите копнуть глубже. Вот несколько идей и направлений для вашего путешествия.

Становимся социальными

Один из самых больших трендов в играх последних лет — это интеграция с социальными сервисами. Twitter, Facebook и Reddit стали важной частью жизни многих людей. Эти люди хотят играть вместе со своими друзьями и семьей. Что может быть круче, чем победить своего отца в Zombie Shooter 13?

Twitter и Facebook предоставляют API, позволяющий вам взаимодействовать с их сервисами. Хотите предоставить пользователю возможность отправить в Twitter его последний рекорд в вашей игре? Нет проблем — просто интегрируйте в игру Twitter API.

В мобильном пространстве также есть два больших сервиса, которые позволяют объединить игроков и дают им возможность легко открывать для себя новые игры: Scoreloop и OpenFeint. Они оба предоставляют API для ОС Android, который позволяет пользователям легко хранить свои результаты онлайн, сравнивать достижения и т. д. Оба API довольно просты и поставляются с хорошими примерами и документацией. Я предпочитаю использовать Scoreloop.

Определение местоположения

Мы лишь вскользь рассмотрели эту тему в главах 1 и 4, я не использую эту функцию ни в одной своей игре. Все устройства с ОС Android поставляются с сенсором, позволяющим определять местоположение пользователя. Это довольно интересно, но использование этой особенности в игре возможно только для создания каких-либо инновационных и никогда ранее не применяемых игровых механик. Она по-прежнему редко задействуется в большинстве игр для ОС Android. Сможете ли вы придумать интересный способ использовать GPS-сенсор?

Многопользовательская функциональность

Эта книга для начинающих, поэтому мы не рассматривали этот вопрос.

Достаточно сказать, что Android предоставляет API, позволяющие реализовать эту возможность. В зависимости от типа игры, сложность реализации многопользовательской игры варьируется. В пошаговых играх, вроде карт или шахмат, это довольно просто. Динамичные экшены или стратегии реального времени — это совершенно другой вопрос. В обоих случаях, вам необходимо иметь понимание сетевого программирования, этой теме посвящено множество материалов, существующих в Интернете.

OpenGL ES 2.0 и выше

До этого момента вы видели только половину OpenGL ES. Мы использовали исключительно OpenGL ES 1.0, поскольку в настоящее время эта версия широко применяется на различным версиях ОС Android. Его природа фиксированных функций хорошо подходит для 3D-программирования. Однако есть новая, более яркая версия OpenGL ES, которая позволяет вам напрямую программировать GPU. Она значительно отличается от той версии, которую вы видели в этой книге, с ней именно вы будете ответственным за получение даже одного тексела (текстурного пиксела) или преобразование координат вершины вручную, все это происходит непосредственно на GPU.

OpenGL ES 2.0 имеет так называемый основанный на шейдерах, или программируемый, конвейер в противоположность фиксированным функциям OpenGL ES 1.0 и 1.1. Для многих трехмерных и (двухмерных) игр использования OpenGL ES 1.x более чем достаточно. Хотя, если вы хотите, чтобы игра выглядела более красиво, вы можете воспользоваться и OpenGL ES 2.0! Не бойтесь — все концепции, которые вы изучили в этой книге, с легкостью могут быть перенесены в эту среду.

Мы также не затрагивали такие темы, как анимированные 3D-модели и более продвинутые концепции OpenGL ES 1.x, такие как объекты-буферы вершин. Как и в случае с OpenGL ES 2.0, вы сможете найти множество ресурсов как в Интернете, как и в книжном магазине. Вы уже знаете основы, пришло время узнать больше!

Фреймворки и движки

Если вы приобрели эту книгу, уже имея какие-то знания касательно разработки игр, вы можете задаться вопросом, почему я не выбрал один из доступных фреймворков, предназначенных для разработки игр для ОС Android. Изобретать колесо заново — это плохо, не так ли? Я хотел, чтобы вы четко усвоили принципы. Хотя это иногда может быть неудобно, в итоге это окупится. Благодаря приобретенным здесь знаниям вы сможете с легкостью выбрать любое готовое решение, я надеюсь, что вы сможете определить, какие преимущества оно вам дает.

Для платформы Android существует несколько коммерческих и открытых фреймворков и движков. В чем разница между фреймворком и движком?

Фреймворк позволяет вам контролировать каждый аспект вашей среды разработки игр. Правда, вам самим придется разбираться, как именно решить ту или иную задачу (например, как организовать ваш игровой мир, как обрабатывать экраны и переходы и т. д.). В этой книге мы разработали (очень простой) фреймворк, на котором создали наши игры.

Движок, с другой стороны, ориентирован на специфические задачи. Он диктует вам, как вы должны решать те или иные проблемы, предоставляя вам простые в использовании модули для типичных задач и общую архитектуру вашей игры. Обратная сторона медали — ваша игра может не соответствовать этим готовым решениям. Вам часто придется самостоятельно модифицировать движок для того, чтобы достичь своих целей, что может быть невозможно, если его исходный код вам недоступен. Движки могут значительно сократить время разработки, но могут и увеличить его в том случае, если вы встретите проблему, которая не была предусмотрена его создателями.

Выбирайте между фреймворком и движком, основываясь на персональных предпочтениях, бюджете и целях. Как независимый разработчик, я предпочитаю фреймворки, поскольку они обычно проще для понимания и позволяют мне поступать так, как мне хочется.

Выбирайте собственный идеал. Далее приведен список фреймворков и движков, которые могут ускорить процесс разработки.

- Unreal Development Kit (www.udk.com) — коммерческий игровой движок, работающий на многих платформах. Он был разработан компанией Epic Games, создателем игры Unreal Tournament. Это говорит о том, что движок довольно качественный. Он использует собственный скриптовый язык.
- Unity (www.unity3d.com) — еще один коммерческий игровой движок, имеющий отличные инструменты и функциональность. Он тоже работает на множестве платформ, включая iOS и Android, а также браузеры, и прост в освоении. Он позволяет использовать множество языков для программирования игровой логики, но Java им не поддерживается.
- jPCT-AE (www.jpct.net/jpct-ae/) — портированная на Android версия движка jPCT, основанного на Java. У него есть несколько отличных особенностей, помогающих в трехмерном программировании. Он работает как со стационарными компьютерами, так и на Android. Его исходный код закрыт.
- Ardor3D (www.ardor3d.com) — очень мощный движок для трехмерного программирования, основанный на Java. Работает как для Android, так и для стационарных компьютеров, имеет открытый исходный код и отличную документацию.
- libgdx (code.google.com/p/libgdx/) — открытый фреймворк вашего покорного слуги, основанный на Java и пригодный для написания 2D- и 3D-игр. Работает на Windows, Linux, Mac OS X и, конечно же, на Android без модификаций кода. Вы можете разработать и протестировать приложение на стационарном компьютере без необходимости иметь под рукой устройство или медленный эмулятор

(или ожидать минуту, пока APK-файл загрузится на устройство). Возможно, после прочтения этой книги он понравится вам больше всего — в этом и заключался мой коварный план. Заметили ли вы, что этот пункт лишь немного больше, чем другие?

- Slick-AE (http://slick.cokeandcode.com) — портированная на Android версия фреймворка Slick, основанного на Java, построенная над libgdx. Он предоставляет множество различной функциональности и простой в использовании API для разработки 2D-игр. Конечно же, он кроссплатформенный и открытый.
- AndEngine (www.andengine.org) — приятный, основанный на Java, предназначенный только для ОС Android двухмерный движок, который частично основан на коде libgdx (он является открытым). Его концепции похожи на концепции знаменитого движка cocos2d, предназначенного для разработки игр для iOS.

Я предлагаю вам опробовать все эти варианты. Они могут несколько ускорить процесс разработки игр.

Ресурсы в сети Интернет

Интернет переполнен ресурсами, посвященными разработке игр. В общем, Google станет вашим лучшим другом, но есть несколько особенных сайтов, с которыми вам обязательно следует ознакомиться.

- www.gamedev.net — один из старейших сайтов, посвященных разработке игр. На нем вы можете найти огромное количество статей, связанных с различными вопросами разработки.
- www.gamasutra.com — еще один «старейшина» среди подобных сайтов. Он более ориентирован на индустрию, имеет множество статей и взглядов на профессиональный мир разработки игр.
- http://wiki.gamedev.net — большая энциклопедия, полная статей о программировании игр на различных платформах, языках и т. д.
- www.flipcode.com/archives — архивы теперь уже не функционирующего сайта Flipcode. Вы можете отыскать здесь несколько жемчужин. Хотя некоторые статьи несколько устарели, он по-прежнему является хорошим ресурсом.
- www.java-gaming.org — самое главное место для разработчиков игр на Java. Сюда часто заходят люди вроде Маркуса Перссона (Markus Persson), создателя игры Minecraft.

Завершающие слова

Сейчас 2:20 утра, и я сижу за кухонным столом со своим верным нетбуком.

В этой позе я провел все ночи последних нескольких месяцев. Надеюсь, я смогу вернуться к нормальному режиму сна.

Написание этой книги было для меня в радость (хотя по утрам не очень), и я надеюсь, что я ответил на ваши вопросы. Осталось еще множество нерассмотренных приемов, алгоритмов и идей. Мы изучили лишь верхушку айсберга. Многое ждет вас впереди.

Я уверен, что рассмотренный материал помог вам и заложил солидный фундамент, на котором мы сможете быстрее реализовывать новые идеи и концепции.

Вам не придется копировать массу кода. Более того — основная часть рассмотренных вопросов актуальна и для любой другой платформы (за исключением разницы между языками или API). Надеюсь, вы сможете взглянуть на общую картину и это позволит вам начать создавать игры вашей мечты.