Юрий Язев

Как самому написать мобильную 2D-игру

Москва
СОЛОН-Пресс
2017

УДК 004.94
ББК 32.973.2
Я 40

Ю. Язев

Как самому написать мобильную 2D-игру. — М.: СОЛОН-Пресс», 2016. — 476 с.: ил.

ISBN 978-5-91359-234-7

Книга посвящена разработке компьютерных и мобильных игр, которые сегодня доминируют на рынке. Именно 2D-игры получили большую популярность на смартфонах и планшетах. Прочитав книгу, Вы самостоятельно сможете создавать новые цифровые развлечения и, возможно, Ваша игра будет завтра популярна на рынке.

Эта книга расскажет: как разрабатывать компьютерные и мобильные игры с помощью бесплатного, мощного, многофункционального и кроссплатформенного движка Torque 2D. В качестве инструмента используется компьютер с операционной системой Windows. В книге обосновывается: почему среди множества имеющихся на рынке средств для разработки игр читателю стоит сделать выбор в пользу этого движка.

В качестве введения рассказывается об истории игровой индустрии и цифровых развлечений. Далее вниманию читателя предлагаются основы движка, его объекты, конструкции и их использование. Уделяется внимание игровой физике. Львиную долю книги занимают практические занятия — подробные инструкции по разработке игр. В течение повествования с нуля разрабатываются 4 полностью законченных игры. Затем происходит их портирование на другие программно-аппаратные платформы. Читателю предлагается: оснастить одну из игр рекламой и опубликовать на площадке цифровой дистрибуции Google Play. В последней главе будет подробный рассказ об устройстве Leap Motion и о том, как его использовать в играх, таким образом, будет разработана игра, управление в которой реализовано с помощью этого контроллера.

Для разработки игр используются самые последние версии движка Torque 2D: 3.2 и 3.3.

Исходный код примеров и другой сопутствующий материал можно скачать с сайта автора: *http://www.t2d-dev.ru*.

ISBN 978-5-91359-234-7

В память о Крисе Касперски

Рецензия

Главной мотивацией, сподвигшей меня когда-то на изучение программирования, было желание реализовать одну игру. Игра, в которую я играл в 1989-м году на ПЭВМ «Микроша», называлась «Лестница», и мне очень хотелось портировать ее на PC. Впоследствии оказалось, что я был не одинок в своем стремлении, но в то время, когда моя мечта наконец воплотилась в реальность (конец 90-х — начало 00-х), сделать это было значительно сложнее, чем сейчас — готовые игровые движки тогда не были широко доступны и всю игровую механику нужно было кодировать на достаточно низком уровне. Программирование простой двумерной игры требовало знаний, находящихся в добром годе самостоятельного изучения материала.

Сейчас же, когда программистам доступно не менее шести готовых и хорошо документированных игровых движков, создание игры становится доступно любому энтузиасту — от инженера с дипломом мехмата и до папы-гуманитария, который просто хочет сделать детям игру, что называется, «своими руками».

И, поэтому мне особенно приятно рецензировать книгу, написанную старым автором журнала «Хакер», ведь в этом 476 страничном труде, посвященным бесплатному игровому движку Torque 2D, читатель найдет исчерпывающую информацию, с помощью которой он по окончанию книги сможет с нуля сделать простую аркадную или логическую игру по своему вкусу. Разумеется, для понимания материала читателю потребуется определенный базовый уровень знаний в области программирования на C++ или C#, но достичь его можно буквально за пару месяцев вдумчивого самостоятельного обучения. Уверен, что с этой книгой путь к реализации собственной игры будет длиться не более полугода. Всего 6 месяцев до мечты — разве это много?

Александр Лозовский,
редактор журнала «Хакер» (*xakep.ru*)

Оглавление

Благодарности

В первую очередь я благодарен своим родителям: Евгению Юрьевичу и Елене Владимировне Язевым. Их забота и поддержка помогают мне преодолевать любые препятствия.

Также хочу поблагодарить сотрудников редакции журнала «Хакер», в частности, главного редактора — Илью Русанена и редактора рубрики Кодинг — Александра Лозовского, с которым мы работаем над статьями уже более 8-ми лет. У вас лучший журнал об IT в России и мире, друзья! Я рад участвовать в его создании вместе с вами.

Предисловие для второго издания

Во втором издании исправлены недочеты и ошибки, попавшие в первое издание. Поэтому оно содержит минимум нового материала. Если вы, дорогой читатель, купили первую книгу «Волшебство момента вращения», то эту книгу можете спокойно пропустить, так как не найдете в ней ничего нового по сравнению с первой книгой. Если же вы не читали первую книгу и хотите научиться создавать мобильные игры разных жанров, портировать их на мобильные платформы и публиковать готовые продукты на площадках цифровой дистрибуции, то рекомендую обратить свое внимание на данную книгу.

Введение

Оглавление

1 О книге

Автор уже много лет занимается разработкой компьютерных игр: он разрабатывал игры как для себя — для самообразования, игры для обучения, в качестве сопроводительных материалов к журнальным статьям, так и принимая участие в больших и не очень проектах. За все это время он не переставал повышать свою квалификацию, читая публикации других авторов. Тем более игровая индустрия — быстро изменяющаяся отрасль бизнеса, участвуя в которой необходимо всегда обновлять свои знания. При этом ему удалось выделить главное правило: самые ценные обучающие материалы те, которые содержат не только теоретическое основание, но и приводящие полезные жизненные примеры с подробными пояснениями того, что они делают и как они это делают, то есть тщательными разъяснениями приводимых листингов.

Поэтому, планируя написать свою книгу, автор решил уделить особое внимание практическим экспериментам, что поможет читателям в пол-

ной мере усвоить предложенный материал. Также, выполняя примеры, читатель не потеряет нить понимания материала, а это, в свою очередь, позволит не потерять интереса к книге.

2 Краткое содержание книги или чем мы будем заниматься на ее протяжении

Глава 1. Введение в Torque 2D. В первой главе речь пойдет на вводные темы: об истории движков семейства Torque, об их развитии. Мы так же коснемся основ и философии игр.

Глава 2. Скрипты и код. Вторую главу мы посвятим изучению бесспорно главного компонента движка Torque 2D — скриптового языка Torque Script. Узнаем, как выполняются скрипты на этом языке. Рассмотрим разницу между C++ и Torque Script. Построим движок из исходников и приступим к экспериментам. Кроме того, изучим язык TAML, служащий для описания ассетов. В конце главы мы выберем нужные для разработки игр инструменты.

Глава 3. Эксперименты. Здесь мы рассмотрим комплектующие части движка Torque 2D. Затем создадим заготовку для наших игр.

Глава 4. Объекты движка. В этой главе мы изучим: какие объекты предоставляет движок Torque 2D, какими свойствами они обладают, для чего служат и как их использовать.

Глава 5. Физические свойства и взаимодействие. В пятой главе мы рассмотрим физику, применяемую в компьютерных играх, в общем, и в Torque 2D, в частности. Мы обсудим физические свойства объектов, каким образом они взаимодействуют друг с другом. Физика придает правдивость действиям, происходящим в виртуальном пространстве.

Глава 6. Разработка аркадной игры Asteroids. С шестой главы мы переходим к практическим занятиям, к непосредственной разработке игр. Первой игрой, которую мы создадим, будет аркада Asteroids. Это очень специфичная игра, имеющая долгую историю.

Глава 7. Torque 2D и мобильные платформы. В этой главе мы обсудим кроссплатформенные возможности движка Torque 2D. Кроме того, мы подготовим Asteroids для запуска на мобильных платформах и создадим билды для Mac OS X, iOS и Android.

Глава 8. Менеджер проектов для Torque 2D. В этой главе мы разработаем приложение для создания заготовок проектов. То есть автоматизируем этот процесс.

Глава 9. Простой физический эксперимент. Мы расширим свои знания относительно физических эффектов в Torque 2D и на их основе построим незамысловатый, но интересный в плане физики игровой эксперимент.

Глава 10. Логическая игра Magic Mancala. Эта глава является центральной во всей книге. В ней полностью описывается процесс создания достаточно большой и законченной логической игры.

Глава 11. MagicMancala в Google Play. Разработав в предыдущей главе игру, в текущей мы подготовим билд для размещения в магазине приложений Google Play, встроим рекламу и, наконец, зальем игру в магазин.

Глава 12. Использование контроллера Leap Motion. В последней главе мы разработаем еще одну игру, управление в которой будет организовано посредством контроллера Leap Motion.

3 Об игровой индустрии

Современная игровая индустрия уже полностью сформировавшаяся отрасль экономики с многомиллиардной выручкой в год. Возраст родной индустрии насчитывает уже около 40 лет! Не слабо для забавы компьютерных энтузиастов! А ведь именно из таковой она выросла. В настоящее время в ней заняты тысячи специалистов по всему миру, занимающие самые разные должности. В игровой индустрии имеется больше ролей, чем в какой либо другой отрасли. Приведу для примера только роли разнообразных программистов: программист игровой механики, программист инструментария, программист физического движка, программист игрового движка, программист искусственного интеллекта и другие. Здесь заняты художники, иллюстраторы, модельеры трехмерной графики, аниматоры, дизайнеры, в том числе: гейм-дизайнеры, левел-дизайнеры, дизайнеры интерьера/экстерьера, писатели — авторы текстов: сюжетных линий, диалогов, др., музыканты, создающие звуковое оформление, артисты, которые выполняют движения для анимационных последовательностей, впоследствии используемые для игровых персонажей. Этот список можно продолжать еще очень долго.

Поскольку, книга, находящаяся в данный момент перед Вами, дорогой читатель, посвящена разработке игр и, следовательно, игровой индустрии, то будет хорошей идеей окунуться в историю последней.

А началось все в начале 70-х годов прошлого столетия. Рождение игровой индустрии напрямую связано с компанией Atari и ее основателем Нолаком Бушнеллом. Ведь именно эта компания в 1972-м году выпустила первую коммерчески успешную игру Pong. В то время она пред-

назначалась для игровых автоматов. Затем были 7 поколений игровых консолей, текущее (Xbox One, Play Station 4) — уже восьмое. Однако для нас в данный момент — это не так интересно, любознательных я направляю в Википедию: к статье про историю индустрии компьютерных игр (http://ru.wikipedia.org/wiki/Индустрия_компьютерных_игр), где есть ссылки на описание каждого поколения игровых консолей.

Для нас из всего этого важны лишь некоторые даты выхода легендарных игр:

— Asteroids — 1979 год (Namco)
— Pac-Man — 1980 год (Atari)
— Tetris — 1984 год (ВЦ Академии наук СССР)

Эти игры во много повлияли на развитие игровой индустрии. Наверное, вы слышали фразу: каждый программист желает разработать Tetris. И, конечно, он может это сделать, только историю уже не изменить. А мы плавно вернемся к истории игровой индустрии.

4 Об игровых движках

Еще 15 лет назад кодовая база для игр создавалась с нуля внутри компаний разработчиков определенных игр. Однако немного позже (примерно в середине первого десятилетия 21-го века) вслед за разработкой других компонентов игр, таких как: арт, звук и другое, создание движков вышло на аутсорсинг, то есть выделилось в отдельную индустрию. Но это происходило постепенно. В то же время, и это имело место в параллельной реальности, поэтому было скорее исключением, чем правилом, еще в 90-е были игры на готовых движках: DOOM, Quake от id Software. Но как я уже сказал: это было редким исключением, а правилом стало в начале 2000-х. Это имело место из-за дороговизны данных технологий — движков, поскольку в то время этот сегмент рынка был полностью монополизирован компанией id Software. А когда сформировалась обособленная кодовая индустрия, бизнес приобрел другой вид, появились движки широкой ценовой категории. Сначала рынок заполнился инструментами разработчиков, другими словами — фреймворками, представляющими собой, скорее, графические движки, нежели игровые, отличающиеся от последних наличием лишь прослойки над графическим api и отсутствием внутриигровых редакторов, как то: средств для level-моделинга, импорта объектов, текстурирования, загрузки и анимации персонажей и другое. Для примера можно привести: Ogre, DarkGDK. Позже на рынок вышли другие — более прокаченные

игроки, предложившие разработчикам полноценные движки: Torque 3D, Unity 3D, Unreal Engine 4.

Начиная с 1992-го года, когда была выпущена одна из первых трехмерных игр — Wolfenstein 3D, снискавшая огромную популярность, вместе с тем был определен некий вектор последующего развития игровой индустрии — все в 3D. После этого все успешные игры должны были быть, как минимум трехмерными. Это продолжалось десятилетие. Затем выяснилось, что в игры стали играть не только юноши и мужчины, но так же дети, женщины и люди старшего поколения, а этой категории игроков нужны другие развлечения. Таких игроков назвали казуальными, играющими в компьютерные игры от случая к случаю и желающие получить от них эстетическое удовольствие, в отличие от хардкорных игроков, играющих, чтобы сражаться. Таким образом, к середине первого десятилетия 21-го века образовалась ниша казуальных игр, по своей сути, незамысловатых, но красочных, не требующих бешеного взаимодействия с клавиатурой и мышкой головоломок. С технической точки зрения такие игры были проще трехмерных хардкорных шутеров, поэтому для их разработки не понадобилось сильной модификации универсальных движков, но, зато в индустрии появились новые игроки, такие как: Torque Game Builder, HGE и другие.

Примерно в это же время на волне бешеной популярности World of Warcraft на игровом рынке образовался новый тренд — глобальные многопользовательские онлайн игры. Результатом чего стало появление специальных движков, предназначенных для MMO-игр: HeroEngine, BigWorld. Большая часть универсальных движков, заточенных под сингплеерные игры, оказалась не у дел, то есть не смогла выжить в изменившейся экосистеме. Тем не менее, некоторые универсальные движки были адаптированы, так, например, сторонние компании создали для Unity 3D серверные решения, в том числе: Photon, SmartFox. У движка Torque 3D вообще не было проблем с образовавшимся трендом, поскольку этот движок изначально имел клиент-серверную архитектуру.

Но и это оказался не предел. В 2006-м году подобно эффекту разорвавшейся бомбы неожиданно вспыхнули социальные сети: Facebook, Одноклассники, а затем ВКонтакте. И пока аналитики анализировали этот необычайный успех веб-сервисов, которые, по сути, в технологическом плане не принесли ничего нового, вслед за социальными сетями произошел всплеск интереса к web-играм. На него разработчики движков отреагировали довольно оперативно, предоставив своим пользователям возможность запуска игр прямо в браузере посредством установ-

ки плагина для последнего. В результате этого, конечные пользователи в браузере получили игры по качеству почти ничем не уступающие клиентским. Вместе с этим на рынке появились новые игроки — для новой отрасли внутри индустрии. С популярностью web-игр дополнительное распространение получила Flash технология, в результате чего скромная Macromedia (разработчик Flash) была приобретена могущественным Adobe. А на базе Flash стали появляться не только игры, но и движки для их разработки. Что касается универсальных движков, то для запуска игр в браузере, как я уже говорил выше, для них предоставляются плагины, подключаемые к браузеру. Это коснулось как Torque 3D, так и Unity 3D.

Под конец первого десятилетия 21-го века случились мобильные технологии. Как гром среди ясного неба появились мобильные устройства по мощности сопоставимые с ПК средней ценовой категории и способные запускать мощные игровые приложения со всеми спецэффектами, которыми обладали низкоуровневые графические интерфейсы. На что разработчики игровых движков ответили в некоторых случаях созданием специализированных конверторов, создающих нативный для конкретного оборудования код (как, например, Unity 3D), а в других — модернизировали свои продукты для кроссплатформенности (к примеру, Torque 2D). Также на рынке появились новые игроки, предлагающие кроссплатформенные фреймворки для всего парка мобильных устройств, выполняющиеся со скоростью нативного кода. Среди подобных средств: Corona SDK, Marmalade SDK, AGK (App Game Kit).

Последний на текущий момент тренд игровой индустрии — виртуальная/дополненная реальность. Подавляющее большинство современных игровых движков уже обзавелись поддержкой данной технологии, среди них: Torque 3D, Unity 3D, Unreal Engine 4. Чтобы реализовать поддержку очков VR разработчикам движков надо не только добавить визуализацию на второй экран (для второго глаза) с отличным от первого содержимым (так как первый и второй глаза могут видеть отличающиеся сцены), но и так же добавить поддержку управления с новых устройств ввода, которые различны для разных гарнитур VR и пока не стандартизированы.

За последние десять лет на рынке интерактивных развлечений кроме всегда на нем присутствующих одиночных игр появились еще четыре категории игровых продуктов. Перечислим все категории:

1) Однопользовательские игры
2) Многопользовательские онлайн игры
3) Игры для социальных сетей

4) Мобильные игры

5) Игры для VR

При этом, хотя MMO-игры разрабатываются уже довольно давно и уже имели некоторый успех, это было скорее исключением, нежели правилом, так как было не заметно на фоне синглплеера.

Для разработки каждого типа игр есть определенный набор движков, потому что с технической стороны между всеми типами игр имеются большие различия. Но как мы увидели в вышеприведенном описании, есть универсальные движки, разработчики которых совершенствовали их, следуя изменениям внутри индустрии, и с помощью этих движков можно разрабатывать игры любого типа. К числу таких движков относятся Torque 2D / 3D. Движки семейства Torque являются центральной темой данной книги.

5 Семейство Torque. Предыстория

История движков Torque берет свое начало в далеком 2001-м году. В то время компанией-издателем Sierra была выпущена компьютерная игра Tribes 2 (рис. 0.1.), (а ведь ранее это издательство так же приложило руку к Half-Life).

Разработкой Tribes 2 занималась компания Dynamix. Ранее она выпустила такие тайтлы, как: Earthsiege (1994), Earthsiege 2 (1995), Starsiege

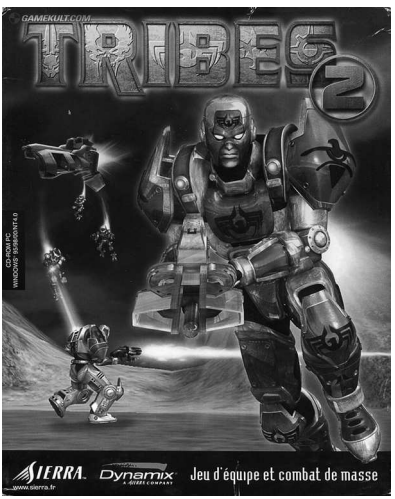

Рис. 0.1. Tribes 2

(1998) (оригинальная версия Tribes), из чего в итоге следует, что кодовая база для движка игры Tribes 2 начала свое существование в 1994-м году! Отрываясь от игр, это повествование еще раз напоминает нам, что современные программные продукты имеют огромную базу унаследованного кода, история которого насчитывает десятилетия! Tribes 2, в отличие от первой части, была построена на основательно модифицированном игровом движке. По сюжету в далеком будущем (события происходят в 40-м веке) люди и киборги, разделенные на четыре фракции с помощью высокотехнологичного оружия, как это часто бывает, выясняют отношения. По геймплею эта трехмерная игра была многопользовательским командным онлайн шутером с видом от первого лица, в котором могли принять участие до 64-х игроков или ботов. Помимо пеших пробежек с ракетницей в руках геймплей предлагал использование реактивного ранца и разного вида транспорта. Надо полагать, что в то время мультиплеерные игры только зарождались, но уже тогда Tribes 2 предлагал широкий выбор режимов игры, в том числе: Capture the Flag, Deathmatch, Arena, Rabbit, Hunters, Duel-MOD, Siege, Gauntlet и Bounty. Во многие из этих режимов можно было играть в командном варианте или «одному против всех».

Я привел это описание с целью показать: какая технология послужила основой для движка Torque.

После издания игры Tribes 2, несмотря на ее успех, через некоторое время дела у издателя Sierra стали совсем плохи, и в итоге оно было расформировано.

Однако в 2000-м году ключевые сотрудники Dynamix организовали новую фирму — Garage Games, которая стала заниматься совершенствованием движка от игры Tribes 2 и продавать его, как самостоятельную технологию для разработки компьютерных игр. Именно тогда технология получила свое имя — Torque Game Engine (TGE). Первая версия движка была выпущена в августе 2001-го. В последующие годы компания стала продавать движок вместе с тем, занимаясь его развитием. Сначала Torque для визуализации использовал подсистему OpenGL. Однако в 2006-м году компания Garage Games осуществила большое технологическое обновление: видео подсистема для визуализации стала использовать DirectX 8.0 вместе с поддержкой шейдеров, в связи с чем движок получил новое название Torque Shader Engine. Но в 2007-м году, в связи с маркетинговыми соображениями движок был переименован в Torque Game Engine Advanced (TGEA). Кроме новой системы визуализации Torque приобрел атласы: динамично подгружаемые части террейна (ландшафта). Следующий большой шаг в развитии, за которым последо-

вало новое переименование, произошел в 2009-м году. На этот раз для визуализации трехмерных сцен движок стал использовать DirectX 9.0c, а новое имя стало просто Torque 3D (T3D).

Между тем GarageGames занималась разработкой не только одного продукта. После того как в середине первого десятилетия 21-го века резко возросла популярность двумерных казуальных игр, Garage Games выделила в отдельное направление развитие движка, заточенного для 2D игр, который получил имя Torque 2D (T2D). Он, в отличие от T3D, для вывода графики использовал OpenGL. Позднее эти два направления развития (2D и 3D версии движка) получили множество ответвлений: были разработаны версии под разные операционные системы, такие как: Windows (первоначально), Linux, Mac OS X, iOS, а так же версии для новых аппаратных платформ: iPad, iPhone, игровые консоли Wii, Xbox 360 (нативный код) и XNA-версия для PC и Xbox 360. Здесь и далее, говоря о Windows, я подразумеваю современные версии данной операционной системы, основанные на NT, то есть, начиная с Windows XP и выше.

Вследствие бурного развития двумерная версия движка для операционной системы Windows получила название Torque Game Builder, а для Mac OS X вместе с iOS — iTorque 2D.

Время шло, в области разработки игр появлялись и исчезали новые инструменты, но Torque продолжал свое бытие. В это время разработчики продолжали его совершенствование, постепенно снижая стоимость. Таким образом, весной 2011-го года цена снизилась до 100 долларов США. С течением времени, когда стало очевидно, что некоторые ответвления движка не получили ожидаемой от них популярности и распространения среди игроделов, было решено оставить только три флагманских продукта: Torque 2D (по цене 128 долларов), iTorque 2D ($149) и Torque 3D ($100). Затем события стали развиваться еще интереснее. Чуть более года спустя, осенью 2012-го года Garage Games передала Torque 3D в сообщество Открытых исходников (Open Source) под удобной лицензией MIT. Чтобы не вникать в нюансы разных лицензией только отмечу, что среди множества разнообразных лицензий и описывающих их документов есть, на мой взгляд, один весомый аргумент, который играет важную роль при выборе лицензии — это необходимость открывать исходный код своей программы, созданной с использованием Открытых исходников. Так, есть лицензии, которые требуют его открывать, а есть такие, которые этого не требуют, MIT как раз относится ко второй группе. На момент передачи в Open Source Torque 3D имел версию под номером 1.2. Вместе с тем, двумерный Torque продолжал продаваться по цене 128

долларов за версию для Windows, и 149 долларов за версию для Mac OS X / iOS. Но, спустя 6 месяцев, — в конце января 2013-го года Torque 2D так же был отдан в сообщество Open Source под MIT лицензией! Это стало очень знаменательным событием для всех фанатов Торка! Не менее значимое, чем аналогичное произошедшее с T3D. При этом iTorque 2D продолжал продаваться. Вместе с тем, при большом желании за деньги так же можно было купить и Torque 2D для Windows. Сейчас T2D и T3D свободные и открытые, их можно скачать с github.com.

В настоящее время, когда ранее закрытые технологии стали свободными, Garage Games избрала другой вид бизнеса. Если раньше движок продавался в большинстве своем разработчикам как большим, так и малым — Indie, то сейчас деятельность компании в основном направлена на сотрудничество с образовательными учреждениями, в которых Torque используется не только, как способ демонстрации разных экспериментальных процессов, но и как средство для обучения программированию и геймдизайну, то есть в некоторых учреждениях используется по своему прямому предназначению. Кроме того, с Garage Games ведут сотрудничество многие другие компании программной индустрии, такие как: Intel, Microsoft, NVidia, Facebook, Sony и многие другие. Они тоже используют движок Torque для создания визуализации в своих продуктах. Лицензиарами движка являются крупные разработчики и издатели компьютерных и видео игр, в том числе: Electronic Arts, Ubisoft, Capcom и другие.

Но все же Garage Games начинала как Indie-компания, создающая качественные средства разработки игровых приложений для небольших Indie-студий, и она по- прежнему верна своим истокам как видно из вышеприведенного описания.

6 Сообщество

За годы существования движков семейства Torque вокруг них сформировалось хардкорное сообщество профессионалов игровой индустрии, а так же любителей создавать интерактивные приложения. Среди форумчан можно встретить сотрудников Garage Games и разработчиков из комьюнити (рис. 0.2).

Форум сайта GarageGames (http://www.garagegames.com) отражает бурную деятельность сообщества; здесь всегда готовы помочь советом, ссылкой, куском кода, если в этом будет необходимость. Можно выделить три раздела для общения торковских пользователей: блоги создают и пишут в основном те, кто организовал новую студию, раз-

работал новую игру с помощью Torque и хочет этим похвастаться, стоит отметить, что это нисколько не противоречит правилам разработчиков, ведь надо как то рассказать о своем продукте среди единомышленников, и это встречается очень хорошо. Есть и такие замечательные участники сообщества, кто добился в разработке игр на Torque совершенства (разработал полезную функциональность, исправил трудно устранимый баг и т.д.) и желает поделиться знанием и опытом с другими членами сообщества, такие люди создают так называемые ресурсы, в которых подробно описывают свои достижения и способ их реализации для помощи другим разработчикам. Между тем большая часть общения происходит непосредственно на форуме, разделенном на категории, относящиеся к определенным движкам, а так же на подкатегории, которые различаются темами обсуждения. Здесь можно задать любой вопрос, касающийся темы разработки игр: геймдизайн, арт, кодинг, тестирование и много другое. Участники сообщества обычно стараются помочь, в том случае, если вопрос логически верно сформулирован и грамотно написан на английском языке. От вас, конечно, не ждут литературного стиля изложения, так как знают, что технология Torque распространена по всему миру, и учитывают, что английский может быть не вашим родным языком, тем не менее, надо стараться писать понятно.

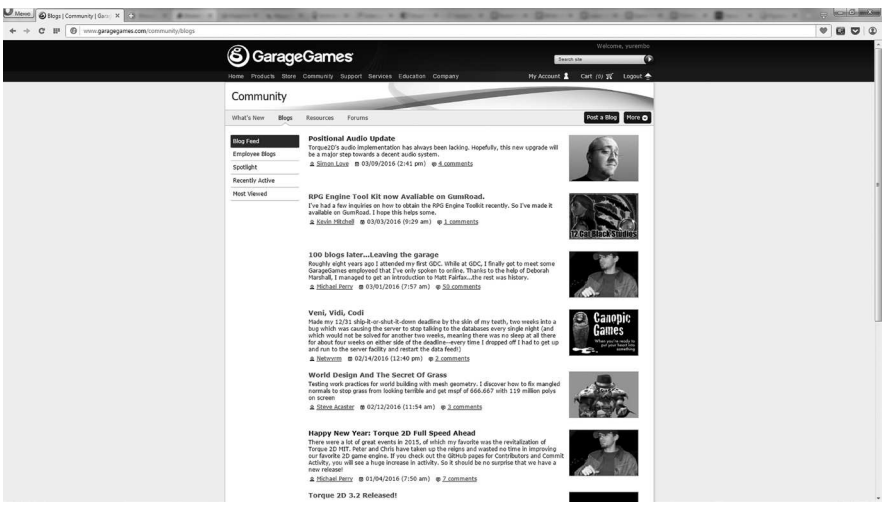

Рис. 0.2. Комьюнити

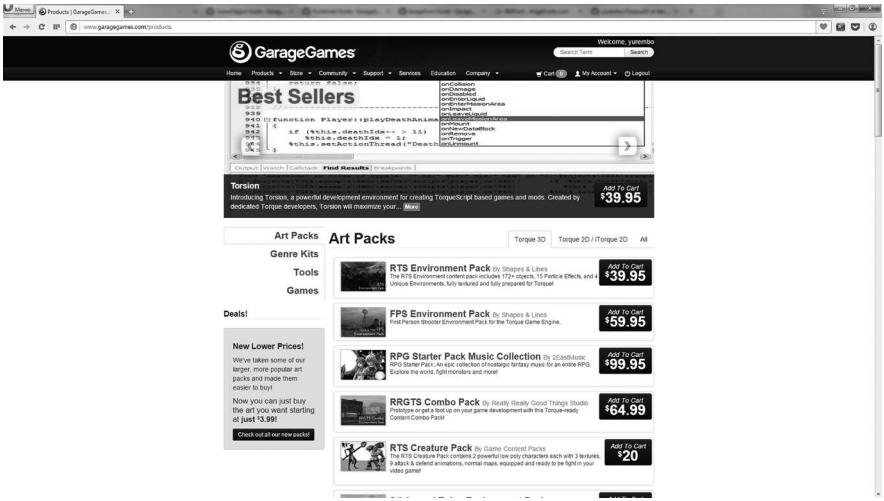

Рис. 0.3. Онлайн магазин GarageGames

Кроме того, на сайте есть магазин цифровой дистрибуции — Store, в котором продаются средства для разработки игр, а так же готовые игры. Среди них есть: редакторы скриптового кода, инструменты для создания визуальных эффектов, средства для создания искусственного интеллекта, готовые материалы изображений, текстур, моделей и многое другое (рис. 0.3). Кроме того, для продажи имеются разные встраиваемые редакторы: диалогов, создания инвентаря, террейнов, заготовки для MMORPG и многое другое.

7 Почему Torque 2D?

В настоящее время на рынке имеется масса движков для разработки игр, самые популярные из которых: Unity 3D, Unreal Engine 4, Torque 2D / 3D, CryEngine, Cocos 2D-X, Defold, Godot и другие. Отсюда вытекает закономерный вопрос: почему автор предлагает использовать Torque 2D? Стоит сразу отметить: компания GarageGames не заплатила ему ни копейки за то, чтобы он написал книгу про их движок. Поэтому для автора тут нет никакой выгоды.

Игровые движки появляются и исчезают, приходят и уходят со сцены. Одни движки сейчас в моде, другие позабыты. В то же время игровой движок Torque 2D держит планку уже второе десятилетие и не со-

бирается сдавать позиции в индустрии, наоборот, завоевывает все новые и новые территории. Это происходит благодаря тому, что он максимально гибкий, на нем можно разрабатывать игры всех жанров и запускать на любых программно-аппаратных платформах. Если, например, на следующей неделе появится какое-то популярное среди игроков устройство, то в сообществе создадут модуль движка для поддержки этой платформы. И станет возможным создавать для нее игры на Torque 2D.

Технические характеристики движка Torque 2D мы подробно рассмотрим в 1-й главе. В то же время по техническим характеристикам имеющиеся на рынке игровые движки во многом похожи, поскольку все разработчики стараются максимально оптимизировать свои продукты, поэтому больших различий нет. Лидирующие движки кроссплатформенные, имеют оптимизированный графический пайплайн, удобный высокоуровневый скриптовый язык, поддерживают качественный звук, различные устройства ввода, мультиплеерные режимы игры по сети и многое другое.

Выбор движка — во многом философский вопрос. Поскольку после его выбора вам, возможно, придется провести с ним ближайшие несколько месяцев или даже лет, пока вы разрабатываете на нем игру. Во главу угла встает удобство использования и близость стиля разработки с помощью выбранного движка. Torque 2D имеет особый стиль разработки игр, в котором не надо использовать средства, схожие с графическими редакторами и системами трехмерного моделирования, используя которые, я часто забываю предназначение кнопок пользовательского интерфейса. В Torque 2D для создания, анимации, оживления персонажей, построения уровней, взаимодействия с объектами используется стройный многофункциональный язык, позволяющий сделать манипуляции коротко и точно. Torque 2D — движок для тру программистов.

На движке Torque 2D (а так же на предыдущей версии движка для Mac — iTorque) создано впечатляющее количество великолепных игр для различных платформ. На сайте GarageGames в разделе Games приведен длинный список некоторых игр, созданных на основе данной технологии: http://www.garagegames.com/games/2d-games.

Игры на движке Torque 2D продаются во всех популярных магазинах цифровой дистрибуции, их можно скачать со многих популярных игровых сайтов: Steam, BigFish, PlayStation Network, Xbox Live, Nintendo Network, App Store, Google Play и многих других.

Рис. 0.4. Раздел двумерных игр сайта GarageGames

Среди этих игр есть платформеры, стратегии, аркады, логические игры, квесты, симуляторы и многое другое. Все что душе геймера угодно!

С помощью Torque 2D можно свободно разработать любую двумерную игру любого жанра с невероятными спецэффектами! Этим мы с вами и займемся в последующих главах этой книги.

Почему 2D?

Хотя мы рассматриваем кроссплатформенные игры, в большей степени данная книга нацелена на разработку игр для мобильных устройств. На PC, Mac и консолях распространение получила 3D-графика, между тем на смартфонах, планшетах и других миниатюрных гаджетах балом правят двумерные игры. Почему так получилось? Потому что в такие игры удобней играть на устройствах с маленьким экраном. Такие игры больше нравятся аудитории мобильных — казуальных игроков.

Достаточно вспомнить популярнейшие мобильные игры: Angry Birds, Cut the Rope, Flappy Bird, Limbo и многие другие — все они двумерные!

Глава 1. Введение в Torque 2D

Оглавление

1 Начало

Настала пора окунуться в разработку игр и рассмотреть техническую сторону семейства движков Torque. Как я уже говорил во введение: в настоящее время особенно, в связи с широким распространением разнообразных мобильных устройств, обладающие высокопроизводительной начинкой, двумерный игры обрели вторую жизнь после их вытеснения с рынка 3D-играми. А на миниатюрных устройствах (смартфонах, планшетах) трехмерные игры не получили распространения, пользователям этих устройств не интересно играть в боевики с трехмерной графикой. Именно по этой причине, а вовсе не потому, что мобильные девайсы по мощности не могут тянуть 3D-игры, последние не стали популярны на мобильных платформах. Зато с 2D все получилось очень хорошо, и пользователям это нравится.

В связи с этим и более ранним бумом казуальных игр на PC и Mac в семействе движков Torque появился двумерный вариант: Torque Game Builder для Windows и iTorque 2D для Mac, позже для iOS. Это мы уже проходили во введении, но там мы не коснулись событий произошедших с движком после его отдачи в Open Source, потому что это происходит в настоящее время и историей пока что не является. А дела развернулись следующим образом. Torque Game Builder перестал продаваться (однако его заменил Torque 2D для Windows (его можно было приобрести за $128)), iTorque 2D по прежнему можно купить за $149. Плюс в семействе Torque случилось пополнение — пришел свободный и полностью открытый Torque 2D MIT. Поскольку для визуализации он использует кроссплатформенную графическую подсистему OpenGL, то сразу после выхода (февраль 2013-го года) исходный код движка можно было скомпилировать под 3 операционные системы: Windows, Mac OS X и мобильную iOS. Следовательно, он изначально был кроссплатформенным. Torque 2D написан на языке C++ с ассемблерными вставками для Intel-подобных процессоров. Таким образом, в нем есть два способа компиляции: 1) из Visual Studio — для Windows; 2) из Xcode — для Mac OS X и iOS; учитывая определенные нюансы, касающиеся работы в этих операционных системах. Так, чтобы движок запускался в Mac OS X (iOS), C++ код обернут кодом на Objective C — родной язык для Apple-платформ.

Но на этом освоение новых платформ и операционных систем не прекратилось! Уже в ноябре 2013-го года открытым сообществом разработчиков была выпущена версия с поддержкой работы в самой распространенной операционной системе для мобильных устройств — Android. Таким образом, стало возможно запускать игры, разработанные на Torque 2D на планшетах и смартфонах с Android! Это потрясающее событие! А два месяца спустя в январе 2014-го года программистом — давним участником сообщества пользователей движков Torque, скрывающимся под ником Frogger, был подготовлен и выложен для скачивания порт Torque 2D для операционной системы GNU Linux. Превосходно! Отчетливо видно как динамично происходит развитие движка в Открытых Исходниках!

Но не все коту масленица (русская поговорка), есть так же отрицательные моменты, произошедшие после передачи движка в Open Source. Torque 2D по сравнению с Torque Game Builder лишился всех встроенных мастеров и редакторов. Очевидно, что это произошло по причине юридических соглашений с третьими фирмами. Эти редакторы представляли собой визуальные средства разработки подобно имею-

щимся в визуальных средах программирования, таких как: Visual Studio, Delphi, Xcode, Eclipse и других. Мастера и редакторы служили для быстрой настройки свойств объектов разрабатываемой игры, добавления однотипных игровых элементов (например, таких как текстовые надписи), наглядного подхода к проектированию и разработке игр. Однако теперь, к сожалению, весь этот незаменимый инструментарий удален. Но ни в коем случае не стоит расстраиваться, потому что все то, что можно было сделать из редактора, можно сделать несколькими строками кода на скриптовом языке Torque Script (о нем речь пойдет в следующей главе). В данный момент внутри сообщества разработчиков идет работа по созданию с нуля этих потерянных редакторов.

Когда писались эти строки (начало 2016-го года) в online магазине компании Garage Games все еще можно купить устаревшие продукты: iTorque 2D и Torque 2D для Windows, остановившиеся на версиях 1.6 и 1.8, соответственно. Между тем оценить внутриигровые редакторы можно в свободно распространяемой демо-версии Torque 2D для Windows с ограниченным сроком использования — 30 дней. При этом версия движка для Mac OS X не имеет демо-версии. Тем не менее, для нас это совсем не критично, так как для изучения и разработки своих игр мы будем использовать свободный движок Torque 2D MIT. Последней на момент написания этих строк версией движка являлась версия 3.3. После того как T2D был отдан в сообщество «Открытых исходников», его развитие происходит очень быстро — важные обновления могут иметь место каждые 2 месяца! Поэтому мы с вами будем следить за тенденциями его развития, отражая это в тексте книги и своих проектах.

2 Torque 2D изнутри

До сего момента мы уже много говорили об игровых движках, однако так и не дали определение этому магическому дуплету слов. Игровой движок — это центральная часть, основной компонент игры, объединяющий и связывающий все остальные визуальные, не визуальные, звуковые и другие материалы. Движок занимается всем жизненным циклом игры. При ее старте он выводит меню, производит настройку технических и программных средств (когда пользователь изменяет опции в меню Options). Во время загрузки игры он подгружает уровни, игровые объекты; рассматривая 2D-игры, ими могут быть: текстуры, музыка/звуки, игровая логика, логика искусственного интеллекта и многое другое. Также во время загрузки уровня движок строит игровые примитивы, из

которых получаются сложные предметы. После загрузки уровня в процессе игры движок продолжает свою работу: он прорисовывает игровые элементы и выводит их на экран дисплея, воспроизводит звуки, получает и обрабатывает пользовательский ввод (с мыши, клавиатуры, джойстика и/или любых других устройств ввода), выполняет всю игровую логику, в том числе логику ИИ, занимается физическими вычислениями, подсчетами очков, жизней и т.п. В том случае если в движке реализована поддержка мультиплеерных игр и загруженная в данный момент игра относится к этому классу, движок выполняет сетевое взаимодействие поверх низкоуровневых протоколов. При переходе с уровня на уровень движок выгружает из памяти пройденный и загружает туда следующий. Когда пользователь желает покинуть игру, выбирая в меню паузы пункт Exit, движок сохраняет текущий игровой прогресс пользователя, осуществляя дисковый ввод/вывод, уничтожает в памяти все игровые объекты и показывает главное меню и/или завершает свою работу, удаляя из памяти объект движка. Вот только краткий список дел, выполняемых игровым движком, на самом же деле у него гораздо больше работы, выполняемой им не только не обращая внимания игрока, но и не требуя заботы прикладного программиста, использующего движок.

Из-за того, что игровой движок содержит такие системные модули как: визуализатор — на низком уровне обрабатывает (оптимизирует объекты, деля их на треугольники) и отображает предметы, передавая данные видеоадаптеру, проигрыватель звука — представляет данные звуковой карте, сетевой обработчик — передает информацию по сети, интерпретатор скриптового кода, а так же другие, то игровой движок, по сути, относится к системному программному обеспечению.

Torque 2D включает в себя все вышеперечисленное. Он является высокопроизводительным, кроссплатформенным, гибким для использования игровым движком с открытым исходным кодом, подходящим для разработки двумерных игр любых жанров. Его исходный код под лицензией MIT доступен на GitHub. На Torque 2D можно разрабатывать игры с помощью трех программно-аппаратных платформ: Windows, Mac OS X, Linux. А разработанные игры можно запустить на 6-ти платформах: Windows, Mac OS X, iOS, Android, Linux и в web. Чтобы запустить игру на определенной платформе, нужна перекомпиляция движка. Прекрасная особенность T2D заключается в наличие всего исходного C++ кода движка, что позволяет изменить движок под требования своего проекта.

Посмотрим, из каких компонентов состоит Torque 2D, то есть речь пойдет о том, как устроен движок.

Torque 2D обладает высоко оптимизированным графическим пайплайном (конвейером), позволяющим выводить высококачественное изображение даже на морально устаревшем железе, реализация вывода графики в пайплайне выполнена с использованием кроссплатформенной графической библиотеки OpenGL. Этот факт, в особенности, позволяет портировать движок под разные операционные системы.

Torque 2D имеет внутренний скриптовый язык — Torque Script. Кроме самого движка в поставке Torque 2D имеется много примеров самых разнообразных игр. Все игры написаны на Torque Script и не требуют модификации дефолтной версии движка. Так как скриптовый код интерпретируется движком на лету, во время запуска игры, то это позволяет запускать стандартные игры на любой поддерживаемой платформе.

Для физических расчетов Torque 2D использует свободный физический движок с открытым исходным кодом Box 2D, который стал фактически стандартом де факто в физических расчетах упругих тел. Box 2D разработан Эрином Катто и был интегрирован в Torque 2D.

Для воспроизведения звука в Torque 2D используется открытая звуковая библиотека OpenAL. Она поддерживает широчайший перечень возможностей, вот некоторые из них: стереозвук, стриминг звуковых данных (постепенная подгрузка данных — по необходимости, а не единовременная загрузка всего файла, который может представлять большой размер), звуковой драйвер SFX-эффектов, поддержка панорамирования, объемного звука, доплеровского эффекта, кроме того, имеется поддержка многоканального звучания.

Выгодное преимущество движку Torque 2D дает мощная сетевая подсистема, используя которую можно создавать многопользовательские онлайновые игры со стабильным коннектом. К тому же в настоящее время многопользовательские игры набирают популярность в среде мобильных игроков. К примеру, игра в шахматы между двумя удаленными пользователями планшетов.

Из добавлений для движка Torque 2D стоит отметить поддержку двух нестандартных устройств ввода. Первое — это контроллер (геймпад) от игровой консоли Xbox 360. Его поддержка реализована в Windows. Следовательно, игры, предназначенные только для этой операционной системы, будут иметь поддержку данного геймпада. Разработчик этого дополнения активист сообщества движков Torque — программист Саймон Лав выложил это обновление в открытый доступ в самом начале января 2014-го года.

Второе поддерживаемое устройство — это Leap Motion (Скачкообразное движение). Данное устройство является новинкой на рынке IT. Для управления не используется никакое дополнительное оборудование, только ваши руки и пальцы. Небольшое USB-устройство — сенсор располагается перед экраном компьютера, рядом с которым происходит захват движения, посредством чего выполняются манипуляции с объектами на экране. Это очень любопытное устройство похожее на Kinect от Microsoft, только распознающее более детальные жесты. Теперь играми, разработанными с помощью Torque 2D, можно управлять с помощью этого устройства, то есть жестами рук. Его поддержка была добавлена в версию 3.0, которая вышла в свет в августе 2014-го года. Вдобавок, начиная с этой версии, T2D стал поддерживать скелетную анимацию, созданную с помощью программы Spine.

3 Жанры 2D игр

Этот раздел посвящен, собственно, тому, что мы с вами можем и будем разрабатывать в рамках двумерных игр. Этот предел очень широк и включает в себя огромное количество жанров. Рассмотрим некоторые из этих жанров; сразу хочу подчеркнуть, что этот список будет далеко не полным и его можно дополнять до бесконечности, поскольку, кроме «чистых» жанров есть такие, которые вступили с другими в симбиотическую связь, и им так же можно дать имена. Например, вам как разработчику игр может прийти идея создать какой-то новый жанр для своей игры, он может быть основан на каких-то «чистых» жанрах, а может — и нет. Кроме того, многие игры можно отнести к нескольким жанрам, то есть они могут обладать спецификой двух и более игровых жанров одновременно.

Выделим 6 «чистых» жанров.

— Аркады (шутеры, стрелялки, они же «экшены») с видом сбоку и сверху занимают наибольшую нишу среди двумерных игр. Под этот жанр можно отнести любой 2D боевик. Задача игрока: крушить врагов, пробираясь вперед к заранее поставленной цели, которая может заключаться в побеге, освобождении или захвате кого-то и многое другое. Такие игры требуют от играющего стремительной реакции и постоянно держат его в напряжении. Из наиболее известных — это Earthworm Jim, Super Meat Boy (рис. 1.1).

— Приключения (адвенчуры или квесты) тоже распространенный вариант двумерных игр. Такие игры обычно имеют глубокую сюжетную

Рис. 1.1. Super Meat Boy

линию, а в задачу игрока входит раскрытие какой-либо тайны. Поэтому такие игры пропитаны диалогами и загадками. Собственно, решением последних игрок занимается в процессе игры, из диалогов узнавая подробности об окружающей обстановке. В режиме решения задач — головоломок этот жанр похож на логические игры. Приключения позволяют расслабиться и неспешно наслаждаться историей, которую рассказыва-

Рис. 1.2. Сломанный меч 2. Диалог

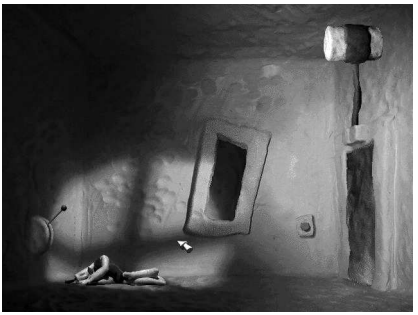

Рис. 1.3. Neverhood

ет игра. Для примера можно привести: Сломанный меч 1 — 3 (рис. 1.2), Братья пилоты, Machinarium, Neverhood (рис. 1.3).

— Гонки с видом сбоку или сверху (в двумерной проекции). Эти игры по обыкновению своему очень динамичны. Чаще всего их смысл заключается в победе в каких-то соревнования, а, чтобы добраться до финиша используется какое-либо транспортное средство. Это могут быть автомобили, машины, оснащенные оружием, катера, боевые вертолеты, танки и многое другое. Если соревнуются боевые машины, то игровой процесс становится более аркадным. Очень популярен вариант с видом сверху (например, Micro Machines), однако в последнее время стали появляться блестящие гонки с видом сбоку, в качестве примера можно привести TruckToy — сэмпл, входящий в число примеров движка Torque 2D.

Рис. 1.4. TruckToy

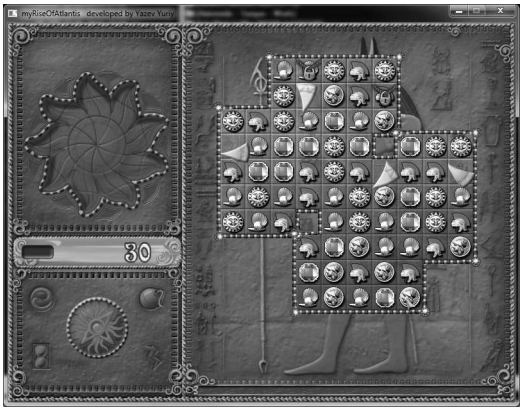

Рис. 1.5. myRiseOfAtlantis

— Логические игры представляют собой разные головоломки чаще всего набор головоломок, требующих от игрока проявление смекалки, внимания и умственных способностей: в отдельных играх надо что-то запомнить, построить логическую последовательность, найти отличия и многое другое. Молниеносная реакция в таких играх не требуется, однако учитывается скорость решения задач, в результате чего строится таблица участников, израсходовавших наименьшее количество времени. И как правило у игрока появляется мотивация оказаться в лидерах. Примеры: пасьянс, сапер, Rise of Atlantis (рис. 1.5) и многие другие.

Казуальные игры представляют собой целый подтип игр, однако в основной своей массе относятся к логическим играм, поэтому мы не относим их к «чистым жанрам».

— Лабиринты. Как и следует из названия жанра, в относящихся к нему играх надо помочь главному персонажу выбраться из запутанного лабиринта.

— Стратегии. В последнее время двумерные стратегии полностью вытеснены трехмерными. Но в былые времена, когда 3D-графика была непозволительной роскошью, миром правили двумерные стратегии. В играх этого жанра игрок управляет не отдельным индивидуумом, а целым взводом, армией, страной, вселенной. Поэтому такие игры требуют планирования, тактики, стратегии для победы игрока в некотором игровом событии. Стратегии делятся на множество типов, которые мы не будем рассматривать в контексте двумерных игр.

4 Элементы 2D-игр и их представление в Torque 2D

Давайте рассмотрим типы объектов, с которыми может взаимодействовать программист при программировании двумерных игр, вообще, и, использующий Torque 2D, в частности. Есть несколько типов объектов, производных от классов движка на C++. Чтобы их использовать для создания игровых объектов необходимо воспользоваться скриптовым языком Torque Script. Его мы в подробностях рассмотрим в следующей главе. А в текущей мы поверхностно рассмотрим разновидности доступных объектов. Обращаю ваше внимание, в этом разделе мы только поверхностно коснемся примитивов, более подробно мы рассмотрим их в главе 4, там же мы научимся их создавать и управлять ими.

В Torque 2D область, где происходит вся визуализация является сценой (объект класса Scene). В ней происходит не только вывод, но и создание, удаление, трансформация объектов и другое. Сцена, в свою очередь, создается внутри объекта класса SceneWindow, который принимает профиль для отображения, настраивает возможность взаимодействия. Также он настраивает позицию, размеры, угол поворота и масштабирование для точки обзора — камеры. Обратим внимание на компоненты — объекты, с которыми мы можем взаимодействовать и управлять.

Основной элемент двумерных игр — это спрайт. Отсюда повелось название целого ряда — спрайтовые игры; при этом спрайты используются не только в 2D-играх, на заре трехмерных игр и игр с изометрической проекцией (к ним относится ряд адвенчур и боевиков-бродилок) вместо представления трехмерных моделей использовались спрайты, повернутые под таким углом к точке обзора, что зрителю кажется: перед ним объемный предмет. Этот эффект называется биллбоард (billboard).

По определению: спрайт — это растровое изображение, перемещающееся (или находящееся на месте) по экрану. В более современном виде данное определение выглядит так: спрайт представляет прямоугольник с наложенной текстурой. Текстура может содержать прозрачные области (альфа-канал), благодаря чему спрайт вписывается в общую картину происходящего на экране, где могут присутствовать десятки и сотни взаимодействующих спрайтов.

1. Спрайт (Sprite) — основной класс для представления объектов в Torque 2D. Как и все остальные объекты, с которыми может взаимодействовать программист, Sprite — экспортируемый из движка тип. Он последовательно наследует два типа: SpriteBase и SceneObject — на верши-

не наследования. Из этого следует, что он принимает все возможности обоих классов, добавляя при этом свои: от первого он наследует свойства для визуализации изображения, а от второго — физические свойства, в том числе способность к нахождению и обработке столкновений.

2. Класс спрайтовая композиция (CompositeSprite) представляет собой набор спрайтов. Каждый спрайт этого набора может иметь любое расположение в сцене, независимый поворот, размер, при этом быть на любом слое глубины. Для объекта CompositeSprite имеется четыре опции размещения элементов:

1) No layout (без размещения) логическая позиция, указанная для объекта, становится физической для подобъекта;

2) Rectilinear layout (прямолинейное расположение) — указанная логическая позиция отображается в прямолинейную;

3) Isometric layout (изометрическое размещение) — логическая позиция отображается в изометрическую;

4) Custom layout (специальное размещение) — логическая позиция преобразуется в пользовательскую.

Спрайтовая композиция может быть использована не только для создания плиток (tile maps), она так же используется для создания сложных, состоящих из нескольких спрайтов персонажей. Кроме того, композиции удобно использовать в физических симуляциях, например, связать два или более спрайта для правдоподобно выглядящей жесткой связи. Вдобавок композиции прекрасно подходят для пакетного рендеринга, когда надо быстро произвести визуализацию большого количества спрайтов. В композиции можно легко и быстро изменить свойство у любого спрайта. CompositeSprite позволяет сделать отсечение для большой, выходящей за границы экрана, визуализируемой области.

3. Класс Scroller наследуется от того же родителя, что класс Sprite, следовательно, имеет те же наследуемые свойства, однако в отличие от последнего объекты класса Scroller позволяют прокручивать свое содержимое как по вертикали, так и по горизонтали.

4. Как и все прочие классы визуальных объектов — класс ShapeVector наследуется от класса SceneObject. Объекты класса ShapeVector отображают элементы векторной графики, такие как простые круги и полигоны. Они могут быть заштрихованы или закрашены. С помощью объекта этого класса можно вывести опорные точки векторной кривой.

5. Объекты класса ImageFont представляют собой надписи, отображаемые в сцене. ImageFont выводит надпись посредством заранее подготовленного битового шрифта. Битовый шрифт — это прямоугольный

растр, представляющий собой таблицу нарисованных символов. Как создать битовый шрифт и вывести надпись с помощью объекта ImageFont мы рассмотри чуть позднее, когда перейдем к экспериментам и созданию своих игр.

6. Объекты ParticlePlayer играют роль частиц в сцене. То есть один объект ParticlePlayer представляет эффект частиц, которые рождаются в «источнике», разлетаются в случайном направлении и через некоторое время умирают — уничтожаются.

7. Триггеры (объекты класса Trigger) — это не визуализируемые объекты, которые играют роль переключателей. Иными словами, эти объекты активируют заданное событие в тот момент, когда какой-то объект зашел/вышел в определенную — контролируемую триггером область.

Выше мы рассмотрели 7 типов основных объектов, входящих в Torque 2D. Кроме того, в Torque 2D есть объекты для воспроизведения звуков и анимации. Рассмотренные объекты используются наиболее часто. В дальнейшем по мере необходимости мы изучим другие объекты движка, а так же научимся их использовать в своих играх.

Кроме того, в Torque 2D есть богатая библиотека элементов пользовательского интерфейса. Многие из которых можно увидеть в примере Torque 2D Sandbox, нажав кнопку Show Tools в правом нижнем углу окна.

5 OK, я независимый разработчик, с чего начать?

Этот вопрос встает не только перед начинающими игроделами, а так же перед каждым, кто пробует себя в новой роли, пробует новое дело. Чтобы создать двумерную инди-игру требуется не так уж много усилий и средств, тем более, что главная часть игры — движок совершенно бесплатный.

Если речь идет о новой игре, то здесь в первую очередь необходимо придумать оригинальный сюжет, на основе которого разработать дизайн документ — логическую составляющую всей игры, важность которой трудно переоценить. Процесс создания дизайн документа называется геймдизайном, иначе это процесс проектирования. Ведь дизайн документ — это совокупность всех факторов, влияющих на удовлетворение (получение фана) игроком от игры. Поэтому дизайн документ составляют: интересный сюжет, игровые механика и динамика, предыстория, цель игры, персонажи, диалоги между ними и многое — многое другое. Из диздока во время игры получается геймплей (чувства, получаемые от игры), важность которого в 2D-игре огромна! В ней может не быть впечатляющей

графики и музыки, но если играть в нее будет интересно, игрок простит разработчику недостатки. Примеров тому масса! Но эта книга не посвящена придумыванию дизайн документа, она посвящена технической стороне игр, поэтому на теме геймдизайна мы останавливаться не будем. Тем не менее для каждой разрабатываемой игры мы будет писать диз-док, чтобы иметь представление о том, что мы хотим создать. Между тем существует много книг по игровому дизайну. Интересующихся я направляю к ним.

После того как дизайн документ придуман, а его идеи опробованы, и их интерес очевиден, то время переходить к наброскам, зарисовкам и иллюстрациям будущих предметов, персонажей и сцен. Этот этап очень важен, так как позволяет спроецировать то, что представляет автор игры у себя в голове на бумагу или экран дисплея. Для представления своих мыслей художнику, если игра разрабатывается в группе, геймдизайнеру надо подробно описать их в диз-доке. Затем на основе набросков создаются законченные рисунки, которые будут представлены в двумерной игре. После этого можно переходить к программированию — созданию законченного представления игры. Поскольку без кодовой базы изображения и звуки представляю обычный контент, никак не взаимодействующих объектов. Написанию кода, созданию взаимодействия между игровыми объектами, реализации игрового процесса будет посвящена остальная часть данной книги.

Вместе с созданием иллюстраций и программированием свою работу может выполнять звуковой инженер, занимаясь подготовкой музыки и звуков для разрабатываемой игры.

Между тем все перечисленные выше роли может вполне выполнить один человек — независимый разработчик игр, как раз на это и нацелена читаемая вами книга.

Издание своего программного продукта еще никогда не было настолько простым как сейчас. Разработчик создает на одной из площадок цифровой дистрибуции аккаунт, привязывает к нему свое мобильное устройство и/или идентификатор операционной системы, затем загружает в магазин свое приложение/игру, которое предварительно должно пройти все тесты, и размещенная игра начинает продаваться, принося доход своему создателю. Можно сделать свою игру бесплатной, тоже хорошо, приятно людям.

В настоящее время существует несколько площадок цифровой дистрибуции:

1) App Store — магазин компании Apple, ведущий продажу приложений и игр для устройств с операционными система OS X, iOS;

2) Google Play — магазин компании Google, распространяющий приложения и игры для операционной системы для мобильных устройств Android;

3) Amazon Appstore — площадка цифровой дистрибуции приложений для операционных систем Android, FireOS, BlackBerry, организованный на базе инфраструктуры платформ облачных сервисов Amazon Web Services компании Amazon.

4) Steam — магазин компании Valve, в нем продаются игры и приложения для всех распространенных платформ;

5) Leap Motion App Home — магазин, предназначенный для продажи игр и приложений, созданных для использования контроллера Leap Motion;

6) Windows Marketplace — магазин корпорации Microsoft, в котором продаются различные приложения и игры для операционной системы Windows 8.0 / 8.1 / 10;

7) Windows Phone Marketplace — площадка компании Microsoft, откуда можно купить игры и приложения, предназначенные для смартфонов с мобильной операционной системой Windows Phone 7.0 / 7.5 / 8.0 / 8.1 / 10.

Вы как разработчик игр, использующий движок Torque 2D, можете продавать свои игры в первых пяти из приведенного выше списка магазинах.

6 Инструменты

В этом разделе мы обсудим инструменты, которые будем использовать для разработки двумерных игр под различные платформы на движке Torque 2D.

Арт

После того как придуман сюжет и написан дизайн документ, можно переходить к созданию графических материалов, для которых уже созданы описания, наброски, зарисовки, одним словом — концепты. Для создания графических ассетов существует широкий набор инструментов — графических редакторов, и я уверен, что вы умеете работать в одном или нескольких из них. Среди коммерческих редакторов на рынке представлены: Adobe Photoshop (редактор растровой графики), Adobe Illustrator (редактор векторной графики), CorelDraw (векторный редактор), Corel Photo-Paint (растровый редактор), Corel Painter (растровая

графика), а так же многие другие, все не перечесть, я назвал лишь самые популярные. Тем не менее этот список ограничивается, когда мы ставим целью разработать коммерческую игру. В таком случае, если мы не можем позволить себе купить лицензию на редактор, придется обратиться к свободному программному обеспечению. К счастью, здесь помимо не представляющих интереса поделок присутствуют обладающие мощной функциональностью графические редакторы. Среди них можно выделить два: GIMP (GNU Image Manipulation Program) (http://www.gimp.org/) — мощный редактор растровой графики по функциональности схожий с лучшими коммерческими предложениями и InkSpace (http://www.inkscape.org/ru/) — графический редактор для векторной графики, заслуживший много положительных отзывов от дизайнеров и художников, считается лучшим свободным векторным редактором.

Прекрасным инструментом для создания фоновых графических поверхностей является редактор текстур. За место подобного инструмента можно использовать любой графический редактор, но редактор текстур позволяет быстро — по выбранному алгоритму сгенерировать текстуру. Из всего разнообразия подобных программ мне больше всего нравится редактор текстур Corel Texture. Раньше до десятой версии включительно он поставлялся в пакете программ CorelDraw, однако в новых версиях пакета его нет.

В связи с тем, что многие шрифты вовсе не бесплатны, напротив, охраняются авторскими правами, и за их использование надо отчислять автору процент от продаж своей игры, нам необходимо использовать собственные шрифты. Для их создания существует ряд программ, как платных, так и свободных. Из числа последних можно выделить два — это: FontForge (http://fontforge.org/) и BMFont (http://www.angelcode.com/products/bmfont/). Если первый представляет полноценный инструмент для редактирования имеющихся и создания новых шрифтов, то второй представляет из себя визуализатор карт символов на основе имеющихся шрифтов. Это необычайно мощный инструмент и в подавляющем большинстве случаев его функциональности достаточно.

Для создания карт/уровней можно использовать свободный инструмент Tiled (http://sourceforge.net/projects/tiled/). Он позволяет компоновать сцены из объектов.

Инструмент Texture Packer (http://www.codeandweb.com/texturepacker) прекрасно подходит для создания карт текстур (texture map). Он включает поддержку широкого набора форматов для импорта и экспорта. В результате Texture Packer создает таблицу изображений, общий размер ко-

торой меньше, чем сумма всех элементов. Что позволяет добиться меньшего объема игры (очень важный критерий для мобильных игр), а из этого следует повышение производительности последней. Torque 2D имеет поддержку Texture Packer.

В играх много используются анимированные образы. Чтобы создать анимацию какого-то объекта можно нарисовать каждое его состояние самостоятельно в графическом редакторе «от руки», но это будет очень долгий и не продуктивный процесс. Гораздо лучше воспользоваться специальными инструментами. Анимировать двумерный объект можно, например, с помощью редактора Flash, но есть способ лучше: воспользоваться специально предназначенной для 2D анимации утилитой Spine (http://esotericsoftware.com/) компании Esoteric Software. К сожалению, она не бесплатна, но такой мощный инструмент как Spine — стоит своих денег. Spine создает скелетную анимацию между связанными графическими изображениями, поэтому, по сути, скелетная анимация прекрасно подходит не только для анимирования персонажей, но и для движения и взаимодействия любых других объектов — изображений. Spine умеет экспортировать созданную анимацию в различные форматы: как в видео, так и в изображения — по кадрам. Вдобавок для Torque 2D создан плагин для непосредственной поддержки экспортируемых из Spine анимаций! В настоящее время этот плагин уже внедрен в исходный код движка, потому для загрузки анимаций, созданных в Spine, не нужно предпринимать дополнительных действий. Прежде чем покупать этот инструмент можно опробовать триальную версию, скачать которую рекомендуется с официального сайта. Триальная версия отличается от полной отсутствием в первой инструментов экспорта анимаций.

Еще одна очень полезная программа для создания спрайтовых анимаций — Spriter (http://www.brashmonkey.com/index.htm). Предназначение и способ управление данным инструментом такие же как у рассмотренного выше Spine. У Spriter так же есть демо- и про- версии. Кроме всего прочего имеется поддержка движка Torque 2D.

Для создания звукового сопровождения хорошо привлечь в группу разработки приятеля — звукоинженера. Но, поскольку это далеко не всегда возможно, можно самостоятельно подготавливать звуки и музыку. Существует огромный выбор различных звуковых редакторов: свободно распространяемых и платных. К последним относятся: Steinberg Cubase, Sony Sound Forge, Steinberg WaveLab, Adobe Audition и другие. Среди первых самым распространенным является Audacity (http://audacity.sourceforge.net/?lang=ru) — свободный, кроссплатфор-

менный звуковой редактор с открытым исходным кодом, обладающий впечатляющими возможностями по обработке музыки и созданию звуков.

Кроме того, необходимые элементы арта можно найти на просторах интернета. На некоторых сайтах энтузиасты выкладывают свои труды, чтобы показать их сообществу других артоделов и, чтобы ими воспользовались игроделы — энтузиасты. Тем самым художники, моделеры, музыканты рассказывают о себе, плюс создают себе хорошее портфолио. Тем более если игра, в которой использовался арт одного из членов сообщества станет успешной и популярной, то это большой повод для гордости, и на таких дизайнеров в первую очередь обращают внимание работодатели из игровых компаний. Например, вот один из таких вебсайтов: http://opengameart.org/, содержащий различный арт, который можно свободно использовать в своих проектах.

Кодинг

Эта книга целиком и полностью посвящена программированию — разработке таких специфических программных приложений как игры с помощью игровых (графических) движков семейства Torque. Поэтому основное внимание будет уделено именно кодингу, как любят говорить мои друзья из редакции журнала «Хакер».

Все обсужденные до сего момента инструменты (с некоторыми исключениями) существуют под все распространенные операционные системы, в том числе: Windows, Mac OS X, Linux. Однако перед тем как двигаться дальше мы должны избрать рабочую среду, в которой будем создавать игры. Я настоятельно рекомендую использовать для этой цели операционную систему Windows версии 7, 8.0/8.1 или 10 (я буду использовать первую из списка) по следующим, по меньшей мере двум причинам: все, без исключения, перечисленные в прошлом подразделе приложения имеют порты под Windows, для ОС Windows разработан очень мощный и удобный редактор скриптового кода — Torsion. А теперь все по порядку, какие инструменты мы будем использовать для кодинга.

Во-первых, для редактирования кода на C++ и компиляции движка мы будем использовать среду программирования Visual Studio. По умолчанию, последние на момент написания этих строчек версии движка Torque 2D компилируются в версиях 2013 и 2015. Можно использовать бесплатную Community версию студии, которую следует скачать с сайта Microsoft со страницы: https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs. Заметьте, для разработки необходимо ис-

пользовать редакцию «для Windows Desktop». Torque 2D будет успешно компилироваться без дополнительных настроек. Во-вторых, редактирование скриптового кода на языке Torque Script мы будем осуществлять в редакторе Torsion. Он был разработан компанией Sickhead Games (http://www.sickhead.com/), давним партнером компании Garage Games.

Вообще, редактировать код Torque Script можно в любом текстовом редакторе, после чего главное сохранить редактируемый файл. Если запустить после этого движок, то он автоматически интерпретирует измененные файлы. Сразу по окончании этой операции Torque запустит игру. Если изменения не содержат ошибок, они будут включены в выполнение игры, иначе новый код будет отвержен. При этом мы не сразу получим оповещение об ошибках, только после закрытия игры информация о них будет сохранена в файле console.log, находящимся в одной папке с игрой. Слишком долгий и неудобный процесс. Вот поэтому разработчики, использующие движки семейства Torque, работают в редакторе Torsion; свои игры мы тоже будем скриптовать с помощью Torsion. Потому что он позволяет, не покидая редактор, запустить игру на выполнение — через нажатие клавиши F5 и сразу получить сообщения о возможных ошибках. Во время редактирования кода Torsion подсвечивает ключевые элементы языка, в том числе при установке курсора на открывающую или закрывающую скобку парная будет выделена; нумерует строки и позволяет сворачивать именованные области кода. Torsion имеет обозреватель проекта, в котором можно выполнить любую операцию над файлами. Также в нем есть обозреватель кода (Code Browser), с его помощью можно осуществить навигацию по коду. В Torsion есть средство автоматической подстановки операторов (авто завершение), появляющееся после ввода точки после имени объекта (выбора пункта меню или нажатия комбинации клавиш), также известное как IntelliSense в Visual Studio. Присутствуют инструменты автоматического создания и удаления комментариев, приведение выделенного текста к написанию с заглавных и строчных букв, создания, удаления отступа с левого края. Torsion поддерживает работу с закладками, в том числе навигацию по ним. Но главная особенность редактора Torsion состоит в возможности отладки скриптового кода; как в профессиональных средах программирования здесь можно ставить точки останова, запустить игру на выполнение и при достижении одной из точек продолжить исполнение кода построчно. Это превосходный механизм, значительно упрощающий разработчикам их работу в части отладки игр, и вместе с окном слеже-

ния за значениями переменных превращает поиск багов в творческий процесс. Torsion сразу после запуска игры под встроенным отладчиком выводит сообщения движка в область вывода, находящейся внизу окна программы. Также туда выводятся сведения об ошибках компиляции и предупреждениях, обнаруживаемых в коде. Torsion может интерпретировать скрипты без запуска движка.

В предыдущем абзаце я описал только самые выдающиеся, на мой взгляд, средства редактора, однако он так же включает другие удобные механизмы, присущие современным редакторам кода. Мы рассмотрим их, когда будем использовать Torsion в своих проектах.

Между тем Torsion не бесплатный, но как я показал выше это незаменимый инструмент для скриптинга на языке Torque Script игровых приложений на движках семейства Torque. Покупку можно осуществить на сайте компании Garage Games в разделе Garage Games Store после регистрации аккаунта.

Разрабатывая игры на движке Torque 2D, нам будет необходимо описывать ассеты для их загрузки в игру, для этого я предлагаю использовать свободный текстовой редактор с подсветкой кода для огромного количества языков программирования Notepad++ (http://notepad-plus-plus.org/). Язык TAML в режиме XML работает по правилам последнего, и это, на самом деле, основной режим работы языка TAML. Поэтому, избрав в опциях Notepad++ поддержку синтаксиса XML, можно спокойно описывать ассеты, опираясь на подсветку ключевых элементов языка XML.

Это все инструменты, которые нам понадобятся при разработке игр. Список хороший, но еще лучше — у нас есть Torque!

7 Используемая версия движка Torque 2D

Так как Torque 2D свободное программное обеспечение с открытым исходным кодом, развитие проекта осуществляется сообществом 7 дней в неделю без выходных. То есть постоянно! Периодически выходят master-версии движка, другими словами, стабильные версии. Последняя master-версия на момент написания книги была Torque 2D 3.3. Однако автор рекомендует использовать версию движка из ветки development, поскольку в нее вносятся не только обновления, но так же исправления и улучшения. Одно критическое исправление было внесено в движок на следующий день после выхода master-версии, но уже не было включено в последнюю, поэтому оно так и осталось в ветке development.

Собственно, для разработки и тестирования представленных в книге игр автор использовал самую последнюю имеющуюся на GitHub версию движка. Поэтому, читателю так же рекомендуется использовать последнюю версию движка из ветки development. Используемая автором версия движка находится на GitHub по адресу: https://github.com/yurambo/Torque2D/tree/development.

Чтобы попасть в эту ветку (не говоря уже о master) код проходит тщательное тестирование и взвешенный подход, его важность для проекта обсуждается среди участников разработки, поэтому перед попаданием в репозиторий, код проходит тщательную проверку. Это дает уверенность в том, что даже в ветке development будет находиться максимально корректный и отлаженный код.

8 Необходимое аппаратное обеспечение

Для работы с материалом читаемой вами книги вовсе не обязателен компьютер верхней ценовой категории: для работы с двумерными играми подойдет нетбук со встроенным видеоадаптером. Тем не менее для удобной работы с движком и проектами на нем я все же посоветую мощный компьютер. К примеру, я работаю на компьютере со следующей конфигурацией: процессор — Intel Core i5 2,67 GHz, оперативная память — 8 GB, видеокарта — NVidia GeForce GTS 450 с 1024 GB встроенной памяти.

На рабочем компьютере у меня установлена операционная система Windows 7, в ней, в первую очередь, тестировались все игры. Кроме того, я проводил тестирование на компьютерах с операционными системами Windows 8.1, Windows 10 и Mac OS X. Дополнительно в главах, где это указано, для тестирования игр использовался планшет с операционной системой Android.

9 Заключение

Подведем итоги: о чем мы узнали из данной главы?

В начале главы мы обсудили текущее положение дел вокруг движка Torque 2D. Мы поговорили о его быстром развитии внутри сообщества Открытых исходников (Open Source). Обсудив полный список поддерживаемых платформ и операционных систем, мы перешли к обзору технической части движка, рассмотрели его компоненты и их возможности. Следующая затронутая нами тема была посвящена жанрам двумер-

ных игр, из нее мы увидели: какие игры имеет смысл разрабатывать для 2D. В следующем разделе речь шла об элементах двумерных игр и об их проекции на объекты движка Torque 2D. Таким образом, мы обсудили примитивы, из которых создаются самые разнообразные игры. Затем дается ответ на вопрос: с чего начать разработку игр? В этом разделе даются советы с чего начать разработку своей игры, как продолжить (на самом деле, очень важный вопрос), как закончить разработку, то есть как и где издать игру и получить за это вечнозеленых президентов. В шестом разделе мы обсудили инструменты, которые нам понадобятся для разработки игр на движке Torque 2D. Среди них присутствуют разные графические, аудио, анимационные редакторы и генераторы шрифтовых карт. В этом же разделе мы рассмотрели важные системы программирования, которые будем использовать в процессе разработки игр для написания скриптов и кода C++. И, наконец, мы рассмотрели какую версию движка Torque 2D будем использовать в своей работе с материалом книги.

В следующей главе мы изучим скриптовый язык движка Torque 2D, каким образом он связан с кодом движка на C++. Кроме того, мы изучим TAML — дополнительный язык движка Torque 2D для описания ассетов и загружаемых материалов. А так же разберемся в необходимых нам в работе инструментах. В общем, в следующей главе мы будем много программировать, поскольку это основное занятие в процессе разработки игр.

Глава 2. Скрипты и код

Оглавление

1 Torque Script

В этой главе мы займемся программированием, начнем изучение скриптового языка Torque Script, используемого в движках семейства Torque. Кроме того, что изначально Torque Script — проприетарный (защищенный авторским правом) скриптовый язык, разработанный Garage Games, а сейчас открытый — под лицензией MIT, он является высокоуровневым языком с динамической типизацией. Последнее свойство мы рассмотрим несколько позже, сейчас обратим внимание на другие особенности языка. Бесспорно, Torque Script является одним из самых значительных компонентов движков Torque 2D / 3D. Подобно большинству современных языков программирования, Torque Script унаследовал синтаксис от C/C++. Даже если вы не совсем уютно чувствуете себя с этими языками, далее я приведу достаточный материал для вспоминания основ программирования в рамках Torque Script. Действительно, синтаксис C/C++ оказался необычайно удобен, поэтому заслужил славу среди программистов разных категорий. Средства управления ходом выполнения программы те же самые, что в C/C++. Вместе с этим Torque Script гораздо проще своих прародителей и, соответственно, легче в изучении. Во многом это связано с динамической типизацией. То есть, по сути, в Torque Script отсутствует большой набор простых типов данных, такой как в C++ или Java. Между тем в Torque Script все же можно выделить два простых типа данных: это строка и число. Строка может быть произвольной длины, тогда как число может быть целым или с десятичной точкой. Следовательно, преобразование между двумя типами данных выполняется гораздо легче, чем в C/C++ с их богатым набором типов данных. Также к простым типам данных можно отнести логический тип, принимающий два значения: true и false, однако его нельзя причислить к самостоятельным типам, так как эти же значения можно представить 1 и 0, соответственно.

Используемые в коде Torque Script переменные в отличие от C/C++ объявлять заранее не нужно. Однако локальные и глобальные переменные различаются, так первые объявляются внутри функции с префиксом «знак процента» — %, например, %ball — будет видима в области определенной функции, внутри которой была объявлена. Локальная переменная может служить счетчиком или итерационной переменной цикла. Глобальные переменные могут быть объявлены в любом месте исходного кода, в том числе в любом файле, при этом они становятся так же доступны отовсюду. Они объявляются с префиксом «знак долла-

ра» — $, например, $wall — видна в любом месте исходного кода программы.

Это главные отличия Torque Script от C++, далее я расскажу об основных конструкциях первого языка, если вы хорошо знакомы с последним, можете пропустить данную часть книги, тем не менее в этих конструкциях есть некоторые отличия, поэтому советую хотя бы взглянуть на примеры кода. Между тем, прежде чем переходить к разбору языковых конструкций, необходимо поближе познакомиться с процессом работы Torque Script.

2 Выполнение Torque Script

В отличие от C++, который является компилируемым языком, Torque Script — интерпретируемый. Отсюда следует, что код на высокоуровневом языке преобразуется не сразу в двоичный, пригодный для выполнения непосредственно на процессоре, а в промежуточный, который впоследствии выполняется сначала в программной среде, и уже затем, пройдя преобразование в машинные команды, отдается процессору. Из этого следует, что интерпретируемые программы выполняются медленнее скомпилированных. Однако виртуальные машины, в которых выполняются интерпретируемые программы, высоко оптимизированы, поэтому разница практически незаметна. Виртуальной машиной для языка Torque Script является сам движок Torque! В момент его запуска он преобразует скрипты, написанные на языке Torque Script в бинарный код — файлы формата dso. После этого на стадии выполнения игры она читает и выполняет их. Если при следующем запуске какой-либо скрипт был изменен, то движок преобразует его и заменяет одноименный устаревший файл. Если в Торке отключена опция создания dso файлов (настройка по умолчанию), тогда движок читает файлы скриптов и выполняет их непосредственно в оперативной памяти. Отчетливо прослеживается, когда на этапе выполнения двоичные файлы читаются, переводятся в формат, пригодный для процессора, а затем выполняются на нем.

Вернемся к процессу выполнения Torque Script. Поскольку этот язык предназначен для описания игровой логики, геймплея и других сущностей, таких как: миссии, пользовательские интерфейсы и др., скорость его выполнения не играет значительной роли. В отличие от C++, на котором написаны критические по времени выполнения компоненты, такие как: визуализатор сцен, обработчик ввода и т.п. Выполнение кода

Torque Script примерно в 50 раз медленнее исполнения кода C++. Замечу, скриптовый код в других игровых движках выполняется приблизительно с такой же скоростью. То, что Torque Script работает медленнее, не должно быть препятствием в решении на нем задач, для которых он предназначен.

3 Для чего нужен Torque Script, если есть C++?

Из предыдущего раздела вытекает разумный вопрос: зачем тогда нужно писать игру на Torque Script, если он выполняется в разы медленнее? Я попытался дать краткий ответ в конце прошлого раздела, но пытливому уму игродела этого будет недостаточно.

Взглянем на индустрию программного обеспечения, не только на игры. В настоящее время в ней широко распространены так называемые управляемые языки: C#, Java и др., выполняемые посредством виртуальных машин, соответственно, .NET, JVM. А все из-за того, что такие программы разрабатывать легче, чем непосредственно в нативном языке, поскольку средства, входящие в управляемые языки, позволяют писать программы короче и быстрее (по времени разработки), так как в состав их входят специальные библиотеки для выполнения узких задач. При этом разработчику не нужно отвлекаться на низкоуровневые проблемы, такие как: выделение и очистка памяти. За всем проследит платформа — виртуальная машина. А программист может полностью сконцентрироваться на прикладной задаче.

То же самое мы имеем, программируя игры на движках семейства Torque. Можно сказать, что Torque представляет собой узкоспециализированную операционную систему! Не такую всеобъемлющую как Windows или Linux, однако во многом самостоятельную платформу для игр. Ведь в Торке есть все необходимые системные компоненты, присущие операционной системе: своя файловая система, диспетчер чтения/записи на жестком диске, менеджер памяти, менеджеры, управляющие видео и аудио потоками, сетевая подсистема, диспетчер пользовательского ввода, система графического пользовательского интерфейса, а так же компоненты для связи с конкретной операционной системой: Windows, Linux, Mac OS X и другие. Последние позволяют выполняться движку под разными операционными системами, взаимодействуя с кроссплатформенным ядром движка. В целом движок написан на C++, а компонент движка — язык Torque Script позволяет абстрагироваться от

этого системного уровня, выйти на уровень прикладных задач, писать непосредственно игровую логику.

Из-за этого, подобно тому, как прикладные программисты перешли на управляемые языки, разработчики игр перешли на скриптовые языки используемых ими движков. И это позволило во многом повысить скорость работы игроделов и улучшить создаваемые ими игры!

Прежде чем перейти непосредственно к кодированию на языке Torque Script нам необходима среда, в которой мы будем тестировать возможности этого языка.

4 Загрузка и установка Torque 2D

1 Загрузка движка

Для обозначенной в конце предыдущего раздела цели прекрасно подойдет сам движок Torque 2D, с помощью которого мы впоследствии будем разрабатывать игры, а пока пусть он послужит тестовым инструментом для выполнения скриптов на языке Torque Script.

Как мы выяснили в главе 1: лучше всего использовать самую последнюю версию движка, находящуюся в ветке development, поскольку в ней находятся самые последние исправления и обновления.

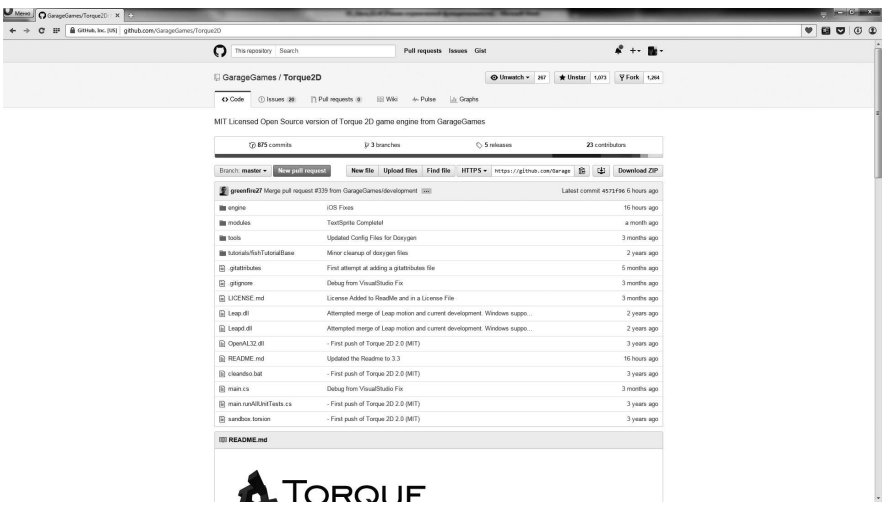

Рис. 2.1. Сайт github.com

Любую версию движка Torque 2D можно скачать с сайта github.com из раздела GarageGames: https://github.com/GarageGames/Torque2D/ (рис. 2.1). Перейдите на указанную страницу. Думаю, у вас уже есть аккаунт на этом сайте, если его нет, самое время зарегистрироваться и создать его. На момент написания этих строк последняя мастер-версия движка была 3.3.

Теперь нам надо сделать ответвление содержимого GarageGames/Torque2D в свой аккаунт. Для этого нажмем кнопку Fork в правом верхнем углу web-интерфейса (см. изображение выше). В итоге ветка кода вместе со стандартным артом движка Torque 2D будет скопирована в ваш аккаунт.

Теперь весь этот код и арт надо загрузить к себе на компьютер. Это можно сделать, нажав кнопку «Download ZIP» web-интерфейса. В результате на компьютер будет скачен zip-архив, содержащий копию текущей ветки, находящейся в вашем аккаунте. Однако с залогом на будущее, когда вы будете вносить модификации в движок, и вам надо будет загрузить сделанные обновления в свою ветку на сайт, сейчас вам надо скачать код с помощью Git-клиента, который организует проект на вашем компьютере и позволит работать с кодом как в системе контроля версиями (чем, собственно, и является Git), обновляя код или откатываясь на предыдущие версии.

Обзор работы с Git выходит за рамки данной книги, поэтому с помощью Git мы сделаем только минимум операций: загрузим движок себе

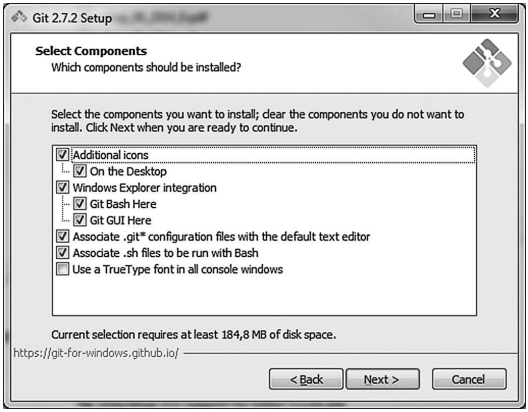

Рис. 2.2. Мастер установки Git (выбор устанавливаемых компонентов)

на компьютер. Чтобы получить Git-клиент, перейдите на сайт: http://msysgit.github.com/. На открывшейся странице нажмите кнопку Download. На диск вашего компьютера загрузится исполняемый файл, предназначенный для установки Git (рис. 2.2).

После установки будут доступны версия, работающая из командной строки и версия с графическим интерфейсом. Мы будем использовать первую, поскольку она удобнее.

Запустите файл Git Bash из папки Git главного меню. Откроется консоль. После первого запуска Git клиента надо идентифицироваться в системе. Для этого необходимо задать имя пользователя и e-mail. Эти сведения будут использоваться для авторства при коммитах — загрузке изменений в репозиторий на сайт. Выполните в консоли следующие команды:

для задания имени:

```
git config --global user.name "ваше имя"
```

для задания электронной почты:

```
git config --global user.email your@email
```

где, конечно, надо вставить ваш ник и адрес электронной почты.

Теперь все готово для клонирования репозитория с нашего аккаунта на сайте GitHub. Воспользуемся консолью: нам нужно использовать приложение git и ее команду clone, которая копирует репозиторий с указанного url в указанную папку, то есть принимает 2 параметра. В моем случае команда выглядит следующим образом:

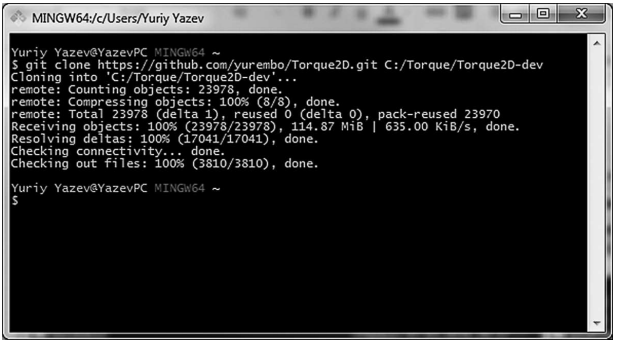

Рис. 2.3. Консоль Git Bash — загрузка движка на диск

```
git clone https://github.com/yurembo/Torque2D.git C:/
Torque/Torque2D-dev
```

Обратите внимание, так как Git кроссплатформенное приложение, в пути к папке символ обратный слеш \, принятый в Windows, заменен на слеш /, принятый в Unix.

В результате весь код, арт и командные файлы будут загружены с сайта GitHub из подготовленной ранее ветки.

Сейчас надо перейти в каталог, куда был загружен движок, как вы знаете, это делается с помощью команды cd:

```
cd C:/Torque/Torque2D-dev
```

Текущая работа проходила с веткой по умолчанию — master. Теперь, нам надо создать локальную ветку development, перейти на нее, переключить удаленную ветку с master на development и закачать новые данные. Все это можно выполнить с помощью одной команды:

```
git checkout -b development origin/development
```

В результате ее исполнения все данные, включенные в ветку development, будут докачены, а исходный репозиторий на локальном компьютере будет обновлен.

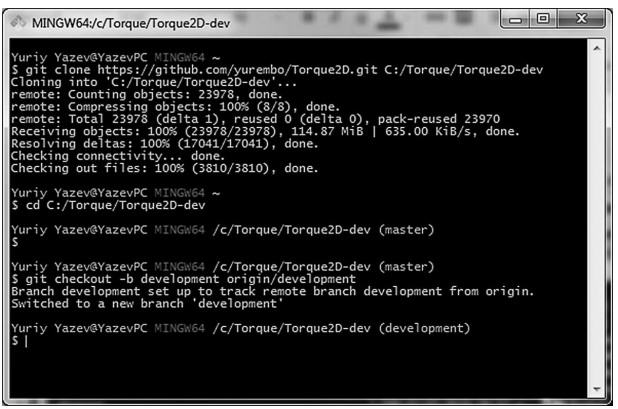

Рис. 2.4. Консоль GitBash с выполненными командами

2 Построение движка

В удобном для вас менеджере файлов откройте папку с движком — в моем случае:

```
c:\Torque\Torque2D-dev\
```

Далее, в зависимости от установленной у вас в системе версии Visual Studio, откройте подпапку с проектом, я использую Visual Studio 2013, поэтому у меня путь к проекту следующий:

```
engine\compilers\VisualStudio 2013\
```

В этом каталоге откройте файл Torque 2D.sln. Когда загрузится Visual Studio, можно сразу приступать к построению движка, никаких предварительных настроек он не требует. Нажмите F7 для начала процесса компиляции и линковки. Обратите внимание, какая версия в ниспадающем меню выбрана для построения: отладочная — Debug или релиз — Release.

Теперь, когда движок построен, можно начинать с ним работать. Перейдите в корневую папку с движком...

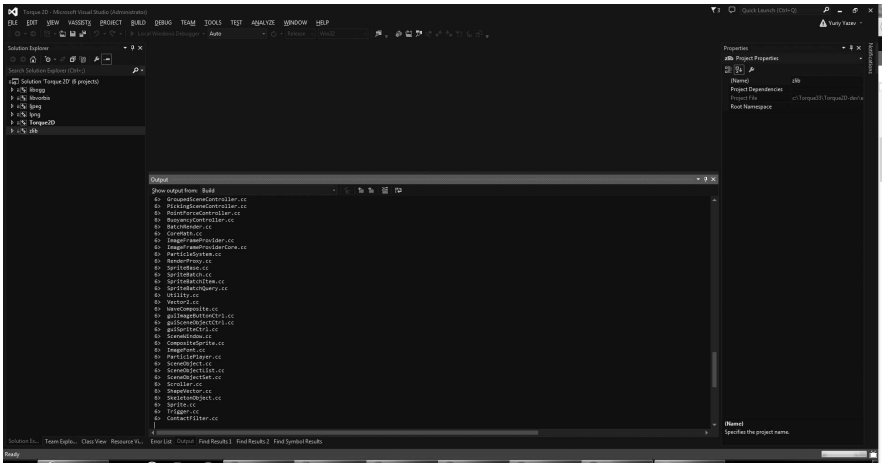

Рис. 2.5. Построение проекта Torque 2D

5 Основные языковые конструкции Torque Script

Для запуска движка надо просто выполнить файл Torque2D.exe. Вместе с движком запустится сэмпл TruckToy, оставим его рассмотрение до будущих времен, сейчас сконцентрируемся на Torque Script, для этого откроем консоль с помощью комбинации клавиш CTRL + ~ (рис. 2.6).

Большую часть окна занимает системный вывод движка, так сразу после запуска в нем видны сообщения об инициализации математических библиотек: стандартной библиотеки языка C, ассемблерные расширения и расширения FPU, MMX, SSE, так же вывод повествует об успешной инициализации библиотеки DirectInput и видеоподсистемы: библиотеки OpenGL, сообщается информация об используемом видеоадаптере.

Внизу окна размещена строка для ввода пользовательских команд на языке Torque Script. Не нарушая программерскую традицию, выведем сообщение: «Hello World!», для этого достаточно написать:

```
echo("Hello World!");
```

После чего нажать клавишу Enter для подтверждения выполнения операции. В результате внизу консольного вывода появятся две новые

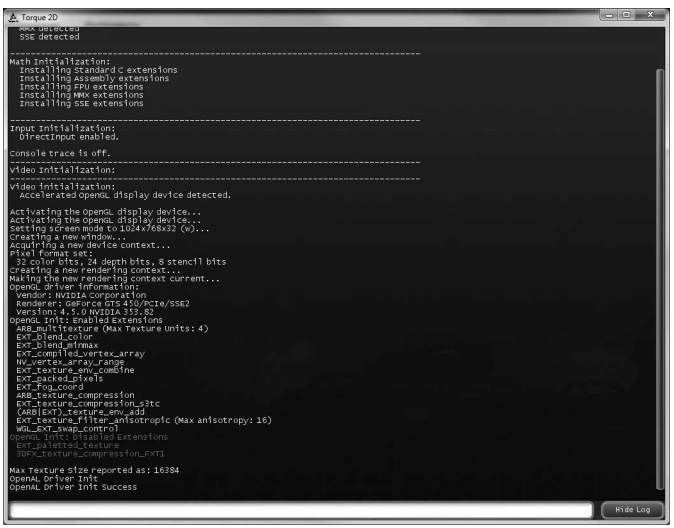

Рис. 2.6. Консоль Torque 2D

строчки: в первой после символов ==> будет продублирован ваш ввод, а во второй — результат выполнения введенной команды: «Hello World!».

Можете попробовать набирать другие выражения и посмотреть на результаты их выполнения, к примеру, команда: echo(5+5) выведет 10 и т.п. Системная функция echo предназначена для вывода сообщений на консоль.

Второй способ выполнения скриптов заключается в создании дополнительного файла с расширением «.cs», содержащим исполняемый код, подключения его к консоли и запуск метода main на выполнение. Создадим в папке с движком файл hello.cs и откроем его для редактирования в блокноте (Notepad.exe). Введем в него для примера следующий код:

```
function writeText()
{
    echo("Hello Torque!");
}
function main()
{
    writeText();
}
```

В этом простом коде движок, зайдя в функцию main, сразу же вызывает другую прикладную функцию writeText, которая, в свою очередь, посредством системной функции echo выводит в консоль текст. Чтобы Torque мог выполнить код из внешнего файла, его сначала надо подключить, введя команду: exec(«hello.cs») — точка с запятой на конце совсем не обязательны, Torque сам их добавит при вводе кода в консоль, но обязательны при вводе во внешние скрипты. После этого, чтобы запустить подключенный скрипт на выполнение, надо выполнить главную функцию — ввести main. В результате в консоль выведется строчка «Hello Torque!». Таким образом, можно выполнить любую функцию, но корректнее начать с main, как это делает операционная система с программами на языках C/C++.

Не отходя далеко от функций, сразу же рассмотрим их детали. По определению, функции служат для логического разделения программы на самостоятельные части, которые выполняют определенные задачи. Функции могут получать и возвращать аргументы. Чтобы функция принимала аргументы, надо после ее заголовка в круглых скобках указать один или более параметров: function writeText(%text). Затем в вызове функции надо передать параметр, указав значение: writeText(«Hello Torque!»); В итоге приведенный выше код примет такой вид:

```
function writeText(%text)
{
    echo(%text);
}
function main()
{
    writeText("Hello Torque!");
}
```

При этом вывод будет точно таким же, так как с помощью функции echo мы выводим параметр %text, который имеет значение «Hello Torque!».

Вернуть результат функция может с помощью ключевого слова return, которое указывается в конце функции после проведения каких-то вычислений — перед закрывающей фигурной скобкой. Функция не обязана возвращать значение. К примеру, return «Torque», возвращает слово «Torque». Код, который получает возвращаемое функцией значение, может обработать результат. В дальнейших проектах мы на практике увидим работу функций.

Torque Script поддерживает все распространенные операции, в том числе: арифметические, логические, побитовые (сдвиги), строковые и другие. Не вижу смысла приводить в данной книге таблицу и описание всех примитивных операций, а с более интересными мы встретимся в процессе разработки игр.

Массивы

Особенности использования переменных — ключевое отличие Torque Script от C/C++ мы рассмотрели в первом разделе данной главы, поговорим об использовании массивов. В Torque Script массивы относятся к типам данных. Как вам известно из компилируемых языков: массив представляет контейнер с заранее определенным размером, то есть количеством мест под элементы заранее определенного типа. В итоге размер массива равен произведению пространства необходимого для одного элемента на количество элементов. Для Torque Script определение массива будет иным: массив представляет контейнер. Это все. Как мы ранее обсуждали, в Torque Script не нужно предварительное объявление, поэтому ему совершенно не нужно знать размер массива. В массивах Torque Script вы можете хранить абсолютно любые объекты: строки, кнопки — элементы интерфейса, персонажей и мн. др. К примеру, мы можем создать глобальный массив для хранения имен игрушечных солдатиков, входящих во взвод:

```
$soldiers[0] = "John";
$soldiers[1] = "Mike";
$soldiers[2] = "Bill";
```

Обращаться к элементам массива можно по номеру — индексу. Так, чтобы получить имя второго солдатика, достаточно написать:

```
%name = soldiers[1];
```

Обратите внимание, счет элементов массива начинается с нуля. И теперь значение переменной %name равно «Mike».

Комментарии

Комментарии в Torque Script определяются точно так же, как в C++. То есть однострочный комментарий, продолжающийся до конца строки, начинается с двойного слеша: //, например:

```
%name = soldiers[1]; // теперь в переменной %name
// хранится значение "Mike"
```

Второй вид комментария может продолжаться на любое количество строк или занимать небольшую часть строки. Он начинается символами /* и продолжается до тех пор, пока не встретится обратная пара символов */. Например:

```
/*
...
какой длинный комментарий
...
*/
```

Условные операторы

Вспомним условные операторы. Как и в любом языке они очень важны, так как позволяют определить ход выполнения программы. Условные операторы передают управление определенному блоку кода в зависимости от условия. Результат условия определяется в условном выражении через операцию отношения и всегда равен одному из двух значений булевой логики: true или false. Судя по этому значению, происходит выбор выполняемой ветки (блока) кода. Имеется шесть операций отношения: == — равно, != — не равно, < — меньше, > — больше, <= — меньше или равно, >= — больше или равно. Один или оба операнда условного выражения могут

быть выражениями. Кроме того, условный оператор может передать управление при невыполнении условия, то есть в случае, если операция отношения вернет false. Для этого, как и в C, служит ключевое слово else, указываемое после блока кода, выполняемого при истинности условия. Приведем пример: пусть, если результат выражения больше 10, тогда условие истинно, и выполнение передается в первую ветвь, иначе, если результат выражения меньше или равен 10, тогда условие ложно, и выполняется вторая ветка кода. Добавьте в файл hello.cs следующую функцию:

```
function calc()
{
   if ((100 / 10) > 10) {
    %num = 400 * 4;
     echo("Expression is greater than 10, var %num = "
@ %num);
   } else {
     %num = 400 / 100;
     echo("Expression is less or equal 10, var %num = "
@ %num);
   }
}
```

Командой exec снова подключите файл hello.cs, тем самым он будет перекомпилирован и подключен. Если в результате компиляции нет ошибок, тогда выполните добавленную функцию calc(). Результат выполнения очевиден: выведется строка из второго блока кода (ветвь else), поскольку выражение возвращает ложное значение.

В приведенном примере кода активно используются операторы работы со строками. Работа со строками является одной из сильных сторон языка Torque Script. Мы рассмотрим работу со строками после условных операторов в текущем разделе, поскольку строки — это неотъемлемая часть языка.

Перечисленные выше операции отношения используются для сравнения числовых данных, однако для того, чтобы сравнить строки используются две дополнительных операции сравнения: для проверки идентичности строк используется отношение $=, например:

```
%pet = "dog";
if (%pet $= "dog") {
…
} else {
…
}
```

В результате управление будет передано на ветку, следующую за условием, так как сравнимые значения эквивалентны. Но если заменить операцию сравнения на !$= (не равно), тогда будет выполнена вторая ветка кода, поскольку при всех сохраненных с предыдущего примера условий, новый оператор отношения вернет false.

Обратите внимание: движок Torque не правильно работает с символами кириллицы.

В Torque Script присутствует еще один условный оператор — оператор ветвления — switch. В случае, когда в результате проверки какого-то условия события могут развиваться по многим путям — веткам кода, можно воспользоваться несколькими операторами if — else, но есть способ лучше: задействовать оператор — переключатель switch, который сравнивает полученное значение (или результат выражения) с одним или несколькими вариантами. Если в результате сравнения находится совпадение, то управление будет передано на соответствующую ветку кода. В случае, когда подходящее значение отсутствует, события могут развиваться по двум путям: происходит выход за пределы оператора switch и выполнение следующего за ним кода, а если внутри логической конструкции switch имеется ветвь default, то управление будет передано на нее. Посмотрим пример: в соответствии с номером переданного цвета (где 1 — красный, 2 — желтый, 3 — зеленый) автомобилю дается соответствующая команда:

```
%color = 3;
switch(%color) {
   case 1 : Stop();
   case 2 : Ready();
   case 3 : Run();
   default : TurnOff();
}
```

То есть автомобиль следует командам светофора, а если вдруг светофор передает другое значение, например, 0, тогда никакой свет не загорается, и автомобиль так продолжать не может, поэтому глушит мотор.

Тем не менее такой подход годиться только для чисел. Подобно сравнению строк в операторе if, где для их сравнения используются специальные операции отношения, в случае сравнения строковых значений с помощью ветвления, используется особенная форма оператора switch — switch$. Для примера в соответствии с именем солдатика поручим выполнять ему определенную команду:

```
%n = 0;
%name = $soldiers[%n];
switch$(%name) {
   case "John" : Attack();
   case "Mike" : Dig();
   case "Bill" : Defense();
   default : Sleep();
}
```

То есть John атакует, Mike роет яму, Bill защищает, а все остальные солдатики спят.

Обратите внимание: в отличие от C/C++ в Torque Script каждую ветку кода не надо завершать оператором break. Второе отличие — это возможность сравнивать строки, как мы видели выше, воспользовавшись модифицированным оператором switch$.

Еще один часто используемый в Torque Script условный оператор ?: подходит для простых условий и элементарных — однострочных действий. При его использовании сначала указывается условие, затем после знака ? записывается оператор, выполняемый в случае истинности условия, далее после знака : ставится оператор, который выполнится, если стоящее на первом месте условие не выполнится — вернет false. Рассмотрим такой надуманный пример, можете записать эту функцию в файл hello.cs, только не забудьте перед ее использованием перекомпилировать файл.

```
function select()
{
    %word = (1 > 0) ? "big" : "small";
    echo(%word);
}
```

В функции select переменной %word присваивается одна из двух строк, зависит от условия: если 1 больше 0 (я же сказал, что пример надуманный), то присваивается строка «big», иначе — «small». Затем значение переменной %word выводится в консоль. Примерно такой вид использования данного условного оператора используется чаще всего, то есть на основании условия, которым чаще всего служит проверка какого-либо объекта на равенство NULL, происходит присвоение переменной какого-то значения, то есть инициализация.

Следующая рассматриваемая нами конструкция будет циклы.

Циклы

Как в любом языке программирования в Torque Script циклы позволяют несколько раз выполнить повторяющиеся действия. В языке Torque Script имеется две конструкции циклов. Первым для рассмотрения будет циклический оператор for. Он используется в том случае, когда количество повторений операций известно заранее. Общая форма оператора for состоит из трех частей (подобно C/C++): инициализация, проверка, обновление:

```
for (инициализация; проверка; обновление)
{
   блок многократно выполняемых операторов
}
```

Здесь инициализация циклической переменной выполняется один раз — в начале работы цикла, затем происходит проверка данной переменной, если результат условия истинный, управление передается в блок многократно выполняемых операторов, после этого значение циклической переменной изменяется — в секции обновления, затем снова происходит проверка, и в случае успеха — выполнение операторов, и так до того, пока условие не будет отрицательным, за чем последует выход из цикла и выполнение следующего за ним кода. Для примера добавим в файл hello.cs функцию loopFor, содержащую цикл for:

```
function loopFor()
{
     for (%i = 1; %i <= 10; %i++)
     {
       echo(%i);
     }
}
```

Откомпилируйте и запустите данную функцию на выполнение. Результат ее работы вовсе не впечатляющий, однако он позволяет увидеть предназначение цикла for: в столбик будут выведены числа от 1 до 10, о чем указывается в коде: мы инициализируем переменную %i значением 1, проверяем, чтобы это значение не превышало 10, если это так, выполняем оператор echo — выводим на консоль значение переменной %i, после этого, увеличиваем ее значение на 1 (с помощью оператора инкремента), затем снова производим проверку, если она успешна — выполняем оператор, и так — по кругу.

Кроме цикла for в Torque Script имеется цикл while, которому не обязательно с самого начала знать количество итераций — выполняемых действий. Условие его завершения может не зависеть от количества выполняемых итераций. Общая форма его записи выглядит следующим образом:

```
while (условие)
{
   блок повторяемых операций
}
```

Как уже говорилось: условие может быть любым выражением, возвращающее булево значение. Если оно истинно, выполняется блок операторов, если ложно, происходит выход из циклического оператора, и выполнение кода продолжается за ним. Приведем пример: пусть имеется переменная, которая инициализируется при старте функции значением 100, затем оператором while проверяется, чтобы ее значение было равно или больше 100, то есть, первая проверка точно проходит, если условие выполняется, тогда в блоке для переменной назначается новое значение, которое мы получаем случайно из диапазона от 0 до 1000, используя функцию getRandom и выводим это значение в консоль, затем проверка происходит снова, так продолжается до тех пор, пока случайное значение не будет меньше 100, а это рано или поздно произойдет (рис. 2.7).

Рис. 2.7. Функция loopWhile

Чтобы осуществить описанные выше действия, в файл hello.cs добавим функцию loopWhile, ниже приведен ее код:

```
function loopWhile()
{
    %val = 100;
    while (%val >= 100)
    {
      %val = getRandom(1000);
      echo("val = " @ %val);
    }
}
```

Подключите файл к движку (тем самым перекомпилировав данный файл в байт-код) и запустите на выполнение добавленную функцию. Результат будет похожим, но не идентичным приведенному выше скриншоту, так как числовые значения выбираются случайным образом.

В этом разделе были рассмотрены базовые конструкции языка Torque Script, которые в той или иной форме имеются в любом языке программирования.

Этот раздел изначально задумывался не как подробное введение в программирование, поскольку я предполагаю, что читатель, как игродел, уже знаком с программированием и кодировал на каком-либо языке; поэтому этот раздел задумывался, как введение в язык Torque Script с приведением кратких, но достаточных комментариев по конструкциям языка. Поэтому я не стал делить раздел на подразделы, посвящая каждый какой-либо отдельной конструкции.

6 Специальные языковые конструкции

Строки

Большую роль в Torque Script играют строки. И не мудрено: TS фактически имеет два типа данных. И если с числами все понятно, то со строками дела обстоят иначе. Стандартные строки или последовательности символов записываются в двойных кавычках. В одинарных кавычках (апострофах) записываются так называемые «меченые» (tagged) строки. Последние используются для оптимизации трафика, то есть строка целиком передается только один раз, а при ее передаче в последующие разы, передается присвоенная ей числовая метка. Этот механизм используется для передачи неизменяемых строк, таких как имена функций.

Кроме того, что Torque Script поддерживает большинство функций работы со строками аналогичных стандартной библиотеки языка C, вдобавок он содержит некоторые уникальные (присущие только ему) конструкции, позволяющие в удобной форме писать код для работы со строками.

Итак, как мы уже видели выше для конкатенации (соединения) двух строк используется символ «коммерческого „а“» — @. Например:

```
%dogName = "bob" @ "ik";
```

В итоге в локальной переменной %dogName будет сохранена последовательность символов «bobik». Если нужно произвести конкатенацию двух строк, между которыми вставить пробел, можно сделать так:

```
%dogName = "bob" @ " " @ "ik";
```

Но гораздо лучше воспользоваться оператором SPC, который между соединяемыми строками вставляет пробел, пример:

```
%dogName = "bob" SPC "ik";
```

В обоих вышеприведенных случаях переменная %dogName будет иметь значение: «bob ik». Конкатенация — наиболее часто используемая строковая операция.

Еще одна часто используемая функция для работы со строками — это getSubStr, которая принимает строку для обработки, номер первого символа и количество символов (этот параметр необязательный), а возвращает часть строки (первый параметр), начинающуюся с номера символа, переданного во втором параметре и продолжающуюся до конца начальной строки (если третий параметр отсутствует) или то количество символов, которое указано в третьем параметре функции. К примеру, код:

```
%war = "WorldWar";
%war = getSubStr(%war, 0, 5);
echo(%war);
```

выведет на консоль слово World. С помощью функции strlen можно узнать длину последовательности символов.

Как я уже сказал: Torque Script содержит большое число функций для работы со строками: почти тот же набор, что язык C, поэтому пристально рассматривать в этой книге эти функции не имеет смысла. Однако когда нам потребуются строковые функции во время разработки игр, при необходимости мы рассмотрим их.

7 Сложные типы данных

Датаблоки

Датаблоки (или блоки данных) — это особая не модифицируемая (можно сказать: константная) языковая конструкция, состоящая из заранее определенных констант. Главным образом она используется для инициализации других, использующих ее объектов. Датаблоки — это серверные объекты, отсюда следует, что один и тот же датаблок на сервере и на всех клиентах имеет одинаковый дескриптор. Кроме того, после создания они не могут быть удалены. Блоки данных содержат в себе только атрибуты. Замечательной особенностью блоков данных является их способность наследоваться. Подобно классам в таком случае датаблок-потомок включает все атрибуты датаблока-предка, при этом он может добавить новые. Хотя наследование датаблоков используется весьма редко, в некоторых случаях оно может быть удобным решением. Приведу пример определения унаследованного датаблока:

```
datablock playerWomanData(DefaultWomanData :
PlayerData)
{
   renderFirstPerson = true;
   className = playerWomanArmor;
   shapeFile = "art/shapes/avatars/playerWoman/woman.
dts";
// etc…
}
```

Датаблок в скриптовом коде всегда является отражением одноименной структуры в коде движка на C++. playerWomanData — это структура в C++ коде, на основе которой будет создан датаблок DefaultWomanData, имеющиеся для которого значения будут унаследованы от датаблока PlayerData. Наследуемый датаблок должен быть создан так же от структуры playerWomanData. В приведенном выше примере внутри командных скобок присутствуют операторы инициализации атрибутов датаблока. В данном случае мы создаем блок данных для аватара: задаем режим отображения персонажа при виде от первого лица, указываем скриптовый класс, путь и файл, откуда надо загрузить модель, которая будет представлять аватар. Мы подробно рассмотрим блоки данных, когда будем использовать их в своих играх.

Классы

Хотя Torque Script не является истинно объектно-ориентированным языком, он имеет поддержку некоторых свойств, присущих ООП языкам. Таким образом, он имеет такую абстрактную структуру данных как класс, который, по сути, является важнейшим понятием в объектно-ориентированном программировании. Класс в Torque Script имеет близкие к ООП языкам возможности, но не все. В Torque Script можно создать класс, унаследовать его от класса, описанного в коде движка на C++, сделать базовым и/или унаследовать скриптовый класс (редко используется), объявить в классе не только переменные-члены, но и функции-члены. Тем не менее в классах Torque Script нет модификаторов доступа (public, private, protected), предназначенных для определения областей доступа к переменным и методам класса. По определению все члены класса являются открытыми (public). В отличие от C++ в Torque Script нет множественного наследования, его так же нет во многих других ООП языках, поэтому — не страшно. Также здесь нет виртуальных функций, но на самом деле в этом языке они не нужны. Кроме того, отсутствуют ключевые слова для определения конструкторов и деструкторов, но никто не запрещает за место них использовать функции с другими именами. Кроме того, как мы увидим в дальнейшем, функции конструкторов выполняют другие конструкции. Если капнуть глубже в теорию ООП, то в Torque Script нет возможности перегрузки функций/операторов, а так же нет конструкций для перехвата и обработки исключений, последнее должно быть реализовано на уровне C++ кода. Все отсутствующие возможности языка с лихвой окупаются его гибкостью.

Базовым классом для всех объектов в Torque Script является SimObject. Он содержит в себе фундаментальные свойства и методы, используемые всеми потомками, этот список очень обширен. Он включает в себя возможность использования «умных указателей», конструкторов/деструкторов, наследует способность отвечать на события добавления (onAdd()), удаления (onRemove()) и многое другое.

Как было отмечено выше: наследование классов в движке Torque чаще всего реализуется в коде на C++ и почти не используется в Torque Script. На самом деле, так и есть. Между тем в Torque Script есть ключевое слово superClass, позволяющее задать для класса базовый класс. Реальный пример описания элемента интерфейса, имеющего базовый класс:

```
%guiContent = new GuiControl(GuiMusicPlayer) {
```

```
   isContainer = "1";
   Profile = "GuiWindowProfile";
   HorizSizing = "right";
   VertSizing = "bottom";
   position = "0 0";
   Extent = "1024 768";
   MinExtent = "8 2";
   canSave = "1";
   Visible = "1";
   tooltipprofile = "GuiToolTipProfile";
   hovertime = "1000";
   canSaveDynamicFields = "1";
   superClass = "GuiMusicPlayerClass";
// etc…
}
```

В примере выше для скриптового класса GuiMusicPlayer с помощью ключевого слова superClass задается базовый класс GuiMusicPlayerClass.

Объекты

Объекты — это экземпляры классов, то есть каждый объект содержит в себе ровно столько (ни больше, ни меньше), сколько данных и методов для работы с ним описано в объявлении класса, от которого производится этот объект. Уверен, это вы знаете и без меня. Перейдем к Torque Script. В этом языке все внутриигровые сущности — объекты. После создания объекта ему присваивается уникальный числовой идентификатор, называемый дескриптором, он однозначно определяет созданный объект, поскольку на одной игровой площадке (уровне) не может быть двух объектов с одинаковыми дескрипторами.

Существование любого объекта начинается с его создания. Рассмотрим этот важный процесс. Подобно созданию экземпляра класса в C++, в Torque Script для этого используется оператор new. Однако в отличие от C++ (или Java), где управление передается в конструктор класса, Torque Script действует иначе, продолжая выполнение со следующей строки, где обычно находится блок кода, в котором находятся операторы инициализации переменных-членов класса. В других языках (к примеру, C#, C++11) подобные конструкции, находящиеся на одном уровне с вызываемым кодом, называются анонимными функциями и/или лямбда выражениями.

Для примера создадим типичный спрайт — движущийся растр (объект класса Sprite) для двумерной игры:

```
%socket = new Sprite(hole)
   {
        Position = %X SPC %Y;
        Size = "13.7 13.7";
        SceneLayer = 30;
        Image = "ToyAssets:socket1";
        class = "Socket";
   };
   %socket.setBodyType(static);
   %socket.setUseInputEvents(true);
   %socket.block = false;
   %socket.magic = "";
```

Рассмотрим этот кусок кода. Выделив оператором new память для экземпляра класса Sprite, в качестве параметра для конструктора, мы передаем имя (hole) для объекта. По этому имени мы сможем обратиться к нему в будущем. Затем в блоке кода происходит инициализация тех переменных-членов класса, которые были объявлены в классе Sprite в коде на C++. Это такие члены: позиция (Position) задается двумя координатами — X и Y, размер (Size) определяет размер растра, SceneLayer (слой) подобно продвинутым графическим редакторам позволяет определить номер слоя, на котором будет находиться данный объект, Image (изображение) — задает устанавливаемое изображение — так называемый ассет (в будущем работу с ассетами мы рассмотрим более подробно), class — задает для данного экземпляра класса Sprite дополнительный подкласс, описанный в скриптовом коде, что позволяет приобрести данному объекту особые свойства и методы — отличные от базового класса из движка. Как можно увидеть: некоторые свойства (другими словами: переменные-члены, такие как: block, magic) объекта инициализируются за пределами блока кода оператора new. Эти переменные-члены не объявлены в классе Sprite, поэтому не могут быть проинициализованы в блоке оператора new. Torque Script резервирует память под эти свойства при первой встрече, то есть во время интерпретации файла с кодом. Другие свойства объекта устанавливаются с помощью методов, в данном случае: setBodyType, setUseInputEvents. В принципе, так как в Torque Script нет областей видимости, значения для задаваемых ими переменных можно определить с помощью рассмотренных выше способов доступа к свойствам. С помощью метода setBodyType (соответствующее свойство BodyType) задается физические свойства объекта, то есть как он будет вести себя в сцене — игровом мире. А метод setUseInputEvents (соответствующее свойство UseInputEvents) включает

или выключается способность объекта отвечать на нажатие на нем курсором мыши (на компьютере) и/или прикосновение пальцем на сенсорном экране планшета или смартфона.

Заметьте, хотя мы задали для объекта глобальное имя — hole, также, дескриптор созданного объекта мы сохранили в локальной переменной %socket, которую можем использовать до выхода из текущей функции.

Когда вызывается метод объекта класса, например:

```
function Socket::lightSockets(%this, %socket,
%switchOn)
{
   // сделать что-то полезное
}
```

то первым параметром идет ключевое слово %this, которое представляет собой дескриптор того объекта, над которым выполняется этот метод. Необходимо явно указывать это ключевое слово. С другой стороны, при вызове глобальной функции параметр %this не передается:

```
function checkMagicSocket(%socketnum, %crystal)
{
   // сделать что-то полезное
}
```

Это все, что я хотел сказать о функциях и методах.

Любой объект можно удалить, применив к нему метод delete().

Контейнеры

Подобно контейнерам, входящим в стандартные языковые библиотеки (такие как STL для C++) в Torque Script тоже есть контейнеры, только в нем они не отделены от языка. В Torque Script имеется три контейнера: строки, SimSet, SimGroup. И этого достаточно для любых целей. Коротко говоря, контейнеры представляют собой динамические массивы с автоматическим управлением памятью, способные хранить объекты любых типов. То есть при добавлении объекта в контейнер, он автоматически расширяется, выделяя дополнительную память, а при удалении объекта из него сужается — очищает память, отведенную под удаленный объект.

По сути, строки (Strings) предназначены для хранения массивов строк, но в то же время их можно использовать для хранения других типов объектов, например, в Torque Script строками представлены вектора и координаты:

```
%coords = "10 12";
```

Обратите внимание на символ пробела между элементами, он служит разделителем за место точки или запятой. Также в строках можно хранить списки имен объектов:

```
%nameList = "obj1 obj2 obj3 obj4";
```

где элементы так же разделяются символом пробела. Затем с помощью функции getWordCount, которая в качестве параметра получает объект-строку, можно получить количество слов, разделенных пробелом, содержащихся в этой строке.

```
%count = getWordCount(%nameList); // %count равен
// четырем
```

Чтобы получить слово по индексу, надо применить к строке функцию getWord, принимающую два параметра: объект-строку и индекс (счет с нуля).

```
%word = getWord(%nameList, 1); // %word равен "obj2"
```

А, чтобы заменить определенное индексом слово в строке, надо воспользоваться функцией setWord, она принимает три параметра: объект-строку, индекс заменяемого слова, новое слово, которое займет место удаляемого.

```
setWord(%nameList, 2, "object"); // %nameList будет
//иметь следующий вид: "obj1 obj2 object obj4"
```

Существует широкий список функций для работы со строками, мы рассмотрели наиболее часто используемые.

Однако для хранения объектов есть способ лучше: воспользоваться другим видом контейнера, таким как: SimSet. Он представляет собой подобие вектора из стандартной библиотеки языка C++, но при этом обладает более широкими возможностями и средствами. Он хранит коллекцию объектов класса SimObject, исходя из того, что это базовый класс для любого объекта (см. выше), следовательно, SimSet может хранить любые объекты. Также SimSet может хранить другие коллекции SimSet, получается своего рода многомерный массив.

Класс SimSet содержит ряд удобных методов для работы с коллекциями, рассмотрим наиболее часто используемые из них, остальные

разберем при использовании их в своих играх. Метод add(SimObject obj) позволяет добавить новый элемент в конец коллекции, тем самым автоматически расширяя ее; clear() — очищает коллекцию; getCount() — возвращает количество содержащихся в коллекции элементов; getObject(int index) — в качестве параметра получает индекс (порядковый номер) элемента в коллекции и возвращает данный объект; getObjectIndex(SimObject obj) — передает объект и, если таковой имеется в коллекции, то возвращается его числовой индекс; isMember(SimObject obj) — проверяет переданный в параметре объект на принадлежность коллекции; getRandom() — возвращает из коллекции случайный объект; remove (SimObject obj) — удаляет из коллекции переданный в параметре объект; listObject() — печатает в консоли список всех содержащихся в коллекции объектов. Кроме того, класс SimSet содержит два события: onObjectAdded (SimObject obj), onObjectRemoved (SimObject obj), генерируемых, соответственно, при добавлении и удалении объекта в/из коллекции. С помощью метода delete(), как и любой другой объект Torque Script, коллекцию можно удалить.

Важно обратить внимание: когда удаляется объект класса SimSet, содержащиеся в нем до удаления объекты не уничтожаются! Это очень важно, так как после удаления коллекции можно потерять ссылки на эти «висячие» объекты, следовательно, образуется утечка памяти, поэтому перед удалением коллекции необходимо уничтожить все входящие в нее объекты, в другом случае, если вам после удаления коллекции нужны содержащиеся в ней элементы, тогда скопировать эти объекты.

Рассмотрим пример использования объекта — коллекции класса SimSet. Снова пример из реальной игры. Возьмем функцию createCrystals:

```
function createCrystals(%numSoc, %X, %Y)
{
   new SimSet(sock @ %numSoc);
   mySock.add(sock @ %numSoc);
   for (%i = 0; %i < 4; %i++) {
  %cry = new Sprite(crys @ %i)
  {
       Size = "4 4";
       SceneLayer = 30;
     Position = %X SPC %Y;
       Image = "ToyAssets:SomeImage";
       class = "Crystal";
  };
```

```
   // дополнительный код
   (sock @ %numSoc).add(%cry);
    }
}
```

Для примера я немного упростил ее, но она по-прежнему осталась интересной. Итак, ей передаются три параметра: номер и пара координат по двум осям. В начале ее выполнения создается коллекция, затем она помещается в другую коллекцию. После выполняется цикл, в котором создаются четыре спрайта, скриптового подкласса Crystal. Когда один спрайт будет создан, в рамках цикла этот спрайт помещается в созданный в первой строчке контейнер. Если эта функция будет вызвана один или более раз, то в результате имеем «русскую матрешку» — в общем контейнере (созданный за пределами данной функции) будут находиться другие коллекции, каждая из которых будет содержать по четыре спрайта.

Еще один класс коллекции, имеющийся в Torque Script — это SimGroup. SimGroup является более строгой коллекцией по сравнению с SimSet. Это проявляется в следующих деталях: в каждый момент времени определенный объект может быть включен только в один контейнер SimGroup, таким образом, коллекция SimGroup автоматически управляет принадлежностью объекта, при его добавлении в один из контейнеров означает его удаление из другого — предшествующего владельца.

Коллекции (контейнеры) — очень мощное средство языка Torque Script, позволяющее хранить объекты любых типов. В практических примерах мы будем много использовать коллекции, тем самым мы досконально разберемся с ними.

8 TAML

Коротко говоря, TAML — это язык описания ассетов. TAML — это аббревиатура от Torque Application Mark-Up Language, что на русский язык можно перевести как прикладной язык разметки для Torque. Этот язык служит для описания файлов различных форматов в пригодный для загрузки движком вид. С его помощью описываются одиночные картинки, последовательности изображений — анимации, звуковое сопровождение — аудио файлы, разнообразные эффекты, среди чего частицы и прочее. Все это — одним словом ассеты.

Кроме описания ассетов с помощью TAML можно описывать и загружать игровые объекты, например, спрайты. Вдобавок Torque 2D, используя TAML, способен записывать файлы. В этих файлах, таким образом,

удобно хранить данные об игровых объектах. С помощью TAML можно работать с тремя типами файлов: XML, JSON, Binary.

При описании в первом формате, как следует из его названия, используется язык XML. Хотя при его использовании файлы получаются раздутыми, многословными, содержащие излишнюю информацию, тем не менее они легко редактируются в любом текстовом редакторе. Это существенный плюс.

Во втором формате используется язык JSON. Это более компактный язык, поэтому описание с его помощью сильно отличаются от тех же описаний на языке XML, так же пригоден для редактирования в текстовом редакторе.

В третьем случаем данные записываются и считываются в бинарные файлы, в этом случае их сложно редактировать, однако файлы получаются небольшого размера, вдобавок при их создании можно воспользоваться функцией компрессии.

Для записи и чтения файлов язык Taml использует, соответственно, методы TamlWrite и TamlRead. Вот как выглядит запись в файл данных о спрайте:

```
%obj = new Sprite();
TamlWrite( %obj, "stuff.taml" );
```

В первой строчке создается спрайт, во второй он записывается в файл. Так как мы не указали ни одно значение свойств, в результате вывод будет выглядеть следующим образом:

```
<Sprite/>
```

По умолчанию для записи/чтения файлов используется формат XML.

Чтобы прочитать файл — восстановить из записи программный объект, надо написать:

```
%obj = TamlRead( "stuff.taml" );
```

В принципе, этих двух функций достаточно для работы с файлами на языке Taml, поскольку они выполняют достаточный перечень операций.

Тем не менее часто бывают случаи, когда необходимо настроить дополнительные опции для записи/чтения файлов. В развернутом виде использование функциональности языка Taml требует создания объекта Taml, вышеприведенный пример записи и чтения данных о спрайте в файл будет выглядеть следующим образом:

```
%obj = new Sprite();//создаем спрайт
%taml = new Taml();//создаем объект Taml
%taml.write( %obj, "stuff.taml" );//записываем данные
// о спрайте в файл
%readObj = %taml.read( "stuff.taml" );//читаем данные
// о спрайте из файла
%taml.delete();//удаляем объект Taml
```

Обратите внимание на вторую и последнюю строчки, в них осуществляется создание и удаление объекта Taml.

Используя этот развернутый формат, после создания объекта Taml, но перед записью/чтение можно определить для этого объекта формат:

```
// устанавливаем xml формат
%taml.Format = Xml;
//устанавливаем JSON формат
%taml.Format = Json;
//устанавливаем двоичный формат
%taml.Format = Binary;
```

Кроме того, формат файла можно определить с помощью второго параметра функции чтения/записи по расширению файла:

```
// записываем в формате XML
%taml.write( %obj, "stuff.taml" );
// записываем в формате JSON
%taml.write( %obj, "stuff.json" );
// записываем в двоичном формате
%taml.write( %obj, "stuff.baml" );
```

Эти стандартные расширения: taml для XML, json для JSON, baml для Binary можно изменить с помощью свойств объекта Taml:

```
%taml.AutoFormatXmlExtension = "xml";
%taml.AutoFormatBinaryExtension = "bin";
```

Чтобы включить/выключить компрессию для двоичных файлов в языке Taml используется метод setBinaryCompression объекта Taml, в качестве параметра ему передается логический флаг — true/false для, соответственно, включения и отключения компрессии:

```
//включить компрессию
%taml.setBinaryCompression(true);
```

Чтобы узнать включена компрессия или нет, надо воспользоваться методом getBinaryCompression объекта класса Taml:

```
%flag = %taml.getBinaryCompression();
```

Вдобавок с помощью дополнительных параметров можно определить формат вывода для файлов в функции Taml: TamlWrite. Примеры:

```
//записываем данные в файл в двоичном формате без
//компрессии
TamlWrite( %obj, "stuff.dat", binary, false );
//переключаем формат вывода на XML
TamlWrite( %obj, "stuff.txt", xml );
```

В своих играх мы будем часто использовать язык Taml, особенно для описания ассетов.

9 Связь с движком — код C++

Игры — это системное программное обеспечение из-за наличия в их движке кода, выполняющего, дополняющего, а иногда заменяющего собой механизмы операционной системы. Например, это такие механизмы, как: управление оперативной памятью, кэширование данных в виртуальной памяти, оптимизированный графический вывод, различные сетевые подсистемы, службы и протоколы, системы искусственного интеллекта, поддержка ряда специфических устройств ввода, а так же многое-многое другое.

В целом Torque 2D, как и другое системное программное обеспечение, большей частью написан на C++. Кроме того, его исходник состоит из ассемблерных вставок для выполнения особенно критичных по времени операций, например, таких как математические. Изрядную долю движка составляет код поддержки определенных моделей процессоров, таким образом, код специально оптимизирован под разные процессоры, тем самым он включает встроенные средства SSE для поддержки оптимизации Intel процессоров, а так же средства 3D NOW — оптимизации процессоров компании AMD.

Безусловно, главная прелесть движков Torque — это полный исходный код. Таким образом, разработчик может заточить движок под требования определенного проекта, но, в то же время это довольно сложное занятие, требующее глубокие познания не только языка C++, стандартной библиотеки и модели ООП, но так же операционной системы и работы аппаратных средств.

В движке — на языке C++ можно написать какой-либо механизм, например, связь с базой данных, а потом вывести управляющие функции в

скриптовый код, чтобы вызывать их из него. Для этого в движках семейства Torque есть прослойка, связывающая код движка на C++ со скриптовым кодом. Эта прослойка представляет собой специальные обертки для функций и переменных языка C++, после перекомпиляции движка, обернутые переменные и функции становятся видны в скриптовом коде. Взглянем на этот слой.

Главную роль здесь играет пространство имен Con. Оно содержит различные функции, которые позволяют использовать переменные и функции движка в скриптовом коде и наоборот.

Общие переменные

Со скриптами и кодом движка можно сделать общую глобальную переменную. Отмечу, это не очень хорошее решение, так как переменную лучше оставить недоступной, а ее значение изменять/получать с помощью методов. Однако чтобы открыть в скриптовый код глобальную переменную из движка, нужно воспользоваться функцией addVariable пространства имен Con. Также можно открыть статический член класса. В следующем примере глобальная переменная движка year отражается на глобальную переменную $year скриптового кода:

```
Con::addVariable("$year", TypeS32, &year);
```

У функции addVariable 3 параметра: в первом — указывается имя переменной в скриптовом коде, на которую отобразится переменная движка, во втором — тип данных отображаемой переменной, последним параметром указана сама отображаемая переменная, описанная в коде на C++. Обратим внимание на второй параметр — тип данных, он не обычен. TypeS32 — это тип-обертка для скриптового кода для типа S32 движка. В движках семейства Torque используются свои обозначения типов данных, которые отображаются на стандартные. Таким образом, S32 — это стандартный тип int, представляющий целочисленные значения. Так сделано по той причине, что Torque является кроссплатформенным движком, и на другой программно-аппаратной платформе (отличной от Wintel) какой-либо тип данных может иметь другое значение.

В коде движка Torque определение нового типа на основе имеющегося осуществляется с помощью макроса ConsoleType, в рассмотренном случае вызов данного макроса выглядит следующим образом:

```
ConsoleType( int, TypeS32, S32 )
```

После этого можно указать, какие типы взаимозаменяемы, то есть равнозначны между собой без преобразования:

```
ImplementConsoleTypeCasters(TypeS32, S32)
```

Мы более подробно рассмотрим Торковские типы данных, когда будем использовать их в разработке, между тем так же можно использовать стандартные обозначения типов данных.

Также можно скрыть глобальную переменную движка из скриптового кода, для этого используется функция Con::removeVariable. Передав ей в качестве параметра ранее зарегистрированное имя глобальной переменной, оно будет не доступно из скриптов:

```
Con::removeVariable("$year");
```

В коде на C++ можно использовать переменную, объявленную в скриптах. Получение и установка значения переменной — это две операции, выполняемые двумя функциями. В Torque имеется четыре типа таких операций:

1) для получения и установки логического значения глобальной переменной в скриптах (getBoolVariable, setBoolVariable),

2) для получения и установки числового значения (getFloatVariable, setFloatVariable),

3) для получения и установки значения локальной скриптовой переменной — зависит от контекста выполнения скриптового кода (getLocalVariable, setLocalVariable),

4) для получения и установки значения строкового типа (getVariable, setVariable).

Последняя функция — getVariable возвращает ссылку на строку. Рассмотрим пример:

```
F32 year = Con::getFloatVariable("$year");
year += 10;
Con::setFloatVariable("$year", year);
```

В приведенном примере мы получаем значение ранее зарегистрированной скриптовой переменной, увеличиваем его на 10, затем последним действием возвращаем увеличенное значение в скриптовую переменную.

Общие функции и методы

Подобно переменным в движке Torque можно создать функцию и/или метод класса, который потом будет доступен из скриптового кода. Сначала мы рассмотрим старый, но до сих пор рабочий способ создания консольных функций и методов. Создание консольных функций в движке Torque во многом аналогично описанию обычных функций на языке C/C++. Так функция на языке C++, которую можно вызвать из Torque Script создается с помощь макроса ConsoleFunction. Он имеет следующий заголовок:

```
# define ConsoleFunction(name,returnType,minArgs,
maxArgs,usage1)
```

Он принимает следующие пять параметров:

1) name — строка, содержащая имя скриптовой функции,

2) returnType — тип возвращаемого ей значения,

3) minArgs — целочисленное значение, представляющее собой минимальное количество параметров описываемой функции,

4) maxArgs — максимальное количество параметров, получаемых описываемой функцией,

5) usage1 — строка — справка, печатаемая в консоли при неудачном завершении выполнения данной функции.

Приведу пример гипотетической консольной функции, вычисляющей сумму двух переданных ей параметров:

```
ConsoleFunction(sumParams, //имя функции
                F32, //тип возвращаемого значения,
//число с плавающей запятой (float)
       3, //минимальное необходимое количество
//параметров
       4, //максимальное количество параметров
       "func adds two float numbers")
{
     F32 res = dAtof(argv[1]) + dAtof(argv[2]);
     return res;
}
```

В теле функции сначала происходит приведение типов данных второго (счет с нуля) и третьего параметров из строк к числам с плавающей точкой, а затем их сложение и помещение результата в переменную res. После этого значение этой переменной возвращается вызвавшему эту функцию скриптовому коду. Приведение типов данных осуществляется

с помощью торковской обертки стандартной функции atof — dAtof. Лидирующая буква d происходит от Dynamix, названия компании — разработчика оригинального движка для Tribes 2, из которого родился Torque (см. Введение). На первом месте в массиве параметров argv под индексом 0 находится строка — имя вызываемой функции.

Макрос для объявления консольного метода имеет следующий заголовок:

```
# define ConsoleMethod(className,name,returnType,
minArgs,maxArgs,usage1)
```

Как видно, кроме пяти аналогичных параметров макроса ConsoleFunction, этот — имеет дополнительный параметр — className (на первом месте). В качестве данного параметра передается имя класса в движке, для определенного экземпляра данного класса можно будет вызвать описываемый метод из скриптового кода. Гипотетический пример:

```
ConsoleMethod(Ninja, Fight, bool, 3, 3, "fight is
failed")
{
     if (object->kick(argv[2]) == true)
return true;
else return false;
}
```

При вызове метода Fight фактически передается только один параметр — строка — имя врага. Однако макрос ConsoleMethod в отличие от ConsoleFunction предоставляет два параметра — элемента массива argv по умолчанию: под индексом 0 так же, как в прошлом случае находится имя метода, а под индексом 1 — идентификатор объекта, для которого вызван метод, этот объект в теле метода представлен ключевым словом object. В примере выше, если метод kick объекта класса Ninja возвращает истину (пинок завершился успешно), тогда консольный метод Fight тоже возвращает истину (драка завершилась успешно для нашего объекта — ниндзя), иначе — возвращается false.

Вызов консольного метода для объекта из скриптового кода осуществляется через точечную нотацию, например:

```
if (%ninja.Fight("bob") == true) echo("Win");
```

Кроме консольных методов, работающих для экземпляров класса, Torque поддерживает консольные статические методы, которые могут

быть исполнены в отсутствии экземпляров класса, то есть посредством самого класса. Макрос создания консольного статического метода имеет похожий вид:

```
#  define ConsoleStaticMethod(className,name,returnType
,minArgs,maxArgs,usage1)
```

То есть список параметров такой же, как у макроса ConsoleMethod. Реальный пример консольного статического метода из движка Torque 2D:

```
ConsoleStaticMethod(GameConnection,
getServerConnection, S32, 2, 2, "() Get the server
connection if any.")
{
   if(GameConnection::getConnectionToServer())
     return GameConnection::getConnectionToServer()-
>getId();
   else
   {
Con::errorf("GameConnection::getServerConnection - no
connection available.");
  return -1;
    }
 }
```

Необходимо обратить внимание: в массиве argv консольной статической функции — ConsoleStaticMethod так же, как в ConsoleMethod два элемента по умолчанию — это: имя метода и имя класса.

Мы рассмотрели старый, хорошо зарекомендовавший себя способ определения консольных функций и методов, тем не менее, когда в мае 2011-го года была выпущена версия 1.1 движка Torque 3D, компания GarageGames изменила вид определения консольных функций и методов, однако для обратной совместимости оставила поддержку старого способа. Я очень и очень сомневаюсь, что когда-нибудь поддержка старого способа объявления будет удалена из движка, поскольку имеется огромное количество старых проектов, использующих старый подход, который поменять на новый лад будет очень затруднительно. Однако при описание новых консольных методов GarageGames рекомендует использовать новый подход. На самом деле, при использовании он намного удобнее предшественника. Выделим изменения: во-первых, заголовок нового макроса имеет вид:

```
#define DefineEngineFunction( name, returnType, args,
defaultArgs, usage )
```

Такое же количество параметров, однако были заменены третий и четвертый параметры. Если раньше на их месте находились minArgs и maxArgs определяющие, соответственно, минимальное и максимальное количество параметров — элементов массива argv, то теперь их места занимают args и defaultArgs. Они выполняют другие функции по сравнению со старыми параметрами. Первый из них (args) представляет собой список параметров через запятую с указанием типов данных в том виде и порядке, в котором этот список будет представлен в определении функции. Параметр defaultArgs так же через запятую представляет список значений по умолчанию для списка параметров, определенного на прошлом шаге. Объявление и определение консольной функции, используя новый способ, выглядит следующим образом:

```
DefineEngineFunction( screenShot, void, ( const char
*file, const char *format, U32 tileCount, F32
tileOverlap ),( 1, 0 ),
"Takes a screenshot with optional tiling to produce
huge screenshots" )
{
   if ( !gScreenShot )
   {
  Con::errorf( "Screenshot module not initialized by
device" );
  return;
   }
   Torque::Path ssPath( file );
   Torque::FS::CreatePath( ssPath );
   Torque::FS::FileSystemRef fs = Torque::FS::GetFileS
ystem(ssPath);
   Torque::Path newPath = fs->mapTo(ssPath);
   gScreenShot->setPending(   newPath.getFullPath(),
                              dStricmp( format, "JPEG"
) == 0,
                              tileCount,
                              tileOverlap );
  }
```

Это реальная функция, взятая из исходника движка Torque. Списки параметров и значений по умолчанию для них указаны в круглых скобках через запятую. Обратите внимание на заголовок функции, а потом на ее тело: параметры используются поименно, как они описываются в заголовке, точно так же они используются в теле. Никакого массива argv[]. С новым описанием не надо запоминать: какой параметр на каком месте стоит, просто использовать именованные параметры, это

главное давно ожидаемое улучшение намного облегчившее жизнь программистам, использующим Torque.

В новом макросе для описания консольного метода, как этого и следовало ожидать, по сравнению с макросом описания консольной функции добавился дополнительный параметр — имя класса:

```
#define DefineEngineMethod( className, name, returnType,
args, defaultArgs, usage )
```

Он играет такую же роль, как в макросе ConsoleMethod, поэтому не буду повторяться с его описанием. Кроме того, объект, над которым выполняется метод, тоже представлен ключевым словом object.

Новый макрос определения консольного статического метода имеет вид:

```
#define DefineEngineStaticMethod( className, name,
returnType, args, defaultArgs, usage )
```

Он выполняет все соответствующие ему действия.

Используя Torque, программист так же может из движка вызвать функцию, описанную в скриптах. Для этого служат функции execute и executef. Их различие состоит в том, что первая принимает фиксированное количество параметров, тогда как вторая — произвольное. Объявление первой функции:

```
const char *execute(S32 argc, const char* argv[]);
```

Как видно, она принимает два параметра: целочисленное число, определяющее количество передаваемых в скриптовую функцию параметров, и массив строк, представляющий список параметров. Для примера вызовем из движка консольную функцию echo, передав ей строковый параметр:

```
function printLine()
{
     static const char* params[2];
     params[0] = "echo";
     params[1] = "Torque is awesome game engine!";
     Con::execute(2, params);
}
```

В C++ функции printLine мы объявляем статический массив строк, состоящий из двух элементов, затем заполняем его: в первый элемент по-

мещаем имя вызываемой функции — echo, во второй — строку параметр, который будет передан функции, указанной в первом элементе. После этого происходит вызов скриптового кода через функцию движка execute, которой передается число — количество параметров — 2 и выше определенный массив, состоящий из двух элементов. В результате Torque выведет в консоль: Torque is awesome game engine!

Функция executef в качестве первого параметра так же принимает число — количество передаваемых параметров, однако последующие параметры не надо помещать в массив, они просто указываются через запятую, их число определяется значением первого параметра. Выше приведенный пример в этом случае примет такой вид:

```
function printLine()
{
   Con::execute(2, "echo", "Torque is awesome game
engine!");
}
```

Результат будет такой же.

Помимо этого, функции execute() и executef() могут использоваться для вызова методов экземпляра класса. В таком случае на первое место в списке параметров помещается ссылка на объект класса SimObject (и/или его потомков), для которого надо вызвать скриптовый метод, а в конец списка — флаг thisCallOnly, в итоге, сигнатура функции принимает такой вид:

```
const char *execute(SimObject *object, S32 argc, const
char *argv[], bool thisCallOnly = false);
```

Второй и третий параметры имеют прежние значения, последний параметр — логический флаг определяет: нужно ли вызывать скриптовую функцию рекурсивно. В следующем примере мы напишем функцию на C++, которая подготавливает массив параметров и вызывает метод в скриптовом коде, передавая ей этот массив вместе с объектом, над которым требуется выполнить этот метод:

```
void Fight(SimObject *ninja)
{
     static const char params[2];
     params[0] = "kick";
     params[1] = "bob";
     Con::execute(ninja, 2, params);
}
```

В функции Fight движка создается статический массив из двух элементов, который заполняется двумя строками: именем скриптового метода, который надо вызвать и значением передаваемого ему параметра. В заключении скриптовый метод вызывается посредством функции execute, принимающей: ссылку на объект, для которого надо вызвать метод, число передаваемых в метод параметров и массив, содержащий эти параметры.

Функция executef для вызова скриптового метода похожа на одноименную функцию вызова скриптовой функции, в то же время она дополнилась параметром — ссылкой на объект класса SimObject, присутствующим в ранее рассмотренной функции execute для вызова скриптового метода. При этом в ней отсутствует флаг thisCallOnly. Вот ее сигнатура:

```
const char *executef(SimObject *, S32 argc, ...);
```

Приведем переделанный ранее приведенный пример:

```
void Fight(SimObject *ninja)
{
     Con::execute(ninja, 2, "kick", "bob");
}
```

Результат тот же.

Чтобы передать параметр в скриптовый метод с помощью функций execute/executef он должен быть приведен к строковому типу. Но далеко не все передаваемые аргументы изначально строкового типа. Для преобразования значений из разных типов данных к строковому типу в движках семейства Torque есть специально приготовленные функции. Несмотря на это, для преобразования можно использовать стандартные функции языка C/C++, но торковские — более удобные. Итак, чтобы преобразовать число с плавающей точкой к строковому типу надо воспользоваться функцией:

```
char *getFloatArg(F64 arg)
```

Для того, чтобы преобразовать целочисленное число в строку можно воспользоваться функцией:

```
char *getIntArg(S32 arg)
```

А, чтобы привести булево значение к строке, использовать функцию:

```
char* getBoolArg(bool arg)
```

10 Заключение

В заканчивающейся главе мы изучали разнообразные особенности программирования игрового программного обеспечения, используя при этом движки семейства Torque.

В начале главы мы в общих чертах взглянули на торковский скриптовый язык Torque Script, разобрали его отличия от C++. Затем мы подготовили тестовую площадку для кодирования на Torque Script, скачав и установив движок Torque 2D. В следующем разделе мы погрузились в базовые конструкции языка Torque Script, тем самым рассмотрели: переменные, функции, массивы, комментарии, условные операторы, циклы. После этого мы изучили специальные языковые конструкции, присущие Torque Script, такие как: строки, сложные типы данных: классы, датаблоки, объекты, коллекции. В восьмом разделе мы изучили дополнительный язык движка Torque 2D, предназначенный для описания ассетов и работы с файловыми объектами TAML. Мы увидели, как с его помощью можно сохранять и восстанавливать состояния игровых объектов в дисковый файл. Разобравшись с TAML, мы, затем перешли на системный уровень — в код C++ и подробно рассмотрели способы, предоставляемые движком Torque для связи языков Torque Script и C++. В этом разделе мы рассмотрели возможность создания общих переменных, функций, методов объекта и класса. Мы узнали, как из скриптового кода обращаться в код движка и наоборот. Кроме того, мы рассмотрели, как старый, так и новый способы создания консольных функций и методов.

В этой главе мы, действительно, уделили программированию много внимания. А в следующей, не снижая темпа, мы еще глубже погрузимся в игрострой и разработаем наши первые экспериментальные игры на движке Torque 2D!

Глава 3. Эксперименты

Оглавление

1 Содержимое директории движка Torque 2D версии 3.3

Эти строки писались в то время, когда последней версией движка была 3.3, хочу подчеркнуть это, поскольку от версии к версии содержимое папки может изменяться. На следующей иллюстрации (рис. 3.1) показано содержимое каталога Torque 2D версии 3.3.

Рассмотрим содержимое каталога. Открываем папку Torque2D-3.3-win. В подпапке engine находятся еще пять подкаталогов. Каталог bin содержит три приложения, каждое из которых имеет свою папку. Приложение bison предназначено для автоматического создания синтаксических анализаторов по данному описанию грамматики. Bison — консольное приложение, разработанное в рамках проекта GNU, относится к «Открытым исходникам» и портировано под все распространенные операционные системы.

Приложение flex предназначено для похожих целей, то есть является генератором лексических анализаторов.

NASM является свободным ассемблером (компилятором с языка ассемблер) для 32-х битных процессоров (x86) для Unix-подобных систем. Как вы помните: Torque 2D кроссплатформенный движок.

Имя	Дата изменения	Тип	Размер
engine	26.12.2015 22:57	Папка с файлами	
modules	26.12.2015 17:50	Папка с файлами	
tools	26.12.2015 17:50	Папка с файлами	
tutorials	26.12.2015 17:50	Папка с файлами	
.gitattributes	26.12.2015 17:50	Текстовый докум...	1 КБ
.gitignore	26.12.2015 17:50	Текстовый докум...	2 КБ
cleandso.bat	26.12.2015 17:50	Пакетный файл ...	1 КБ
console.log	05.03.2016 23:58	Текстовый докум...	2 КБ
Leap.dll	26.12.2015 17:50	Расширение при...	1 198 КБ
Leapd.dll	26.12.2015 17:50	Расширение при...	1 931 КБ
LICENSE.md	26.12.2015 17:50	Visual Studio Code	2 КБ
main.cs	26.12.2015 17:50	Файл "CS"	3 КБ
main.runAllUnitTests.cs	26.12.2015 17:50	Файл "CS"	2 КБ
OpenAL32.dll	26.12.2015 17:50	Расширение при...	156 КБ
preferences.cs	05.03.2016 23:58	Файл "CS"	1 КБ
README.md	26.12.2015 17:50	Visual Studio Code	5 КБ
sandbox.torsion	26.12.2015 17:50	Torsion Project File	1 КБ
sandbox.torsion.exports	13.01.2016 3:19	Файл "EXPORTS"	1 КБ
sandbox.torsion.opt	15.01.2016 19:51	Файл "OPT"	1 КБ
Torque2D.exe	27.12.2015 15:38	Приложение	3 201 КБ
Torque2D_DEBUG.exe	26.12.2015 23:00	Приложение	7 893 КБ
Torque2D_DEBUG.ilk	26.12.2015 23:00	Incremental Linke...	16 626 КБ
unicows.dll	26.12.2015 17:50	Расширение при...	253 КБ

Рис. 3.1. Содержимое каталога Torque 2D 3.3

В подпапке Compilers каталога engine находятся решения для сред разработки: Visual Studio 2013 / 2015, Xcode и Xcode проект для iOS; в папке android находится проект для Eclipse, в android-studio для Android Studio, в папке emscripten — для транслятора Emscripten, который предназначен для трансляции кода с языков C/C++ в JavaScript, в каталоге Make — командные файлы для построения движка под Linux. Эти решения предназначены для открытия исходного кода движка в соответствующей системе программирования под определенной операционной системой.

В каталоге lib находятся вспомогательные свободные библиотеки, по сути, не являющиеся частью движка. Библиотека ljpeg (расшифровывается как lossless jpeg — jpeg без потерь), служит для работы с графическими файлами формата jpeg.

Библиотека lpng (расшифровывается как light png) предназначена для загрузки/обработки/сохранения изображений формата png. В отличие от предыдущей данный формат (и библиотека) поддерживает альфа-канал (канал прозрачности) изображений.

OpenAL является открытым кроссплатформенным интерфейсом программирования приложений обработки аудиоданных. Основной особенностью данной библиотеки является работа со звуком в трехмерном пространстве с использованием эффектов EAX. Изначально разработа-

на в ныне несуществующей компании Loki Software, а теперь поддерживается Creative Technologies.

В каталоге libogg находятся файлы для реализации открытого формата мультимедиа контейнера Ogg.

В папке libvorbis размещены исходные и заголовочные файлы на C для поддержки свободного формата сжатия звука с потерями vorbis, который по качеству превосходит mp3.

В директории freetype находится библиотека для растеризации шрифтов в мобильной операционной системе Android.

Каталог LeapSDK содержит заголовки для работы с устройством Leap Motion.

Последняя подпапка этого каталога содержит zlib — свободную кроссплатформенную библиотеку сжатия и распаковки данных. zlib использует алгоритм без потерь данных Deflate. У читателя может возникнуть закономерный вопрос: зачем Торку работать с архивами? Движки семейства Torque умеют работать с артом, запакованным в архивы zip, плюс для невозможности открытия файлов сторонними средствами позволяют устанавливать для архивов пароли.

В подкаталоге source папки engine находится весь исходный код движка.

Следующая папка корневого каталога движка — modules содержит скрипты и арт для всех демонстрационных игр.

В каталоге tools находятся дополнительные инструменты и плагины. Так в подпапке doxygen находится утилита для генерации документации на основе исходных текстов.

В папке TexturePacker находится плагин для данной утилиты для поддержки формата пригодного для Torque 2D.

В каталоге Visual Studio Visualizer находится визуализатор отладчика для Visual Studio. Он работает следующим образом: в исходном коде движка Torque 2D есть несколько типов данных, которые отображаются не корректным образом в выводе отладчика. Применив этот адд-он, все встает на свои места. Для разных версий Visual Studio (2013 и 2015) существуют разные решения.

Последний имеющийся плагин представляет собой инструкцию по настройке утилиты Zwoptex для поддержки движка Torque 2D. Эта утилита предназначена для создания таблиц текстур. Поскольку она предназначена исключительно для операционной системы Mac OS X, для нас она не представляет большого интереса.

Папка Torque2D.app, находящаяся в корневом каталоге движка представляет собой исполняемую программу в операционной системе Mac OS X.

Далее в каталоге tutorials находится туториал по работе с движком.

Находящийся в этой же папке командный bat файл cleandso служит для удаления файлов с двоичным кодом dso, которые получились в результате компилирования скриптового кода.

Файл main.cs играет роль инициализатора приложения. После запуска исполняемого файла движок читает файл main.cs, последний указывает, какие действия надо выполнить в начале работы приложения: откуда загружать дополнительные скрипты, включить или нет режим логирования, профайлер и др. Можно сказать: этот файл выполняет такую же роль, как функция main в программе на C/C++.

На файл main.runAllUnitTests.cs передается управление при запуске модульных тестов из движка.

В файле preferences.cs содержатся начальные параметры для видео, аудио подсистем, размер окна приложения, настройки запуска приложения и прочее.

OpenAL32.dll — динамическая библиотека, содержащая скомпилированный код для работы со звуком.

Leap.dll и Leapd.dll — динамические модули, соответственно, релизной и отладочной версий библиотеки для работы с контроллером Leap Motion.

Sandbox.torsion — файл проекта для среды программирования Torsion.

Torque2D.exe — собственно, исполняемый файл движка для операционной системы Windows.

unicows.dll (так же известна, как «Microsoft Layer for Unicode») — библиотека, позволяющая использовать один и тот же Unicode код для разных версий Windows (имеется в виду: NT, поддерживающая Unicode и 9x, не поддерживающая Unicode). Это осуществляется путем преобразования вызовов системных API между Windows NT (Unicode) и Windows 9x (не Unicode). В итоге, данная библиотека нужна для поддержки работы движка даже в старых версиях Windows! В последнем обновлении движка этот файл был удален.

Из таких файлов состоит содержимое папки движка Torque 2D.

Рис. 3.2. Пример TruckToy из SandBox

2 Обзор SandBox

SandBox (русс. песочница) — это демонстрационное приложение, включающее 34 небольшие показательные игры. Если выполнить файл Torque2D.exe, то запустится SandBox, а выполняться по умолчанию будет игра TruckToy (рис. 3.2).

Чтобы управлять движением автомобиля, надо щелкать мышью, соответственно, справа от нее для движения вправо, и слева — для движения влево. Но главная особенность этой демо-игры заключается не в этом, а в обработке физического взаимодействия между машиной и трассой, которая состоит из спусков, подъемов и преград, а поскольку физика вычисляется близко к реалистичному поведению, то ненароком можно перевернуть транспортное средство.

Чтобы открыть список всех демо-игр, надо нажать кнопку Show Tools, которая находится в правом нижнем углу. Поверх запущенной игры появятся элементы управления меню (рис. 3.3).

Из списка в левой части окна можно выбрать любую из доступных игр. В центре находится список параметров. Для каждой выбранной игры этот список включает свои параметры — для настройки определенных возможностей игры. Список, находящийся в правой части окна

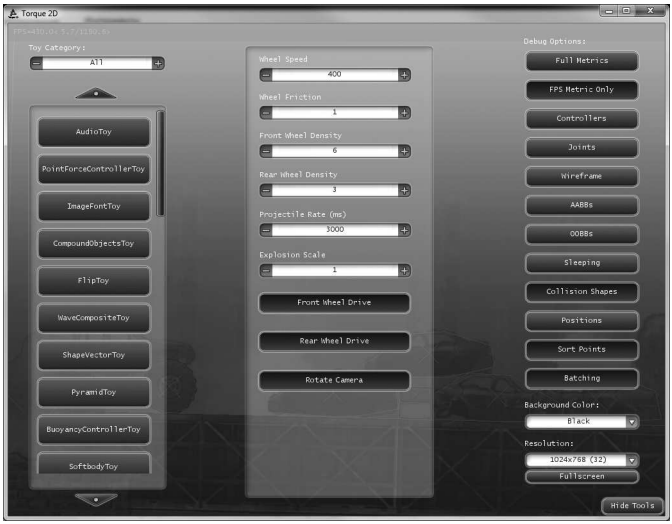

Рис. 3.3. Меню SandBox

включает свойства графического вывода, в нем можно включить или выключить вывод метрик (как то: количество кадров в секунду), выводить объекты в каркасном или текстурированном режиме и прочее.

Демонстрационные игры подготовлены таким образом, что каждая представляет определенную особенность движка Torque 2D, а, поскольку весь скриптовый код открыт и предоставлен, то с реализацией этих возможностей можно познакомиться воочию! А мы не будем на страницах книги детально рассматривать торковские демо-игры, оставим их рассмотрение в качестве домашнего задания.

3 Создание нового проекта

Torque 2D 3.3, распространяемый под свободной лицензией MIT, не обладает средствами для создания нового проекта, другими словами, в его поставке отсутствует менеджер проектов. Однако создать новый проект на движке Torque 2D 3.3 можно двумя путями: включить новый проект в репозиторий SandBox, второй способ состоит в создание проекта с нуля, то есть независимого от SandBox.

Чаще требуется создать независимый от Sandbox'а проект (второй из предложенных выше вариантов). Для этого, к примеру, на уровне папки

modules (хотя независимый проект может располагаться где угодно) создадим папку для нашего проекта, пусть будет: MyProject. Сразу же скопируем в нее исполняемый файл Торка, сейчас он называется Torque2D.exe, но в целевой папке его можно переименовать. Еще понадобятся динамические библиотеки: Leap.dll и OpenAL32.dll. Также в ней создадим подкаталог modules, в котором разместим необходимые для будущей игры ассеты. Вообще, ассеты заслуживают отдельного разговора, поэтому перенесем его на другой раз.

В каталоге с игрой должен находиться файл main.cs, с него начинается выполнение любой торковской игры. Содержимое этого файла с некоторыми исключениями примерно одинаковое:

```
// Set log mode.
setLogMode(2);
// Set profiler.
//profilerEnable( true );
// Controls whether the execution or script files or
///compiled DSOs are echoed to the console or not.
// Being able to turn this off means far less spam in
//the console during typical development.
setScriptExecEcho( false );
// Controls whether all script execution is traced
//(echoed) to the console or not.
trace( false );
// Sets whether to ignore compiled TorqueScript files
//(DSOs) or not.
$Scripts::ignoreDSOs = true;
// The name of the company. Used to form the path to
//save preferences. Defaults to GarageGames
// if not specified.
// The name of the game. Used to form the path to save
//preferences. Defaults to C++ engine define
//TORQUE_GAME_NAME
// if not specified.
// Appending version string to avoid conflicts with
//existing versions and other versions.
setCompanyAndProduct("GarageGames", "Torque 2D" );
// Set module database information echo.
ModuleDatabase.EchoInfo = false;
// Set asset database information echo.
AssetDatabase.EchoInfo = false;
// Set the asset manager to ignore any auto-unload
//assets.
// This cases assets to stay in memory unless assets
//are purged.
AssetDatabase.IgnoreAutoUnload = true;
```

```
// Scan modules.
ModuleDatabase.scanModules( "./modules" );
// Load AppCore module.
ModuleDatabase.LoadExplicit( "AppCore" );
//--------------------------------------------------
function onExit()
{
    // Unload the AppCore module.
    ModuleDatabase.unloadExplicit( "AppCore" );
}
function androidBackButton(%val)
{
    if (%val) {
//Add code here for other options the back button can
//do like going back a screen.  the quit should happen
//at your main menu.
        quit();
}
}
```

По комментариям все должно быть понятно, отмечу лишь некоторые моменты: $Scripts::ignoreDSOs = true; — позволяет задать пропуск скомпилированных DSO-файлов; строка: ModuleDatabase.scanModules(«./modules»); задает каталог с модулями игр, т.к. для одного исполняемого файла может быть несколько игр (подобно Sandbox); с другой стороны строка: ModuleDatabase.LoadExplicit(«AppCore»); указывает основные модули, которые необходимо загрузить в начале загрузки приложения до того, как будут загружены модули игр.

Функция onExit вызывается на выходе (во время завершения приложения) и выполняет одну команду: ModuleDatabase.unloadExplicit («AppCore»); Эта команда выгружает основные модули. Функция androidBackButton вызывается, когда пользователь нажимает аппаратную кнопку Back на смартфоне. В таком случае игра на Torque 2D должна завершиться, поэтому вызывается метод quit. Кроме того, в обработчик нажатия можно добавить другие процедуры, которые необходимо выполнить при завершении игры, например, записать какие-то данные в БД SQLite.

Перейдем, собственно, к цели данного раздела — созданию Torsion-проекта. Для этого в каталоге с проектом создадим текстовый файл MyProject с расширением torsion. Откроем его с помощью Notepad++ и напишем следующий код:

```
<TorsionProject>
<Name>MyProject</Name>
<WorkingDir/>
```

```
<EntryScript>main.cs</EntryScript>
<DebugHook>dbgSetParameters( #port#, "#password#",
true );</DebugHook>
<Mods>
<Folder>modules</Folder>
</Mods>
<ScannerExts>cs; gui</ScannerExts>
<Configs>
<Config>
<Name>Release</Name>
<Executable>MyProject.exe</Executable>
<Arguments/>
<HasExports>true</HasExports>
<Precompile>false</Precompile>
<InjectDebugger>true</InjectDebugger>
<UseSetModPaths>false</UseSetModPaths>
</Config>
<Config>
<Name>Debug</Name>
<Executable> MyProject _debug.exe</Executable>
<Arguments/>
<HasExports>false</HasExports>
<Precompile>false</Precompile>
<InjectDebugger>true</InjectDebugger>
<UseSetModPaths>false</UseSetModPaths>
</Config>
</Configs>
<SearchURL/>
<SearchProduct>main</SearchProduct>
<SearchVersion>HEAD</SearchVersion>
<ExecModifiedScripts>true</ExecModifiedScripts>
</TorsionProject>
```

Поскольку приведенный код не нуждается в особом пояснении, обратим внимание лишь на отдельные моменты. Вначале:

```
<TorsionProject>
<Name>MyProject</Name>
```

указывается название торсион-проекта; в тэгах <EntryScript> и </EntryScript> задается главный скрипт проекта; между тэгами <DebugHook> </DebugHook> вызывается оператор dbgSetParameters для установки удаленного отладчика, в большинстве случаев не используется; тэги <Folder> служат для задания папки, проверяемой при загрузке Torsion'а на наличие скриптовых файлов; следующим тэгом <ScannerExts> выбираются расширения для отбираемых файлов с ко-

дом. Далее следуют секции для установки отладочной и релизной версий проекта, в них задаются имена исполняемых файлов:

```
<Name>Release</Name><Executable>MyProject.exe</
Executable>
```

и

```
<Name>Debug</Name> <Executable>MyProject_debug.exe</
Executable>
```

На этом создание Torsion-проекта завершено; сохраняем и закрываем файл. Щелчком по нему запустится редактор Torsion.

4 Создание сцен и объектов — заготовка для игр

Продолжим работу над созданным в прошлом разделе синглпроектом MyProject. Сначала мы создадим необходимую структуру каталогов, затем рассмотрим базовые объекты, которые необходимо создать для получения основы игры — среды визуализации. Вместе с этим мы обратим внимание на создание некоторых типичных для двумерной игры сущностей.

На первом этапе в каталоге игры (MyProject) создайте подпапку modules. В ней разместите три каталога: AppCore, myProject, ToyAssets. В первом будут храниться скрипты с кодом для инициализации игры; во втором — скрипты, содержащие функциональность конкретной игры, в данном случае — MyProject; в третьем каталоге будут находиться необходимые ассеты — арт. По традиции в каждом каталоге можно создать подпапку с именем 1, но в новых версиях движка это не является обязательным. И уже в последней размещаются все нужные файлы.

В каталоге AppCore/1 создайте подкаталог scripts. Откройте в Torsion проект MyProject, если он еще не открыт. В навигаторе проекта (панель слева в Torsion — рис. 3.4) последовательно раскройте папки: modules/AppCore/1/, щелкнув правой клавишей мыши на папке scripts, выберете из контекстного меню пункт New Script, назовите новый скрипт constants.cs. Так как мы разрабатываем общую основу для игр, которая будет одинаковой почти для любого проекта, конечно, исключения возможны, то нам необходимо поместить в этот файл константы для устройств, работающих под управлением операционных систем iOS и Android:

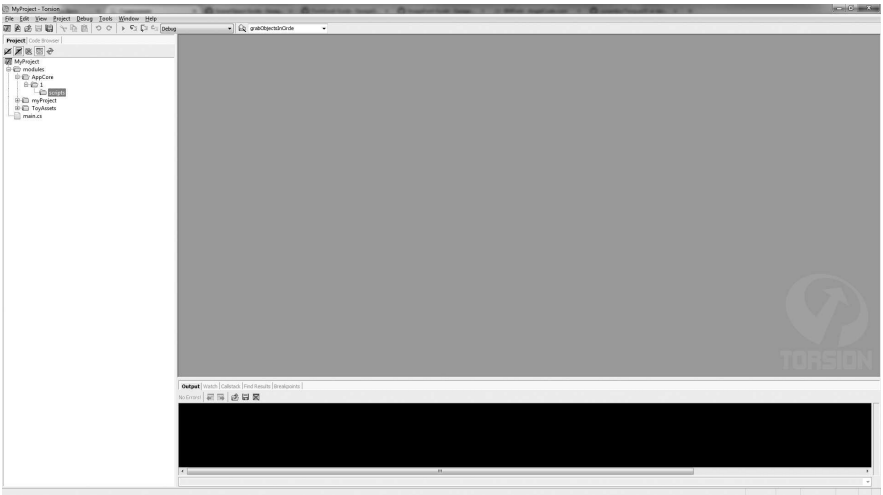

Рис. 3.4. Навигатор проекта в Torsion

```
//константы для определения параметров экрана
//мобильного устройства
$iOS::constant::iPhone = 0;
$iOS::constant::iPad = 1;
$iOS::constant::iPhone5 = 2;
$iOS::constant::Landscape = 0;
$iOS::constant::Portrait = 1;
$iOS::constant::ResolutionFull = 0;
$iOS::constant::ResolutionSmall = 1;
$iOS::constant::iPhoneWidth = 480;
$iOS::constant::iPhoneHeight = 320;
$iOS::constant::iPhone4Width = 960;
$iOS::constant::iPhone4Height = 640;
$iOS::constant::iPadWidth = 1024;
$iOS::constant::iPadHeight = 768;
$iOS::constant::NewiPadWidth = 2048;
$iOS::constant::NewiPadHeight = 1536;
$iOS::constant::iPhone5Width = 1136;
$iOS::constant::iPhone5Height = 640;
$iOS::constant::OrientationUnknown = 0;
$iOS::constant::OrientationLandscapeLeft = 1;
$iOS::constant::OrientationLandscapeRight = 2;
$iOS::constant::OrientationPortrait = 3;
$iOS::constant::OrientationPortraitUpsideDown = 4;
```

Здесь определяются возможные ориентации и размеры экранов мобильных устройств на базе iOS.

Похожим образом создайте файл canvas.cs, в нем будет содержаться код для инициализации области видео-вывода движка Torque 2D:

```
$canvasCreated = false;
function initializeCanvas(%windowName)
{
    // в случае если канва уже создана, то не
//пересоздавать
    if($canvasCreated)
    {
        error("Cannot instantiate more than one
canvas!");
        return;
    }
    videoSetGammaCorrection($pref::OpenGL::gammaCorrec
tion);
    if ( !createCanvas(%windowName) )
    {
        error("Canvas creation failed. Shutting
down.");
        quit();
    }
    $pref::iOS::ScreenDepth = 32;
    if ( $pref::iOS::DeviceType !$= "" )
    {
        %resolution = iOSResolutionFromSetting($pref::
iOS::DeviceType, $pref::iOS::ScreenOrientation);
    }
    else if ($platform $= "Android")
    {
        %resolution = GetAndroidResolution();
    }
    else
    {
        if ( $pref::Video::windowedRes !$= "" )
            %resolution = $pref::Video::windowedRes;
        else
            %resolution = $pref::Video::defaultResolut
ion;
    }
    if ($platform $= "windows" || $platform $=
"macos")
    {
        setScreenMode( %resolution._0, %resolution._1,
%resolution._2, $pref::Video::fullScreen );
    }
```

```
    else
    {
        setScreenMode( %resolution._0, %resolution._1,
%resolution._2, false );
    }
    $canvasCreated = true;
}
//--------------------------------------------------
//перерисовка канвы
//--------------------------------------------------
//function resetCanvas()
{
    if (isObject(Canvas))
        Canvas.repaint();
}
//--------------------------------------------------
//функция для получения разрешения на основе типа
//устройства
//--------------------------------------------------
---------- function iOSResolutionFromSetting(
%deviceType, %deviceScreenOrientation )
{
//вспомогательная функция для получения разрешения из
//настроек
    %x = 0;
    %y = 0;

    %scaleFactor = $pref::iOS::RetinaEnabled ? 2 : 1;
    switch(%deviceType)
    {
        case $iOS::constant::iPhone:
            if(%deviceScreenOrientation ==
$iOS::constant::Landscape)
            {
                %x =  $iOS::constant::iPhoneWidth *
%scaleFactor;
                %y =  $iOS::constant::iPhoneHeight *
%scaleFactor;
            }
            else
            {
                %x =  $iOS::constant::iPhoneHeight *
%scaleFactor;
                %y =  $iOS::constant::iPhoneWidth *
%scaleFactor;
            }
        case $iOS::constant::iPad:
            if(%deviceScreenOrientation ==
$iOS::constant::Landscape)
```

```
            {
                %x =  $iOS::constant::iPadWidth *
%scaleFactor;
                %y =  $iOS::constant::iPadHeight *
%scaleFactor;
            }
            else
            {
                %x =  $iOS::constant::iPadHeight *
%scaleFactor;
                %y =  $iOS::constant::iPadWidth *
%scaleFactor;
            }
        case $iOS::constant::iPhone5:
            if(%deviceScreenOrientation ==
$iOS::constant::Landscape)
            {
                %x =  $iOS::constant::iPhone5Width;
                %y =  $iOS::constant::iPhone5Height;
            }
            else
            {
                %x =  $iOS::constant::iPhone5Height;
                %y =  $iOS::constant::iPhone5Width;
            }
    }

    return %x @ " " @ %y;
}
```

Файл состоит из трех функций и определения нескольких глобальных переменных. В функции initializeCanvas происходит создание области видео-вывода (иными словами, канвы). В функции resetCanvas канва проверяется на существование, если это так, тогда происходит перерисовка. Функция iOSResolutionFromSetting устанавливает разрешение экрана в соответствии с параметрами устройства. Эта функция получает два аргумента: тип устройства и ориентацию, возвращаемым значением является пара чисел: ширина и высота экрана в пикселах.

Следующий файл, который нам нужно создать — это defaultPreferences.cs, он будет содержать общие настройки игры:

```
// глобальные переменные, определяющие параметры
//работы игры:
//состояние приложения,
// используемые им системные функции, управление,
//параметры аудио подсистемы,
```

```
// настройки движка
/// Game
$Game::CompanyName              = "YazevSoft";
$Game::ProductName              = "MyProject";
/// iOS
$pref::iOS::ScreenOrientation   =
$iOS::constant::Landscape;
$pref::iOS::ScreenDepth             = 32;
$pref::iOS::UseGameKit          = 0;
$pref::iOS::UseMusic            = 0;
$pref::iOS::UseMoviePlayer      = 0;
$pref::iOS::UseAutoRotate       = 1;
$pref::iOS::EnableOrientationRotation = 1;
$pref::iOS::EnableOtherOrientationRotation = 1;
$pref::iOS::StatusBarType       = 0;
/// Audio
$pref::Audio::driver = "OpenAL";
$pref::Audio::forceMaxDistanceUpdate = 0;
$pref::Audio::environmentEnabled = 0;
$pref::Audio::masterVolume   = 1.0;
$pref::Audio::channelVolume1 = 1.0;
$pref::Audio::channelVolume2 = 1.0;
$pref::Audio::channelVolume3 = 1.0;
$pref::Audio::sfxVolume = 1.0;
$pref::Audio::musicVolume = 1.0;
/// T2D
$pref::T2D::ParticlePlayerEmissionRateScale = 1.0;
$pref::T2D::ParticlePlayerSizeScale = 1.0;
$pref::T2D::ParticlePlayerForceScale = 1.0;
$pref::T2D::ParticlePlayerTimeScale = 1.0;
$pref::T2D::warnFileDeprecated = 1;
$pref::T2D::warnSceneOccupancy = 1;
$pref::T2D::imageAssetGlobalFilterMode = Bilinear;
$pref::T2D::TAMLSchema="";
/// Video
$pref::Video::appliedPref = 0;
$pref::Video::disableVerticalSync = 1;
$pref::Video::displayDevice = "OpenGL";
$pref::Video::preferOpenGL = 1;
$pref::Video::fullScreen = 0;
$pref::Video::defaultResolution = "1024 768";
$pref::Video::windowedRes = "1024 768 32";
$pref::OpenGL::gammaCorrection = 0.5;
/// Fonts.
$Gui::fontCacheDirectory = expandPath( "^AppCore/
fonts" );
```

В файле есть шесть секций с параметрами: для самой игры, для устройства под iOS, для аудио подсистемы, для видео подсистемы, настройки движка и указатель на папку со шрифтом.

В категории параметров игры задаются название компании разработчика и название приложения.

В категории iOS устройств устанавливается используемая ориентация и глубина экрана (в битах), использование звуков, возможность изменения ориентации экрана игры и прочее.

Среди параметров аудио подсистемы имеются: название используемого аудио драйвера (по умолчанию OpenAL), общий уровень звука (громкость), уровень звука для каждого из трех каналов, громкость для звуковых эффектов.

В категории видеопараметров присутствуют: задание видеодрайвера — поддерживается только OpenGL, включение или отключение вертикальной синхронизации, разрешение (в пикселях), глубина (в битах), разрешить или запретить разворачивать окно игры на весь экран и др.

Для движка настраиваются параметры, связанные с испускателями и системами частиц, устанавливается глобальный фильтр текстур (билинейный), устанавливается TAML схема (по умолчанию пустое значение — “”).

Последний файл, который необходимо создать в папке scripts — openal.cs, он завершит инициализацию игры, настроив и запустив аудио подсистему. Напишите в этот файл такой код:

```
//инициализация аудио подсистемы
//-----------------------------------------------------
// описатели аудио каналов
//-----------------------------------------------------
$musicAudioType = 1;
$effectsAudioType = 2;
//-----------------------------------------------------
// инициализировать OpenAL
// запустить OpenAL драйвер
//-----------------------------------------------------
function initializeOpenAL()
{
    // если драйвер работает, выключить его
    shutdownOpenAL();
    echo("OpenAL Driver Init");
    if (!OpenALInitDriver())
    {
        echo("OpenALInitDriver() failed");
        $Audio::initFailed = true;
```

```
    }
    else
    {
        // установить главную громкость
        alxListenerf(AL_GAIN_LINEAR,
$pref::Audio::masterVolume);
        // установить канал громкости
        for (%channel = 1; %channel <= 3; %channel++)
            alxSetChannelVolume(%channel, $pref::Audio
::channelVolume[%channel]);
        echo("OpenAL Driver Init Success");
    }
}
//--------------------------------------------------
// выключение OpenAL
//--------------------------------------------------
function shutdownOpenAL()
{
    OpenALShutdownDriver();
}
```

В этом файле присутствуют только две функции: первая — initializeOpenAL — инициализирует драйвер и запускает аудио подсистему, вторая — shutdownOpenAL — выключает аудио, выгружая драйвер. Мы не будем останавливать на этих функциях подробно.

На уровне выше подпапки scripts (то есть, в папке с именем 1) надо создать два файла. Первый — это файл main.cs. Ему передается управление от модуля main.cs корневого каталога игры.

```
function AppCore::create( %this )
{
// подключить системные скрипты
exec("./scripts/constants.cs");
exec("./scripts/defaultPreferences.cs");
exec("./scripts/canvas.cs");
exec("./scripts/openal.cs");
// инициализировать канву
initializeCanvas("MyProject");
// установить цвет канвы
Canvas.BackgroundColor = "SeaGreen";
Canvas.UseBackgroundColor = true;

// инициализировать аудио
initializeOpenAL();
ModuleDatabase.loadGroup("gameBase");
ModuleDatabase.loadExplicit("myProject");
}
```

```
function AppCore::destroy( %this )
{
}
```

В этом файле содержатся две функции: AppCore::create и AppCore::destroy. Первая — подключает рассмотренные выше скрипты и вызывает из них функции инициализации видео и аудио подсистем. Кроме того, она устанавливает параметры для канвы: использование и задание фонового цвета. Цвет фона задается строковой константой, в данном случае — SeaGreen. Разнообразные цветовые константы определены в движке, в файле engine\source\graphics\color.cc. Последним действием функция загружает — передает управление в папку myProject. Вторая функция — AppCore::destroy пуста, ей не требуется ничего удалять при завершении работы движка, всем занимается C++ код. Второй файл, который нам необходимо создать — это module.taml (как мы обсуждали выше, его удобно создать с помощью редактора NotePad++). Этот файл играет промежуточную роль при передаче управления от главного файла main.cs из корневого каталога игры к рассмотренному выше файлу из подкаталога AppCore/1. Вот код из файла module.taml:

```
<ModuleDefinition
    ModuleId="AppCore"
    VersionId="1"
    Description=""
    ScriptFile="main.cs"
    CreateFunction="create"
    DestroyFunction="destroy">
</ModuleDefinition>
```

Когда в файле main.cs корневого каталога игры командой ModuleDatabase.scanModules(«modules»); запускается поиск модулей, то первым в очереди обнаруживается файл module.taml, и уже он передает управление файлу main.cs подкаталога AppCore/1 строкой: ScriptFile=«main.cs». Кроме того, в taml файле определяется, какая функция используется при создании: CreateFunction=«create», а какая — при уничтожении игры: DestroyFunction=«destroy».

Теперь мы перейдем в подкаталог myProject/1. Как я говорил ранее: здесь находятся файлы для конкретной игры, в данном случае — myProject. Создадим подпапку scripts, где разместим 3 файла: control.cs, scenewindow.cs и scene.cs.

В первом файле мы напишем код для управления приложением, в данном случае мы хотим, чтобы процесс закрывался, когда пользова-

тель нажимает клавишу Escape. Сначала надо инициализировать карту событий (ActionMap), это происходит в функции InitControl:

```
function Control::InitControl()
{
 new ActionMap(moveMap);
 moveMap.bindCmd(keyboard, "escape", "Control.
exitGame();", "");
 moveMap.push();
}
```

В первой строке тела функции создается новая карта событий. Затем (во второй строке) мы привязываем вызов функции Control.exitGame() к нажатию на клавишу Escape на клавиатуре. Третьим действием мы запихиваем сформированную карту событий в стек. Другими словами, передаем ее на выполнение.

Теперь введем функцию Control::exitGame();

```
function Control::exitGame()
{
   quit();
}
```

Она выполняет только одну команду — quit(), которая завершает приложение и исполняемый процесс.

Во втором файле — scenewindows.cs мы напишем код для создания окна игры и сцены — области визуализации. Первая нужная нам функция — createSceneWindow:

```
function createSceneWindow()
{
// если окно сцены не существует, то...
if ( !isObject(mySceneWindow) )
{
   // создать окно сцены
   new SceneWindow(mySceneWindow);

   // присоединить окно сцены к канве
   Canvas.setContent( mySceneWindow );
}

// устанавливаем профиль для GUI
mySceneWindow.Profile = GuiDefaultProfile;

// настроить камеру
mySceneWindow.setCameraPosition( 0, 0 );
```

```
mySceneWindow.setCameraSize( 100, 75 );
mySceneWindow.setCameraZoom( 1 );
mySceneWindow.setCameraAngle( 0 );
}
```

Сначала в ней создается окно, оно привязывается к ранее созданной канве. Затем для этого окна устанавливается профиль пользовательского интерфейса. Он загружается из файла, в дальнейшем мы рассмотрим этот процесс. В последней секции данной функции происходит настройка параметров камеры: устанавливается ее позиция, размер, коэффициент увеличения масштаба (в данном случае он не нужен, поэтому 1) и поворот.

Обратной функцией созданию окна является функция удаления окна — destroySceneWindow:

```
function destroySceneWindow()
{
// если окно не существует, тогда сразу выйти
if ( !isObject(mySceneWindow) )
return;

// удалить окно
mySceneWindow.delete();
}
```

В дополнительных комментариях код данной функции не нуждается.

Создание и разрушение сцены происходит в соответствующих функциях: createScene и destroyScene файла scene.cs. Ниже приведен их код с комментариями:

```
function createScene()
{
// удалить сцену, если она уже существует
if ( isObject(myScene) )
    destroyScene();

// создать сцену
new Scene(myScene);
}
function destroyScene()
{
// если сцена не существует, то сразу выйти
if ( !isObject(myScene) )
    return;
// удалить сцену
myScene.delete();
}
```

В файле scene.cs есть еще одна функция: createSprite, как следует из ее названия, она создает спрайт. Спрайт в двумерной графике представляет собой четырехугольник с наложенной текстурой — растром. Вот код данной функции:

```
function createSprite()
{
 // создать спрайт
%sprite = new Sprite();

// установить для спрайта статический тип,
// при котором на него не будет воздействовать
//гравитация
%sprite.setBodyType( static );
// поставить спрайт в нулевые координаты, то есть в
//центр экрана
%sprite.Position = "0 0";
// размер спрайта
%sprite.Size = "20 5";

// слой, на котором будет находиться спрайт
%sprite.SceneLayer = 31;

// загрузить изображение из ассета
%sprite.Image = "ToyAssets:label";

// добавить спрайт на экран
myScene.add( %sprite );
}
```

В начале выполнения данной функции создается объект класса Sprite, после чего для этого объекта устанавливаются свойства. С помощью метода setBodyType с параметром static, для спрайта задается способ взаимодействия с окружением, static означает, что на объект не будет воздействовать гравитация. Затем позиция спрайта устанавливает в 0 по X и в 0 по Y, это означает центр экрана. Далее мы устанавливаем размер для спрайта по обеим осям координат. Задавая номер слоя, на котором будет находиться объект, мы тем самым определяем взаимное расположение нескольких объектов, то есть, кто кого будет перекрывать. Но, так как в нашей текущей сцене имеется только один объект, то никаких проблем не будет. О слоях надо добавить то, что, поскольку Torque 2D двумерных движок и не имеет третьей оси (Z), то слои представляют для нее альтернативу. Всего имеется 32 слоя: от 0 до 31. 0-ой слой самый ближний, 31-ый — самый дальний. Следующее свойство — Image спрай-

та указывает на загружаемый графический ассет. Описание ассетов мы рассмотри чуть ниже. Наконец, последним действие мы добавляем наш только что проинициализированный спрайт в стек экрана, из которого он будет визуализироваться в соответствии со своими свойствами.

Теперь перейдем на уровень выше — из папки scripts. В родительском каталоге 1 надо создать файл main.cs. Этот файл будет содержать два метода: конструктор и деструктор. Первый имеет следующий вид:

```
function myProject::create( %this )
{
   exec("./gui/guiProfiles.cs");
   exec("./scripts/scene.cs");
   exec("./scripts/scenewindow.cs");
   exec("./scripts/control.cs");
   createSceneWindow();
   createScene();
   mySceneWindow.setScene(myScene);
   createSprite();
   new ScriptObject(Control);
   Control.InitControl();
}
```

Вначале в нем подключаются рассмотренные выше файлы с кодом, плюс подключается файл с профилем пользовательского интерфейса. Затем вызываются описанные ранее функции для создания окна приложения и сцены визуализации. Двумя последними строчками создается скриптовый объект, затем для него вызывается инициализатор управления.

Профиль пользовательского интерфейса имеет следующий вид:

```
if(!isObject(GuiDefaultProfile)) new GuiControlProfile
(GuiDefaultProfile)
{
   Modal = true;
};
```

Содержащий его файл должен называться guiProfiles.cs, он должен находиться в подпапке gui каталога 1.

Деструктор имеет следующий вид:

```
function myProject::destroy( %this )
{
   destroySceneWindow();
   Control.delete();
}
```

Он разрушает объект — окно плюс удаляет скриптовый объект, служащий для управления.

Кроме того, на этом уровне есть файл module.taml, с которого начинается работа с main.cs файлом. Вот содержащийся в нем код:

```
<ModuleDefinition
    ModuleId="myProject"
    VersionId="1"
    Description="myProject"
    Dependencies="ToyAssets=1"
    ScriptFile="main.cs"
    CreateFunction="create"
    DestroyFunction="destroy">
</ModuleDefinition>
```

Указан идентификатор модуля, версия, описание, зависимости, выполняемый скриптовый файл, описанные в нем конструктор и деструктор.

Третья подпапка каталога modules (ToyAssets) содержит два элемента: файл module.taml и подпапку assets. Первый включает описание пути загрузки ассетов: графических и других материалов, вот его код:

```
<ModuleDefinition
    ModuleId="ToyAssets"
    VersionId="1"
    Description="Torque Label"
    ToyCategoryIndex="0">
   <DeclaredAssets
     Path="assets"
     Extension="asset.taml"
     Recurse="true"/>
</ModuleDefinition>
```

В описании имеются: идентификатор модуля, который будет использоваться для указания пути в скриптовом коде, версия, строка — описание, индекс категории. В секции DeclaredAssets находится описание места и способа поиска файлов описания ассетов. В данном случае указан путь — свойство Path — значение имя подпапки assets, в которой будет произведен поиск файлов с расширением (свойство Extension) «asset.taml». Также мы указываем возможность рекурсивного поиска (свойство Recurse устанавливаем в значение true).

В подпапке assets создайте подкаталог images, в который надо поместить изображение. Оно будет загружено в движок и наложено на созданный нами ранее спрайт. Я подготовил изображение label.png, кото-

рое можно взять среди материалов книги. Кроме того, вы сами можете нарисовать прямоугольный растр в любом графическом редакторе. Я считаю, что наиболее подходящим форматом графических файлов для игр является PNG. Теперь рядом с графическим файлом в папке images надо создать описывающий этот файл модуль — label.asset.taml. При сканировании папок для поиска ассетов движком во время запуска игры, он первым делом натыкается на подобные файлы описания, и по информации из них загружает соответствующий контент. Запишите в файл label.asset.taml такой код:

```
<ImageAsset
    AssetName="label"
    ImageFile="label.png"
    />
```

Первая строчка сообщает о начале описания графического ассета, во второй — ассету дается имя, с помощью которого данный ассет можно загрузить в игру, в третьей строчке указывается файл с изображением, который необходимо загрузить и контент которого будет представлен ассетом.

На этом заготовка для игр готова! Сохраните все открытые файлы. Теперь можно испытать нашу заготовку, запустите исполняемый модуль

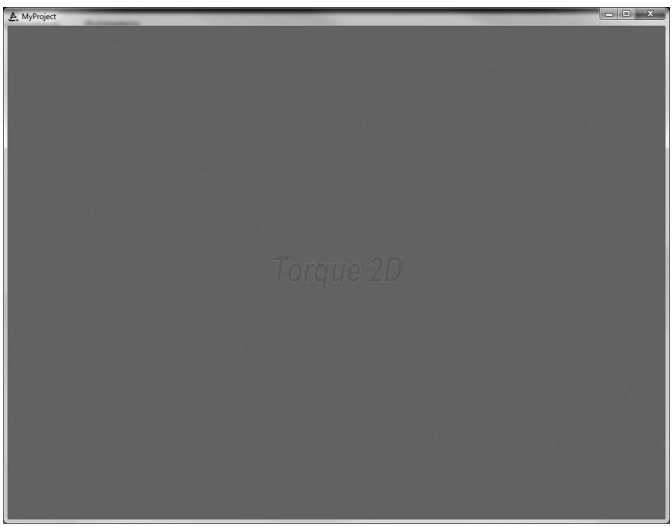

Рис. 3.5. Готовая заготовка для наших игр

из корневой папки с игрой (я назвал этот файл MyProject.exe, то есть просто переименовал файл Torque2D.exe). Если все предыдущие операции выполнены верно, то запустится оконное приложение, в котором фон будет закрашен в светло-зеленый цвет, а в центре окна будет выведена надпись Torque 2D, представленная изображением (рис. 3.5).

5 Заключение

В текущей главе мы сделали большой шаг: от теоретического материала к практике. Мы создали заготовку для своих игр на движке Torque 2D. Мы рассмотрели содержимое каталога движка, набор демонстрационных игр на Torque 2D — SandBox, научились создавать проекты, в качестве основы использующие функциональность SandBox'а, а так же, что более интересно, создавать их с нуля. В результате мы разработали небольшое, но включающее элементы двумерных игр на движке T2D приложение. Оно создает окно для приложения, сцену для визуализации, скриптовый объект для управления, спрайт, на который накладывает загруженное из файла изображение; инициализирует аудио подсистему, канву. В процессе разработки приложения мы увидели работу с внешними ресурсами — графическими ассетами, научились описывать их с помощью языка TAML и загружать их в игру. Мы увидели системные объекты движка Torque 2D необходимые для любой игры: объект — окно и объект — сцена.

Теперь имея большой багаж знаний об играх, их разработке и программировании с использованием движка Torque 2D, мы можем переходить дальше к более интересным и захватывающим темам, к непосредственной разработке двумерных игр на движке Torque 2D!

Перед тем как мы займемся разработкой игр, нам необходимо узнать об имеющихся в Torque 2D объектах и их физических свойствах. Поэтому следующие две главы посвящены этим темам.

Глава 4. Объекты движка

Оглавление

Какие средства предоставляет Torque 2D игроделу? С помощью чего программист строит свой проект, используя движок Torque 2D? В этой главе мы разберемся с этими вопросами, представив набор объектов,

готовых для создания двумерных игр. Этот набор включает все необходимое, что может понадобиться для разработки любой 2D-игры. Описание всех объектов осуществляется на языке Torque Script, а файловых материалов — на TAML.

1 Игровые объекты

Класс Scene

Объект класса Scene — это контейнер для всех объектов. Хотя бы один такой объект обязательно должен присутствовать в каждой игре, так как, благодаря ему, спрайты, текстовые надписи и другие игровые объекты выводятся на экран. В игре может присутствовать несколько объектов класса Scene, например, на одном — визуализируется поле боя, а на втором — отображается радар.

Создание объекта класса Scene не отличается от создания других объектов движка, достаточно на Torque Script написать такую строчку:

```
%scene = new Scene();
```

Таким образом, в локальной переменной %scene сохранен указатель на объект класса Scene. Объекты этого класса управляют глобальными свойствами мира игры. В рамках Torque 2D можно приравнять игровой мир (world) с объектом Scene.

К числу глобальных свойств игрового мира относится гравитация. В Torque 2D гравитация может иметь любое направление, например, чтобы задать притяжение аналогично притяжению нашей планеты Земля (надеюсь, я допишу эту книгу до того момента, когда человечество переселится на какую-либо другую планету) надо задать значение для свойства Gravity объекта Scene одним из двух способов: непосредственно присвоить значение свойству или с помощью метода setGravity:

```
%scene.Gravity = "0 -9.8"; // присвоение значения
//свойству
%scene.setGravity(0, -9.8); // использование метода
```

Для свойства передается строка — дуплет числовых значений, разделенных пробелом и соответствующих притяжению по оси X (первое значение) и Y (второе значение). Подобное мы имеем в параметрах метода.

Чтобы получить значение гравитации, можно считать его из свойства:

```
%gravity = %scene.Gravity;
```

Или воспользоваться методом getGravity, возвращающим строку, состоящую из двух значений:

```
%gravity = %scene.getGravity();
```

Основная возможность объекта типа Scene — это управление своим содержимым, которое состоит из других игровых объектов, предназначенных для визуализации. Чтобы включить в объект Scene какой-либо объект, надо для первого вызвать метод add, в параметре которого передать объект для добавления: `%scene.add(%object);`

Чтобы удалить объект из сцены (Scene) достаточно для последней вызвать метод remove, в который передать ссылку на удаляемый объект: `%scene.remove(%object);`

Для того чтобы получить количество всех объектов, содержащихся в сцене, и оттого предназначенных для отображения, достаточно вызвать метод getCount сцены: `%count = %scene.getCount();`

Вышеприведенным образом, с помощью трех методов: add, remove и getCount работают все контейнеры в TorqueScript.

Прекрасным свойством класса Scene является возможность определения объекта, находящегося в определенной точке. Эта возможность позволяет выбрать все объекты, расположенные в заданных координатах. Кроме того, для выбора может быть использована не только координатная точка, а так же круг произвольного радиуса и прямоугольник любого соотношения сторон. Ниже приведены примеры.

В первом примере мы сообщаем движку, что надо выбрать все объекты, находящиеся в координатах: X = 50, Y = 10 — первый параметр, далее три пустых параметра говорят о том, что не важно какой визуализируемой группе принадлежит объект, ровно как не важно какому слою и группе коллизии:

```
%scene.pickPoint("50 10", "", "", "Any");
```

Во втором примере выбираются все объекты, попадающие в круг с радиусом 3, находящимся в координатах: X = 80, Y = 60:

```
%scene.pickCircle("80 60", 3);
```

В третьем примере мы выбираем все объекты, находящиеся в прямоугольнике 5 на 9 с координатами 20 на 20. Все остальные для выбора параметры оставляем по умолчанию:

```
%scene.pickArea("20 20", "5 9");
```

Во время разработки часто нужно увидеть дополнительные сведения об имеющихся в сцене объектах. Класс Scene позволят получить такую информацию. Путем задания уровня отображения отладочной информации, можно увидеть: контроллеры, точки соединения, границы столкновения и т.д. Кроме того, Torque 2D может вывести число кадров в секунду и многие другие метрики. Следующая строчка кода включает отображение границ столкновений объектов:

```
%scene.setDebugOn("collision");
```

Чтобы вывести сетки объектов, надо написать:

```
%scene.setDebugOn( "wireframe" );
```

Чтобы включить отображение количества кадров в секунду достаточно вызвать следующий метод:

```
%scene.setDebugOn( "fps" );
```

Напротив, что выключить, надо вызвать:

```
%scene.setDebugOff( "fps" );
```

А, чтобы вывести все метрики, достаточно написать:

```
%scene.setDebugOn( "metrics" );
```

Между тем как вы увидели: объект класса Scene представляет собой виртуальный контейнер для других объектов, содержимое этого контейнера визуализируется и выводится на экран. Плюс к этому нам нужен объект, который бы представлял окно для приложения операционной системы. Как раз для этого служит класс SceneWindow.

Класс SceneWindow

Попросту говоря, объект этого класса представляет элемент пользовательского интерфейса типа окно. SceneWindow создан для того, чтобы содержать игровую сцену (Scene). Изменение параметров объекта SceneWindow (например, размера путем перетаскивания границ окна) не оказывает никакого влияния на содержащуюся внутри сцену. При-

ложение может иметь одно или несколько объектов класса SceneWindow, то есть одно или несколько окон.

Чтобы создать глобальный объект SceneWindow с именем mySceneWindow достаточно написать такую строчку:

```
new SceneWindow(mySceneWindow);
```

Объект SceneWindow обязательно содержит профиль по умолчанию в качестве профиля используется стандартный профиль GuiDefaultProfile. Третьим обязательным действием объект SceneWindow должен быть передан канве (объект Canvas). Канва представляет весь экран, область для отображения. Поэтому надо занять определенное ограниченное пространство, представляемое окном приложения, что мы осуществляем с помощью следующей строки:

```
Canvas.setContent(mySceneWindow);
```

Объект канвы тоже должен быть предварительно создан. Для этого в Torque 2D есть специальная функция *createCanvas*. В качестве параметра она принимает заголовок окна, а возвращает булево значение, в случае если канва успешно создана — true, а в случае неудачи — false. Кроме того, предварительно (перед созданием канвы) в Torque 2D должна быть задана гамма коррекция с помощью функции videoSetGammaCorrection, параметром она получает значение коррекции. После завершения работы приложения восстанавливается стандартная гамма коррекции.

После создания канвы необходимо установить режим экрана. Это осуществляется с помощью функции setScreenMode. Режим экрана состоит из 4-х параметров, значения которых зависят от платформы, на которой в данный момент выполняется игра. Первый параметр представляет ширину области вывода в пикселах, второй — высоту, третий — глубину цвета (количество бит на пиксель, может принимать два значения: 16 бит или 32), последний — четвертый параметр — булевый флаг, обозначающий работу игры в полном экране или нет. Если режим установлен успешно, функция возвращает true, иначе — false.

Для получения каждого из установленных параметров существуют соответствующие функции: getDesktopResolution — возвращает разрешение экрана и глубину цвета; getDisplayDeviceList — возвращает графическую подсистему, как мы помним: в Torque 2D используется OpenGL; функция getResolutionList в качестве параметра принимает имя графической подсистемы (OpenGL), а возвращает список поддерживаемых

разрешений; функции nextResolution и prevResolution, соответственно, устанавливают следующее и предыдущее значение разрешения из списка. С помощью функции setDispalyDevice можно установить другую графическую подсистему, но в данных реалиях это сделать не получиться, так как поддерживается только OpenGL (очевидно, эта возможность предусмотрена на будущее) и установить другие параметры вывода, такие как: разрешение, глубина цвета и возможность запуска в полноэкранном режиме. Кроме того, изменить параметры вывода можно, воспользовавшись функцией setRes. С помощью функции switchBitDepth можно изменить глубину цвета.

После создания окна приложения и присвоения его канве обязательными действиями являются настройки следующих параметров: установка позиции камеры, обычно ее надо установить в середину сцены, для этого, вызвав метод setCameraPosition, передать ему два нулевых параметра:

```
mySceneWindow.setCameraPosition(0, 0);
```

Затем надо задать размеры визуализируемой области в единицах измерения Torque 2D:

```
mySceneWindow.setCameraSize($screenWidth,
$screenHeight);
```

Также надо установить уровень масштаба изображения (другими словами, приближения) обычно 1:

```
mySceneWindow.setCameraZoom(1);
```

Наконец, надо установить угол поворота камеры, обычным значением передается 0:

```
mySceneWindow.setCameraAngle(0);
```

Фон окна можно залить определенным цветом. Сначала разрешаем использование цвета для заливки поверхности окна, другими словами — фона:

```
SandboxWindow.setUseBackgroundColor(true);
```

Затем указываем: каким цветом следует залить окно (в данном случае, желто-зеленым):

```
SandboxWindow.setBackgroundColor(YellowGreen);
```

В Torque 2D определено много именованных констант, соответствующих определенным цветам по цветовой схеме RGB.

Теперь, когда мы рассмотрели объекты сцены — контейнера для всех игровых элементов и окна, содержащего сцену, перейдем к рассмотрению классов, служащих для создания игровых объектов.

Класс SceneObject

Базовым классом для всех игровых объектов в Torque 2D является SceneObject. Объекты этого класса могут быть добавлены в стек сцены — на экран. Такие объекты содержат только самые общие свойства и методы, которыми должен обладать любой объект в мире двумерной игры, это такие свойства: позиция (координаты по осям X и Y), размер (также размерность по двум осям), номер слоя, базовые физические свойства и др. Этот класс так же содержит свойства визуализации, однако они используются только в классах потомках. Сам по себе объект класса SceneObject невидимый и может отображаться только в режиме отладки.

Рассмотрим некоторые — самые важные свойства.

Свойство LifeTime, тип float — задает время жизни объекта в секундах. По истечению этого периода объект автоматически удаляется. По умолчанию у объекта задано бесконечное время жизни. Если объект живет бесконечно долго (данное свойство не изменено), тогда программисту надо самому удалить объект, вызвав метод delete (или safeDelete) данного объекта.

Пример:

```
%object = new SceneObject();
%object.Lifetime = 10;
```

В данном примере для объекта %object срок жизни задается равный 10 секундам.

У каждого свойства, помимо обращения к его значению через имя, есть функции для получения и установки значения, например:

```
%lifeTime = %object.getLifeTime(); //поместить в
//переменную %lifeTime время жизни объекта
%object.setLifeTime(10);//задать время жизни объекта
```

Свойство SceneLayer типа int — устанавливает номер слоя, на котором находится данный объект. Номер слоя управляет расположением

объекта в отношение к камере по оси Z; всего в Torque 2D имеется 32 слоя, которые нумеруются от 0 до 31: 0 означает самый ближний к камере слой, 31 — самый дальний от камеры.

Пример:

```
%object.SceneLayer = 31;
```

Объект %object максимально отдаляем от камеры. То есть он будет перекрываться объектами, расположенными на других слоях.

Свойство SceneGroup задает номер экранной группы (визуализации), в которой находится данный объект. Количество экранных групп так же ограничено 32.

```
%object.SceneGroup = 15;
```

Свойство Position позволяет установить объект в определенных двумерных координатах экрана, соответственно, по осям X и Y. Чтобы задать координаты объекта, надо передать пару координат в одной строке, соответственно, X и Y:

```
%object.Position = "50 60";
```

Также задать значение координаты по определенной оси можно с помощью функций setPositionX(%x) и setPositionY(%y), соответственно. С другой стороны, чтобы получить значение координаты по одной из осей надо воспользоваться функцией getPosition() и взять из возвращенного значения только одно число вот так: getPosition().X и/или getPosition().Y.

Свойство Size задает размер объекта по двум осям: X и Y. Этому свойству в качестве значения так же передается строка, содержащая два числовых значения, разделенных пробелом. Например:

```
%object.Size = "8 4";
```

Метод setSize имеет две перегруженных формы: с одним параметром и двумя. В первом случае одно и то же значение применяется к размеру по обеим осям: `%object.setSize(10);`. Во втором случае — с двумя параметрами: первое значение — применяемый размер объекта по оси X, второе — размер по Y: `%spaceship.setSize(10, 5);`. Чтобы получить размер объекта в виде дуплета значений, надо вызвать метод `getSize: %object.getSize();`.

С помощью свойства angle задается угол поворота объекта против часовой стрелки: `%object.Angle = 90;`. Также, для задания и получения значения этого свойства можно воспользоваться, соответственно, методами: setAngle и getAngle:

```
%object.setAngle(90);//поворачиваем объект
%angle = %object.getAngle();//сохраняем в переменной
//угол поворота объекта
```

Следующее свойство — bodyType (тип объекта) определяет способность взаимодействия данного объекта с другими объектами. В Torque 2D имеется три типа объекта: static (статический), dynamic (динамический), kinematic (кинематический). Объекты этих типов взаимодействуют согласно физическим законам. Подробнее мы рассмотрим все типы в следующей главе: «Физические свойства и взаимодействия». По умолчанию каждый объект динамического типа.

Чтобы задать тип объекта, достаточно присвоить этому свойству желаемое значение (static, dynamic, kinematic): `%object.bodyType = static;` Можно, также, воспользоваться методом `setBodyType`: `%object.setBodyType(static);` С другой стороны, чтобы получить значение этого свойства, надо вызвать метод getBodyType, который возвращает строковое значение типа объекта, например:

```
echo(%object.getBodyType());
==> static
```

Класс SpriteBase

Еще один базовый класс, в свою очередь, наследуется от SceneObject. Представляет собой класс-предок для всех визуализируемых объектов, например: Sprite, Scroller, хотя сам не умеет отображаться на экране, то есть его нельзя поместить в стек сцены. Из этого следует, что SpriteBase играет роль абстрактного класса.

Тем не менее, как было сказано выше: SpriteBase включает различные свойства для визуализации и отображения. Всего их 3. Рассмотрим их.

Чтобы загрузить и отобразить одиночное изображение, используется свойство Image. В качестве значения ему присваивается корректный ассет изображения — ImageAsset. Пример:

```
%object.Image = "ToyAssets:ground";
```

Также можно использовать методы: setImage(assetImageID) — для установки изображения, getImage() — для получения имени ассета — загруженного изображения.

Загруженное изображение может быть поделено на кадры, чтобы выбрать для отображения определенный кадр — по порядковому номеру, этот номер надо присвоить свойству Frame:

```
%object.Frame = 1;
```

С другой стороны, можно воспользоваться методами: setImageFrame(int) — для выбора определенного кадра, getImageFrame() — для получения номера отображаемого кадра.

Чтобы отобразить последовательность сменяющихся изображений — анимацию, надо воспользоваться свойством Animation. Значением этого свойства выступает корректный animationAssetID:

```
%object.Animation = "ToyAssets:TileAnimation";
```

Для загрузки и проигрывания анимации используется метод playAnimation(animationAssetID), а, чтобы узнать имя загруженного анимационного ассета — воспользоваться методом getAnimation.

С анимацией связана одна функция обратного вызова. Если анимационная последовательность не зациклена, тогда при достижении последнего фрейма анимации, вызывается событие onAnimationEnd(%this), где в параметре передается указатель на объект SpriteBase. С помощью этого указателя можно узнать имя проигрываемой анимации и предпринять соответствующие меры:

```
%anim = %this.getAnimation();
```

Воспользовавшись методами класса SpriteBase можно: остановить анимацию — stopAnimation(), приостановить анимацию (на текущем кадре) — pauseAnimation(), установить кадр для проигрывания анимации — setAnimationFrame(int), получить номер текущего кадра анимации — getAnimationFrame(), узнать время анимации — getAnimationTime, узнать, завершена ли анимация — getIsAnimationFinished(); кроме того, можно изменить скорость проигрывания анимации: setAnimationTimeScale(float), при этом по умолчанию скорость анимации равна 1; чтобы получить скорость проигрывания анимации, надо воспользоваться методом getAnimationTimeScale.

Класс Sprite

В предыдущей главе мы рассмотрели некоторые методы и атрибуты класса Sprite. Как было видно из описания: объекты этого класса представляют собой двумерные прямоугольники с наложенной текстурой. Эти объекты обладают широким набором свойств, например, такими как: положение (по двум осям), размер, физический тип, изображение — текстура и прочие. Кроме того, за место текстуры может быть использована анимация. Определенно, класс Sprite наиболее часто используемый в Torque 2D. При этом он последовательно наследуется от SpriteBase и SceneObject, которые мы рассмотрели выше. В дополнение к свойствам, представленных выше классов, Sprite имеет только 2: FlipX — для зеркального отражения спрайта по оси X и FlipY — для оборота спрайта по оси Y.

SkeletonObject

Данный класс представляет объект похожий на Sprite, отличие заключается в том, что SkeletonObject выводит на своей поверхности скелетную анимацию. Этот класс был добавлен в третью версию движка. Поскольку он наследуется от SceneObject, то включает все его возможности. SkeletonObject использует данные из SkeletonAsset (ассеты рассматриваются во втором разделе данной главы). По сути, скелетная анимация — это техника, в которой персонаж (объект) представлен двумя частями: поверхность (называемая кожей — skin) используется для его рисования и набор иерархически соединенных костей (другими словами, скелет) используется для анимации кожи (skin). Класс SkeletonObject обладает следующими свойствами и методами:

Для загрузки скелетной анимации используется свойство Asset:

```
%object = new Skeleton();
%object.Asset = "SpineToy:Goblins";
```

С ним можно работать через методы: setSkeletonAsset(string) — для загрузки анимации и getSkeletonAsset() — для получения.

Кроме того, у скелетной анимации есть имя — свойство AnimationName. Для его установки и получения, соответственно, используются методы: setAnimationName(string) и getAnimationName().

Свойство Skin позволяет установить «кожу» для отображения поверх скелета: getSkin(string) и получить ее getSkin().

Свойство RootBoneScale позволяет установить фактор масштабирования для основной кости: setRootBoneScale(float) — для установки, getRootBoneScale() — для получения.

Смещение основной кости по осям X и Y можно выполнить с помощью свойства RootBoneOffset: setRootBoneOffset(float) для установки смещения и getRootBoneOffset() — для получения значений смещения.

Скелетную анимацию можно зациклить, воспользовавшись свойством AnimationCycle. Для его установки служит метод setAnimationCycle с булевым параметром, а для получения значения — getAnimationCycle.

Последние 2 свойства служат для переворота анимации, соответственно, по осям X и Y. Чтобы перевернуть анимацию по X надо воспользоваться методом setFlipX с передачей булева параметра, также по Y — с применением метода setFlipY.

Для получения продолжительности анимации в секундах существует метод getAnimationDuration().

Чтобы смешать несколько анимаций, служит метод setMix().

Для создания объекта класса SkeletonObject может послужить следующий код:

```
%spineObject = new SkeletonObject();
%spineObject.Asset = "MyModule:Goblins";
%spineObject.Skin = "GoblinGirl";
%spineObject.setAnimation("walk", true);
%spineObject.RootBoneScale = "0.025";
%spineObject.RootBoneOffset = "0 -5";
MyScene.add(%spineObject);
```

CompositeSprite

Объект класса CompositeSprite состоит из произвольного числа других спрайтов, для каждого из которых можно определить отдельное изображение или анимацию. Каждый составной спрайт может иметь произвольные позицию, размер, ориентацию, находиться на любом слое и независимо иметь другие параметры. В целом, CompositeSprite может иметь следующие свойства разметки:

— No Layout — логическая позиция равна физической;

— Rectilinear Layout — логическая позиция проецируется прямолинейно;

— Isometric Layout — логическая позиция проецируется на изометрическую;

— Custom Layout — логическая позиция проецируется на пользовательскую;

CompositeSprite предназначен не только для создания тайловых сеток (мозаик), его так же можно использовать для сложных составных персонажей. Кроме того, использование CompositeSprite позволяет добиться высокой производительности при большом количестве объектов. Поскольку этот объект использует пакетирование и позволяет изменять любые свойства входящих в него спрайтов. Вдобавок повышенная производительность достигается путем быстрой визуализации и при необходимости отсечения части CompositeSprite не попадающей на экран.

Кроме наследуемых от SceneObject свойств, CompositeSprite открывает несколько новых:

— DefaultSpriteSize — размер для создаваемых внутри спрайтов;

— DefaultSpriteAngle — угол поворота для спрайтов;

— BatchLayot — тип пакетного размещения для визуализации; поддерживаются следующие режимы:off, rect, iso, custom — все они описаны выше.

— BatchCulling — устанавливает: надо ли проводит пакетный отбор — отбраковку спрайтов;

— BatchIsolated — устанавливает: надо ли изолировать спрайты при визуализации разных пакетов;

— BatchSortedMode — устанавливает режим сортировки при визуализации;

С помощью дополнительных методов CompositeSprite может управлять своим содержимым:

— addSprite — добавляет спрайт в коллекцию в указанную логическую позицию (относительно начала координат объекта CompositeSprite);

— removeSprite — удаляет выбранный спрайт;

— clearSprites — удаляет все спрайты из коллекции;

— getSpritesCount — возвращает количество спрайтов в композиции;

— selectSprite — выбирает спрайт в заданных логических координатах;

— selectSpriteId — выбирает спрайт по идентификатору;

— selectSpriteName — выбирает спрайт по имени;

— deselectSprite — отменяет выбор спрайта;

— isSelectedSprite — проверяет: выбран ли спрайт;

— setSpriteImage — задает для спрайта изображение;

— getSpriteImage — возвращает изображение спрайта;

— setSpriteAnmation — устанавливает для спрайта анимацию;

— getSpriteAnimation — возвращает анимацию, загруженную для спрайта;

— clearSpriteAsset — очищает любой загруженный в спрайт ассет: изображение или анимацию;

— setSpriteVisible, getSpriteVisible — соответственно, устанавливает и возвращает состояние видимости;

— setSpriteLocalPosition, getSpriteLocalPosition — соответственно, устанавливает и возвращает локальную позицию спрайта;

— setSpriteAngle, getSpriteAngle — устанавливает и возвращает угол поворота;

— setSpriteSize, getSpriteSize — устанавливает и возвращает размер спрайта;

Scroller

Класс Scroller несколько отличается от других игровых объектов. Как и Sprite он наследуется от SpriteBase. Подобно спрайту Scroller загружает статическое изображение, но в отличие от первого позволяет это изображение автоматически прокручивать по вертикали и горизонтали. Свойства для визуализации изображения унаследованы от SpriteBase; Scroller добавляет небольшое количество новых свойств и методов, предназначенных для прокрутки изображения.

С помощью свойств RepeatX и RepeatY можно задать количество повторений картинки, соответственно, по осям X и Y. Обычный спрайт растягивает картинку на всю площадь своего содержимого. С другой стороны, Scroller может показать текстуру больше или меньше раз по этим осям. В зависимости от размера скроллера, разрешения экрана и размера текстуры можно сплющить или растянуть изображение по этим осям.

```
%object = new Scroller();
%object.RepeatX = 0.5;
%object.RepeatY = 0.5;
```

Свойства ScrollX и ScrollY устанавливают скорость прокрутки в направлении по осям X и Y, соответственно. Положительные значения позволяют задать прокрутку слева направо по оси X и сверху вниз — по Y.

```
%object.ScrollX = 10;
%object.ScrollX = 10;
```

Поля ScrollPositionX и ScrollPositionY позволяю установить позицию отображения текстуры на экране, соответственно, по X и Y. Значения

имеют диапазон от 0 до 1, вместе с тем они соответствуют размеру загруженного изображения. Например, если этим свойствам присвоить значение 0.5, то изображение будет сдвинуто на половину по вертикали и горизонтали:

```
%object.ScrollPositionX = 0.5;
%object.ScrollPositionY = 0.5;
```

Дополнительные методы используются для установки значений вышеприведенным свойствам:

setRepeat — имеет 2 параметра и устанавливает количество повторений по обеим осям;

setScroll — имеет 2 параметра и устанавливает скорость прокрутки по обеим осям;

setScrollPosition — также имеет 2 параметра, устанавливает смещения по двум осям.

ImageFont

Класс ImageFont присутствовал в Torque 2D до версии 3.3. То есть до настоящей мастер-версии движка — 3.3 он являлся предпочтительным способом вывода текста.

Как и все визуальные объекты ImageFont унаследован от SceneObject. ImageFont служит для вывода текста в сцене (Scene). Для этого используются растровые шрифты, которые представляют коллекции символов.

ImageFont представляет следующие свойства:

Image — изображение — ImageAsset, хранящее карту символов (минимум 96 символов):

```
%font = new ImageFont();
%font.Image = "ToyAssets:FancyFont";
```

Text — содержит текст для вывода с помощью заданного шрифта;

```
%font.Text = "yurembo";
```

TextAlignment — выравнивание текста: Left, Center, Right:

```
%font.TextAlignment = "center";
```

FontSize — размер текста по обеим осям: X и Y:

```
%font.FontSize = "3 2";
```

FontPadding — задает расстояние между символами:

```
%font.FontPadding = 2;
```

В отличие от остальных объектов ImageFont по умолчанию статический из-за чего на него не действует гравитация и другие физические силы, но это легко исправить, присвоив свойству BodyType значение dynamic.

TextSprite

Класс TextSprite появился только в версии Torque 2D 3.3, master-версия которой стала доступна незадолго до сдачи рукописи в печать. Он пришел на смену классу ImageFont, который был удален. TextSprite как и многие игровые объекты унаследован от SceneObject. TextSprite представляет собой полноценное решение для отображения текстовых надписей в сцене (Scene). Для этого используются растровые шрифты. Последние можно создать в свободной программе Angle Code's Bitmap Font Generator. Или любой другой, поддерживающей сохранение в таком же формате. Для загрузки растрового шрифта в формат пригодный для TextSprite используется FontAsset, он рассматривает во втором разделе данной главы.

Свойство Font предназначено для присвоения битового шрифта — объекта FontAsset.

```
%font = new TextSprite();
%font.Font = "ToyAssets:ArialFont";
```

Text служит для задания выводимого текста.

```
%font.Text = "yurembo";
```

Свойство FontSize задает размер для шрифта. Свойства FontScaleX и FontScaleY являются множителями для получения, соответственно, измененной ширины и измененной высоты шрифта. Эти ширина и высота влияют не только на размер букв, но и так же на промежутки между символами.

Свойства TextAlignment и TextVAlignment используются для выравнивания текста по горизонтали и вертикали относительно содержащего его прямоугольника. Для первого свойства используются такие значе-

ния: Left, Center, Right, Justify. Последнее значение используется, когда необходимо растянуть строку на весь вмещающий ее прямоугольник, чтобы текст полностью заполнил его. В качестве значений для свойства TextVAlignment используются: Top (выравнивание по верхней линии), Middle (выравнивание по середине), Bottom (выравнивание по нижней линии).

```
%font.TextAlignment = "Center";
%font.TextVAlignment = "Middle";
```

Свойства OverflowX и OverflowY настраивают поведение текста, когда он не влезает в предназначенный ему прямоугольник, соответственно, в ширину и высоту. Первое свойство имеет такие значения: Wrap, Visible, Hidden, Shrink. При использовании первого — часть текста, которая не влезает в строку, переносится на новую. Visible — лишний текст выходит за края прямоугольника. Hidden — лишний текст не отображается. Shrink — происходит изменение ширины символов таким образом, чтобы строка вошла в прямоугольник. У свойства OverflowY имеется 3 значения: Visible — текст выходит за границу прямоугольника и по прежнему виден; Hidden — скрывает лишние символы; Shrink — манипулирует высотой символов для того, чтобы все они вошли в прямоугольник по вертикали.

```
%font.OverflowModeX = "Visible";
%font.OverflowModeY = "Visible";
```

AutoLineHeight — отвечает за установку высоты всех символов равной высоте шрифта. Если отключена, высота символов подгоняется под настраиваемую линию высоты.

CustomLineHeight — значение не влияет на высоту шрифта, однако настраивает вертикальное расстояние между строками.

Kerning — расстояние между символами по диагонали.

Самая прекрасная особенность TextSprite — это возможность настраивать отображение и поведение отдельного символа. Для каждой буквы можно изменить ее цвет, масштаб, смещение и т.д. Для настройки свойств определенного символа используются методы класса TextSprite.

Чтобы сбросить настройки всех букв в начальные, надо вызвать метод ResetCharacterSettings.

Метод SetCharacterBlendColor устанавливает новый смешенный цвет для определенного символа. Первым параметром метод получает номер символа в строке (счет от 0), вторым — цвет.

Метод GetCharacterBlendColor возвращает смешенный цвет определенного символа в строке. В параметре метод принимает номер этого символа.

Метод GetCharacterHasBlendColor возвращает булево значение, которое положительно в случае, если определенный символ (под номером в параметре) имеет пользовательский цвет.

ResetCharacterBlendColor удаляет кастомный смешенный цвет с символа, номер которого передан в параметре. После этого этот символ окрашивается в смешенный цвет по умолчанию класса TextSprite.

SetCharacterScale — метод получает 3 параметра: номер символа, множитель по X и множитель по Y, он устанавливает масштаб определенного символа, используя множители.

GetCharacterScale — получает номер символа и возвращает его масштаб по обоим осям.

ResetCharacterScale — принимает номер символа и сбрасывает его масштаб к значению по умолчанию, т.е. 1.

SetCharacterOffset — задает смещение определенного в параметре символа по обоим осям.

GetCharacterOffset — возвращает смещение определенного символа.

ResetCharacterOffset — сбрасывает смещение для определенного символа.

ParticlePlayer

ParticlePlayer как и классы других объектов наследуется от SceneObject со всеми вытекающими отсюда последствиями. Объекты этого класса представляют собой эффекты частиц — испускатели частиц, если конкретнее. В качестве исходных данных используется ParticleAsset, который мы рассмотри в следующем разделе.

Свойство Particle устанавливает ассет для частиц:

```
%player = new ParticlePlayer();
%player.Particle = "ToyAssets:Bonfire";
```

В приведенном выше примере создается эффект частиц, для которого заполняется свойство Particle соответствующим ассетом. Также со свойством можно работать посредством методов.

Свойство CameraIdleDistance позволяет установить дистанцию между камерой и испускателем частиц. За пределами этого расстояния ParticlePlayer перейдет в режим ожидания. Находясь в нем, испускатель не выбрасывает частицы и не отображается.

ParticleInterpolation — включает и отключает интерполяцию частиц. Когда она включена непосредственно вычисляются позиция, поворот, размер точек между тиками.

Свойство EmmisionRateScale управляет фактором масштабирования частоты выброса для испускателя частиц.

Свойство SizeScale изменяет размер самих частиц.

ForceScale настраивает фактор изменения сил, действующих на испускатель частиц.

TimeScale влияет на время жизни (нахождения на экране) частиц.

Метод play активирует испускатель частиц. Метод stop, соответственно, останавливает выбрасывание. setEmitterPaused позволяет приостановить выбрасывание, с другой стороны, getEmitterPaused узнает состояние. setEmitterVisible устанавливает видимость испускателя, а getEmitterVisible возвращает состояние видимости. Методы: setPaused, getPaused позволяют, соответственно, установить и узнать о состоянии паузы испускателя.

Trigger

Объект класса Trigger не визуализируется, но от того, что унаследован от SceneObject, имеет все свойства последнего, в том числе позицию, угол поворота и прочее. К наследию класса SceneObject Trigger добавляет 3 свойства и 3 метода — функции обратного вызова. Они относятся к определению и реакции на столкновения.

EnterCallback — булево свойство, включение которого активирует работу события onEnter триггера.

```
%object = new Trigger();
%object.EnterCallback = "false";
```

StayCallback — свойство устанавливает триггер проверять событие onStay.

LeaveCallback — заставляет триггер проверять событие onLeave.

onEnter(%this, %object) — событие вызывается в момент, когда объект проникает в триггер, другими словами, когда впервые прикасается к нему.

onStay(%this, %object) — когда объект находится в границах триггера или частично в границах, это событие вызывается каждые 16 миллисекунд. Следовательно, вызов события происходит 64 раза в секунду, поэтому для сохранения высокой производительности, чтобы не нагру-

жать движок, обработчик должен быть максимально прост, то есть выполнять как можно меньше действий.

Обратный вызов onLeave(%this, %object) происходит, когда объект покидает пределы триггера, то есть только в тот момент, когда объект больше не находится в границах триггера.

Обратите внимание: созданный по умолчанию триггер не будет вызывать обработчики событий. Чтобы он стал это выполнять, для него надо создать коллизию.

2 Ассеты

В предыдущем разделе мы рассмотрели: какие классы игровых объектов есть в движке Torque 2D. Они — управляющие объекты движка и без внешних данных не представляют ничего интересного. Под внешними данными я имею в виду файлы: изображения, звуки, элементы интерфейса и т.д.

Чтобы не создавать графические материалы самостоятельно, можно воспользоваться теми, что располагаются в прилагаемых к книге материалах. Это свободный контент.

Основное понятие, связанное с контентом (другими словами: артом) игры в Torque 2D, является ассет. Ассет представляет собой специальным образом определенный элемент контента, который может быть загружен движком. Этими элементами как мы упоминали выше, могут быть: изображения, звуки и др. А их описание осуществляется с помощью знакомого нам языка TAML. В этом описание указывается имя загружаемого ассета, его модуль, подгружаемый файл контента, размеры изображения и многое другое в зависимости от типа ассета. В самой простой форме определение ассета может иметь следующий вид:

```
<ImageAsset
AssetName="background"
ImageFile="sky.png"
/>
```

В первой строчке указывается тип определяемого ассета, в данном случае — ImageAsset. Этот тип предполагает одиночное изображение. В будущем мы рассмотрим остальные типы ассетов, когда будем использовать их в своих играх. Во второй строчке указано имя ассета (значение свойства AssetName), которое будет использоваться непосредственно движком для доступа к ассету. Вообще, доступ к ассету осуществляется

через двухкомпонентное имя: сначала идет имя папки, в которой расположен модуль с описанием ассета, а затем непосредственное имя ассета — AssetName:«ToyAssets:background». Таким образом, директории, включающие модули, играют роль пространств имен, где в разных именных пространствах могут располагаться ассеты с одинаковыми именами.

Как же движок ищет и загружает ассеты? Во время старта движка, он запускает AssetManager, который в папке игры, а так же в ее подпапках ищет *.taml файлы и выполняет описанные там указания по передачи управления в скрипты. Вместе с тем в файле module.taml указано какие файлы надо искать для загрузки ассетов:

```
<DeclaredAssets
   Path="assets"
   Extension="asset.taml"
   Recurse="true"/>
```

Здесь указано, что описание ассетов находятся в подпапке assets, эти файлы имеют расширение «asset.taml». И указывается, что поиск необходимо проводить рекурсивно. Все найденные ассеты помещаются в специальную базу данных Torque 2D — AssetBase. Откуда они свободно могут быть использованы средствами языка Torque Script. Находящиеся в базе данных ассеты могут быть удалены, выгружены или переименованы.

ImageAsset

ImageAsset самый распространенный вид ассета. Он представляет способ обращения к изображению, которое может содержать любое количество условных кадров.

Ниже приведено описание ассета типа ImageAsset на языке TAML в формате XML. Я считаю этот формат удобнее, чем JSON, хотя он многословный, зато более описательный. Поэтому далее в книге я буду применять именно его.

```
<ImageAsset
    AssetName="angelFishImage"
    ImageFile="angelfish1.png"
    CellCountX="2"
    CellCountY="2"
    CellWidth="256"
    CellHeight="256" />
```

В 1-й строке задается тип ассет, во 2-й — определяется его имя, через которое мы можем обращаться к данному ассету из движка, в 3-й — используемое изображение — имя файла, в 4-й и 5-й — количество кадров в исходном изображении по вертикали и горизонтали, в 6-й и 7-й — ширина и высота кадров. В качестве исходного изображения могут быть представлены файлы широкого набора форматов.

AnimationAsset

Рассмотрим другой тип ассета — анимация. Torque 2D поддерживает 2 вида анимации: традиционный фреймовый (по кадровый) способ представления анимационной последовательности и скелетная анимация. При первом способе: изображение делится на кадры одинакового размера — атлас, а в сопровождающем файле описываются свойства: количество кадров по ширине и высоте, последовательность, скорость проигрывания и другое. Описание анимации так же содержит: ссылку на используемый графический ассет, порядок следования кадров, общую продолжительность анимации:

```
<AnimationAsset
    AssetName="triggerfish1Anim"
    Image="@asset=Content:triggerfish1"
    AnimationFrames="0 1 2 3 3 2 1 0"
    AnimationTime="0.4" />
```

Ссылка на графический ассет представляет имя уже определенного ассета.

Из дополнительных возможностей, AnimationAsset может именовать кадры. Это осуществляется с помощью свойства NamedAnmationFrames, одновременно, свойство NamedCellsMode должно быть включено — иметь значение true.

Свойство AnimationCycle включает или выключает зацикливание анимации, когда после проигрывания анимации один раз, она начинается сначала. По умолчанию свойство включено.

С включенным свойство RandomStart проигрывание анимации начинается со случайно выбранного кадра. По умолчанию: RandomStart = false;

С помощью метода getAnimationFrameCount можно получить количество кадров в анимационной последовательности.

SkeletonAsset

С другой стороны, скелетная анимация — это в рамках Torque 2D относительно новый вид анимации, так как добавлен только, начиная с версии 3.0. Для создания скелетной анимации используется специальное программное обеспечение. В пригодный для движка Torque 2D формат, созданные анимации экспортирует программа Spine (http://ru.esotericsoftware.com/) (рис. 4.1).

В данной книги мы не будем обсуждать способы и принципы создания скелетной анимации, сразу перейдем к загрузке готовой анимации в движок. Итак, Spine выдает 3 (или более, в зависимости от количества компонентов персонажа) файла: используемое PNG — изображение, содержащее компоненты персонажа, атлас для изображения (подобный ImageAsset в Torque 2D) и файл с описанием, в котором в JSON формате перечислены сведения об анимации: bone, skin. На основе этих файлов можно создать SkeletonAsset. Он является своего рода указателем для движка на использование этих файлов. Чтобы их активировать, достаточно заполнить только два поля, не считая имени ассета:

```
<SkeletonAsset
AssetName="goblins"
AtlasFile="goblins.atlas"
SkeletonFile="goblins.json"/>
```

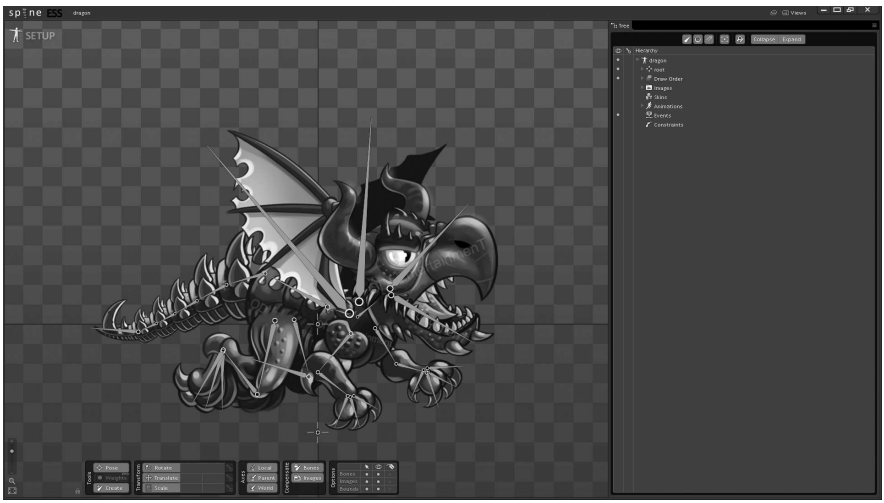

Рис. 4.1. Программа Spine

Для использования скелетной анимации применяется объект класса SkeletonObject, он рассмотрен в первом разделе данной главы.

FontAsset

Данный тип ассета был добавлен в версию движка 3.3. FontAsset представляет способ для загрузки растрового шрифта для возможности его использования объектом TextSprite (описан в 1-м разделе данной главы).

FontAsset представляет только одно свойство — FontFile, оно предназначено для указания ссылки на файл битового шрифта, который будет загружен движком. Ниже показан пример описания ассета:

```
<FontAsset
AssetName="ArialFont"
FontFile="Arial.fnt"
/>
```

Теперь шрифт можно использовать с помощью объекта TextSprite!

Для создания растрового шрифта используется программа Angle Code's Bitmap Font Generator (рис. 4.2).

В результате программа выдает растровую карту символов и конфигурационный файл с расширением fnt. В этом файле содержится ссылка на растр, описание карты символов и шрифта: в каких координатах карты какая буква находится. Рекомендуется создавать белые буквы на прозрачном фоне. Это позволяет использовать смешение цветов шрифта.

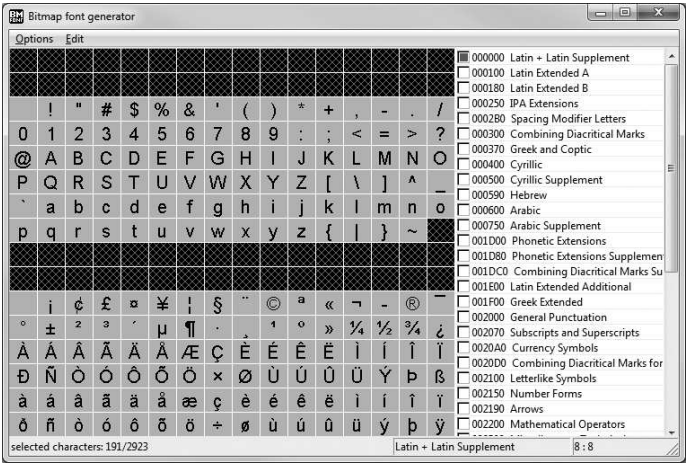

Рис. 4.2. Главное окно Angle Code's Bitmap Font Generator

ParticleAsset и ParticleAssetEmitter

Для представления сведений о частицах служит ParticleAsset. По существу, ParticleAsset является контейнером для ParticleAssetEmitter и должен включать хотя бы один из них. Испускатель является объектом, который выбрасывает бесконечное число частиц заранее определенным образом. Он обладает 71-м свойством для конфигурации эффекта частиц.

Чтобы разобраться с этим разберем гипотетический пример огня. Получается: в одном объекте ParticleAsset находятся два объекта ParticleAssetEmitter: для пламени и для дыма. Ниже показано описание данного ParticleAsset на языке TAML:

```
<ParticleAssetAssetName="Fire">
    <ParticleAssetEmitterEmitterName="Smoke">
    </ParticleAssetEmitter>
    <ParticleAssetEmitterEmitterName="Flames">
    </ParticleAssetEmitter>
</ParticleAsset>
```

Обратите внимание: этот эффект ничего не воспроизводит.

ParticleAsset и ParticleAssetEmitter содержат поля, значения которых влияют на эффект. Эти поля разделены на 2 группы. Первая группа содержит поля, которые не могут быть изменены на протяжении времени, но могут быть динамически изменены из скриптов.

Вторая группа содержит так называемые поля диаграммы. Такое поле может быть изменено автоматически с течением времени. Для этого применяется анимационный трюк — кадрирование изображений — установка ключей в определенные временные позиции. При этом каждый ключ содержит значение. Установка нескольких ключей на временной шкале позволяет изменять значение определенного параметра в связи с течением времени.

Таким образом, для изменения размера частицы по оси X с течением времени, мы можем написать такой TAML-код:

```
<SizeXLife>
    <KeyTime="0"Value="0"/>
    <KeyTime="0.5"Value="5"/>
    <KeyTime="0.9"Value="20"/>
    <KeyTime="1"Value="50"/>
</SizeXLife>
```

В начале выполнения размер (свойство SizeXLife) равно 0, через пол секунды — значение времени 0.5, размер равен 5, еще через 0.4 секун-

ды — при значении времени 0.9, размер равен 20, а когда сначала проходит секунда — значение временной шкалы 1, размер равен 50. Таким образом, параметры изменяются с течением времени.

В итоге к первой группе относятся не изменяемые во времени свойства, а ко второй — изменяемые. Существует большое количество первых и вторых. Так к неизменяемым во времени полям для объекта ParticleAsset относятся:

— LifeMode — устанавливает режим жизни для эффекта, другими словами, от режима зависит, что произойдет с эффектом, когда истечет его время; есть 4 режима: INFINITY — бесконечное существование, CYCLE — перезапуск эффекта после конца его времени жизни, KILL — удаление эффекта после истечения его времени жизни, STOP — по истечению времени эффект останавливается, фактически не удаляясь. По умолчанию действует режим INFINITY.

— LifeTime — время жизни эффекта, измеряемое в секундах.

К изменяемым во времени свойствам ParticleAsset относятся:

— LifetimeScale — как долго живет частица;

— QuantityScale — количество испускаемых частиц;

— SizeXScale — множитель размера частицы по оси X;

— SizeYScale — множитель размера частицы по оси Y;

— SpeedScale — множитель скорости частицы;

— SpinScale — множитель для вращения частицы;

— FixedForceScale — множитель постоянной силы, действующей на частицу;

— RandomMotionScale — множитель движения, действующий на частицу;

— AlphaChannelScale — управляет состоянием альфа-канала цвета частицы;

Неизменяемые свойства объекта класса ParticleAssetEmitter:

— EmitterName — имя испускателя;

— EmitterType — тип испускателя; существует 6 типов, от которых зависит регион испускания частиц: POINT — частицы создаются строго в указанной позиции; LINE — частицы создаются по всей длине определенной линии; BOX — частицы создаются внутри определенного куба; DISK — частицы создаются в диске с определенным радиусом; ELLIPSE — создание частиц происходит в эллипсе; TORUS — создание частиц осуществляет в торе с заданными минимальным и максимальным радиусом.

— EmitterOffset — задает смещение испускателя относительно позиции эффекта ParticlePlayer;

— EmitterAngle — угловое смещение испускателя относительно ParticleAsset;
— EmitterSize — размер испускателя;
— Image — задает ассет изображения, используемый для частицы;
— Frame — номер кадра из анимации;
— Animation — анимация для частицы;

Изменяемые во времени свойства для объекта класса ParticleAssetEmitter имеют те же названия, что для объекта ParticleAsset только без оконечного слова Scale и, соответственно, не являются множителями.

На этом мы завершим рассмотрение классов ParticleAsset и ParticleEmitterAsset.

3 Заключение

В процессе разработки игр на Torque 2D нам предстоит очень часто работать с ассетами, поэтому эта тема особенно важна и необходимо четко понимать работу с ассетами.

В этой главе мы рассмотрели 2 большие темы: игровые объекты и ассеты. Первыми являются всевозможные игровые сущности (спрайты, эффекты частиц, надписи и др.) и игровые механизмы (скроллеры, триггеры), а вторыми — вспомогательные механизмы для загрузки данных в движок.

В следующей главе мы рассмотрим физические свойства объектов, которые непосредственно влияют на взаимоотношения с миром игры. Эти свойства можно включать, выключать, настраивать числовые коэффициенты, словом, изменять физическое взаимодействие между динамическими объектами. Это очень важная тема, поскольку качество реализации физики в игре здорово влияет на ее восприятие пользователем.

Глава 5. Физические свойства и взаимодействия

Оглавление

Физика в играх довольно сложный вопрос. В упрощенном варианте, чтобы найти столкновение, нам надо вычислить соприкосновение и пересечение двух или более физических тел. Обработать столкновение и получить итоговый результат в виде какой-то ответной реакции или действия. Это много математики и вычислений. А столкновение — это только один из множества возможных вариантов физического взаимо-

действия. Еще есть: вращение, трение, сопротивление воздуха и воды, гравитация, взрывы, волны и так далее.

Однако Torque 2D упрощает физические вычисления до того, что нам не надо их производить! При создании объектов мы просто задаем их физические свойства и на основе этих значений, объекты участвуют в физических симуляциях. То есть по заданным свойствам движок выполняет вычисления, выдавая результирующие действия.

Для осуществления взаимодействий соответствующих законам физики в Torque 2D используется физический движок Box 2D, который получил право считаться стандартом де-факто в обработке физических взаимодействий в двумерном измерении.

Box 2D — движок для обработки физических взаимодействий между объектами, представляющими твердые тела. Он является проектом с открытым исходным кодом и распространяется под свободной лицензией. Изначально Box 2D был разработан Эрином Катто. Движок написан на языке C++ и является кроссплатформенным.

Первое правило Box 2D: все расстояния и размеры измеряются в метрах. Чтобы использовать возможности Box 2D в своих проектах, нам не понадобится напрямую применять этот движок. В Torque 2D имеется удобная физическая подсистема, построенная на основе Box 2D и включающая все свойства последнего.

Самое важное физическое понятие в Torque 2D относится к свойствам объектов, они могут быть трех типов: статические (static), динамические (dynamic) и кинематические (kinematic). Рассматривая в предыдущей главе класс Scene, я говорил про свойство gravity, задающее гравитацию для игрового мира. Гравитация воздействует только на динамические объекты. Такого типа объекты создаются по умолчанию.

Среди физических свойств, относящихся к объектам, присутствуют VelocityIteration и PositionIteration. Эти свойства управляют количеством итераций, выполняемых движком Box 2D для обработки физики за один тик. Если уменьшить их значение, возрастет общая производительность игры, однако уменьшится качество просчета физических эффектов. С другой стороны, если повысить значения этих свойств, качество физических эффектов возрастет, тем не менее снизится общая производительность Torque 2D. Поэтому значения, заданные по умолчанию в большинстве случаев оказываются самыми подходящими в отношении качество / скорость работы. Отсюда следует, что изменять их не стоит. Однако в вашей игре могут быть какие-то специфические случаи, тогда необходимо изменять и экспериментировать со значениями этих свойств. К примеру, если в ва-

шей игре очень много взаимодействующих частиц, которые настолько малы, что небольшая погрешность не имеет значения или попросту не заметна, в таком случае можно понизить значения этих свойств для увеличения общей производительности. Или, например, в вашей игре имеется сцена с замедлением времени, тогда наоборот, надо повысить значения свойств, чтобы сделать обработку физики наиболее реалистичной.

1 Обработка физических взаимодействий с помощью Box 2D

Как уже было сказано выше: для обработки физических взаимодействий в Torque 2D используется физический движок Box 2D. Но программисту, использующему Torque 2D, вовсе не придется проводить исследования в области нового движка (имеется в виду Box 2D), поскольку T2D предоставляет удобные обертки для работы с физическим движком. Настала пора поближе познакомиться с Box 2D.

Самый простой класс, представляющий игровой объект в Torque 2D — это SceneObject. Вся функциональность объектов Box 2D инкапсулируется в этом классе, и мы можем ей воспользоваться через API предоставляемым Torque 2D.

Каждый производный объект от класса, унаследованного от SceneObject, имеет позицию и ориентацию (угол поворота) в мировом (двумерном) пространстве, попросту говоря, в сцене (на экране). Это достигается путем получения каждым объектом SceneObject так называемого «тела» от Box 2D. Это происходит по умолчанию. И как раз, благодаря этому, объекты SceneObject (и все его наследники) имеют позицию и ориентацию, основываясь на которых, можно создать «форму столкновения», другими словами, оболочку, реагирующую на взаимодействия с другими объектами в сцене. Любое изменение физических свойств объекта в движке Torque 2D (в том числе изменение его позиции, угла поворота и др.) осуществляется путем модификации свойств физического тела Box 2D, которое происходит незаметно для программиста.

Физические типы объектов

Как я упомянул выше: в Torque 2D есть 3 типа объектов. Настала пора рассмотреть их более детально. Самый простой тип объектов — статический (static).

Static

Статические объекты не участвуют в физических симуляциях, следовательно, они не обрабатывают столкновения с любыми другими объектами. На статические объекты не действует никакая сила, в том числе гравитация, поэтому объекты данного типа неперемещаемые. Тем не менее программист может задать координаты и угол поворота статического объекта вручную. Поскольку статические объекты не участвуют в физическом взаимодействии, то они являются прекрасным видом для декоративного оформления игровых миров. К примеру, такие объекты могут служить для представления каких-то платформ в стрелялке, или игрового стола в карточной игре. Коротко говоря, выбор бесконечен!

Dynamic

Динамические объекты самые широко используемые в Torque 2D. Из-за этого по умолчанию созданный объект имеет данный тип. Динамические объекты могут взаимодействовать с объектами любых типов. Кроме того, они подвержены воздействию всех внешних сил, в том числе гравитации и сил линейного и углового ускорения.

Кроме использования настройки силы тяжести глобального Scene объекта, можно настроить гравитацию для конкретно SceneObject объекта с помощью метода setGravityScale данного класса, например: %object.setGravityScale(0.5); Из выполнения приведенного оператора следует, что сила тяжести к определенному объекту %object уменьшится вдвое. Точно для этой же цели можно изменять свойство GravityScale объекта класса SceneObject.

Так как на тип динамических объектов приходится наибольшее количество вычислений, то стоит создавать их в умеренных количествах, иначе есть риск привести свою игру к медленному выполнению, когда все будет тормозить.

Kinematic

В отличие от динамического типа на кинематические объекты не воздействуют силы, такие как гравитация. Они движутся в соответствии со своими линейным и угловым ускорениями. Кинематические объекты не взаимодействуют с объектами своего типа, при этом они могут сталкиваться с объектами динамического и статического типов.

В итоге имеем исчерпывающий набор типов объектов для физических симуляций (взаимодействий), которые можно широко применить в своих играх для создания красивых эффектов и занимательной механики!

Изменение типа объекта

Как было сказано выше: для изменения типа объекта достаточно в метод setBodyType объекта, чей тип надо изменить передать нужный тип: %object.setBodyType(static); или присвоить новый тип свойству BodyType данного объекта: %object.BodyType = static;.

Однако в таком случае необходимо помнить, что если для объекта нужно создать оболочку коллизии или рассчитать другие физические параметры, а так же обработать возможные столкновения, то в таком случае движку придется проделать большую работу, что скажется на производительности игры. Поэтому самый лучший сценарий — это создать объекты и предварительно настроить их свойства, не сваливая множество объектов в кучу, другими словами, не создавая несколько объектов в одних и тех же координатах. Так как в таком случае движку придется обрабатывать коллизии сразу всех этих объектов. Объекты попадают в игру вместе с чем на них начинают действовать силы, в том числе сила тяжести только после того, как созданные объекты будут добавлены в сцену (объект класса Scene). Поэтому перед этим действием надо настроить их свойства и корректные расположения.

2 Физические свойства объектов класса SceneObject и его потомков

После того как мы рассмотрели работу с физикой в Torque 2D, мы можем продолжить изучение свойств игровых объектов (в частности, класса SceneObject). Эти свойства рассматриваются только сейчас в виду их отношения к физике.

Булево свойство Active определяет: участвует ли данный объект в физическом взаимодействии независимо от его типа. По умолчанию имеет значение true, то есть участвует. Присвоить значение этому свойству можно непосредственно по его имени:

```
%object.active = false;
```

или с помощью метода setActive:

```
%object.setActive(false);
```

Получить его значение можно методом getActive:

```
%object.getActive();
```

Это свойство непосредственно управляет физическим телом объекта движка Box 2D. Кроме того, когда объект неактивен на него не воздействуют внешние силы, такие как: гравитация.

Когда Box 2D выполняет просчет симуляции и обнаруживает, что определенный объект не был перемещен и не участвует в каких-либо физических взаимодействиях, то движок помечает его спящим, то есть устанавливает его свойство Awake в значение false: %object.Awake = false; В этом состоянии объект требует к себе меньше внимания со стороны движка, а, следовательно, тратит меньше ресурсов вычислительной системы, повышая ее производительность. По умолчанию: объект автоматически просыпается при прямой модификации его пользователем, например, при изменении координат, или при попадании в коллизию. При этом засыпать объект умеет по умолчанию. Кроме того, программист может вручную «разбудить» объект, установив свойство Awake в true.

Вдобавок программист может отключить автоматический перевод объекта в режим сна, для этого надо присвоить свойству SleepingAllowed данного объекта значение false: %object.SleepingAllowed = false;

По умолчанию алгоритмы, используемые движком Box 2D предназначены для обработки объектов, движущихся с низкой и средней скоростью. В этом случае булево свойство Bullet сброшено. Вместе с тем данные алгоритмы не ресурсоемкие. С другой стороны, когда в игре есть быстро движущийся объект (например, пуля) стандартные механизмы обнаружения и обработки столкновений могут не сработать, что повлечет пролет объекта сквозь препятствие, будь то его не существует. Если в вашей игре присутствуют стремительно движущиеся объекты, то для них надо установить свойство Bullet в значение true. Тогда Box 2D будет проводить для них более точные расчеты и не позволит объектам миновать препятствие. Тем не менее в таком случае возрастает ресурсоемкость при выполнении физических вычислений, поэтому свойство Bullet надо включать только для немногих объектов, скорость движения которых по настоящему высока и наблюдаются проблемы при использовании стандартных алгоритмов обнаружения столкновений, что определяется опытным путем.

Для установки/отмены свойства Bullet достаточно написать: %object. Bullet = true;

3 Координатная система и силы в Box 2D

Начало координат в Box 2D и вместе с реализующим этот движок Torque 2D находится в центре экрана (0, 0). Таким образом, те объекты,

которые находятся левее центра, имеют отрицательные значения координаты X, а те, что правее — положительные. Подобным образом обстоят дела с осью Y: находящиеся ниже центра объекты имеют негативное значение координаты Y, а выше — положительное. Поэтому, например, чтобы поднять объект, надо переместить его в положительное направление оси Y. Измерение расстояния осуществляется в метрах.

Как мы уже обсуждали: пользователь движка может вручную — в коде установить расположение объекта: %object.Position = «20 40»;

Также пользователь может задать угол поворота объекта: %object. Angle = 45;

Между тем если просто установить значение позиции и/или угла поворота в коде, то в игре это приведет к резкой смене положения — рывком. Чтобы этого избежать, надо применить к объекту силу. В результате он плавно изменит свое положение. В физическом движке Box 2D главным образом присутствуют 2 силы: линейная и ангулярная (или угловая). Первая передвигает объект в положении плоскости, вдоль осей координат: X и Y. Вторая — отвечает за вращение объекта, то есть изменяет угол поворота вокруг оси Z. При воздействии силы объект перемещается плавно в зависимости от уровня примененной силы. Чтобы применить линейную силу к объекту, надо для него вызвать один из методов: setLinearVelocityX или setLinearVelocityX. В качестве одного параметра эти методы принимают количество силы примененной к соответствующей оси (зависит от метода). Кроме того, можно задать силы сразу по обеим осям, присвоив строку, состоящую из двух числовых значений свойству LinearVelocity: %object.LinearVelocity = «10 15»; Скорость перемещения объекта задается в метрах в секунду.

Сила (скорость) вращения объекта измеряется в градусах в секунду и устанавливается для объекта путем модификации значения свойства AngularVelocity: %object.AngularVelocity = 14; Также изменить ангулярную скорость объекта можно с помощью метода setAngularVelocity, в качестве параметра передав числовое значение — новую скорость вращения.

Torque 2D предоставляет функции для получения интерполированных значений позиции и угла объекта: getRenderPosition и getRenderAngle. Если объект не движется и/или не вращается, тогда функции, возвращающие обычные и интерполированные значения выдают один о тот же результат. Таким образом, имеет смысл ими пользоваться для движущегося объекта. Но все-таки эти функции редко используются и не представляют для нас особого интереса.

В Torque 2D есть большое число методов для изменения скоростей и сил объектов класса SceneObject и его потомков, ниже представлен список методов:

```
— setLinearVelocityX(velocityX)
— getLinearVelocityX()
— setLinearVelocityY(velocityY)
— getLinearVelocityY()
— getLinearVelocityFromWorldPoint()
— getLinearVelocityFromLocalPoint()
— setLinearVelocityPolar(angle, speed)
— getLinearVelocityPolar()
— setLinearDamping(damping)
— getLinearDamping()
— setAngularDamping(damping)
— getAngularDamping()
— applyForce(force, point)
— applyTorque(torque)
— applyLinearImpulse(impulse, point)
— applyAngularImpulse(impulse)
```

Преобразование систем координат

Любой объект, помещенный в сцену, имеет мировые и локальные координаты. Мировые координаты означают его расположение относительно сцены, а в локальных координатах описывается расположение субъектов относительно центра объекта. Здесь особенно важно понятие вектора — направление движения, то есть изначальное приращение по осям координат для объекта идентично его движению в соответствии с глобальной системой координат. Однако после того как угол поворота объекта будет изменен, изменится вектор его движения, и приращение по оси X будет перемещать его в направлении под соответствующим углом. К примеру, если объект повернуть на 90 градусов, то он изменит свое направление в локальной системе координат, тогда увеличение значений по оси X в его локальной системе координат будет перемещать объект по оси Y в мировой системе.

Для того, чтобы получить расположение в локальной/мировой системах координат в Torque 2D служат методы:

```
— getLocalPoint( worldPoint )
— getWorldPoint( localPoint )
```

А, чтобы получить локальный/глобальный вектора — служат методы:

```
— getLocalVector( worldVector )
```

```
— getWorldVector( localVector )
```

Еще одна важная возможность движка — это центр масс — довольно редко используемая и принимаемая во внимание возможность. Когда программист добавляет к SceneObject форму коллизии, вместе с тем добавляется масса к объекту. Поскольку форма коллизии не обязана быть симметричной (другими словами, она может отличаться от формы объекта), центр объекта не всегда является центром массы. Центр масс можно получить с помощью функций:

```
getLocalCenter()
getWorldCenter()
```

Эти методы применимы к SceneObject возвращают центр масс, соответственно, в локальных и мировых системах координат. Локальный центр масс возвращается относительно позиции объекта, тогда как мировой — относительно начала координат.

Центр масс взаимодействует с силами. Если применить линейную силу к центру масс, то объект просто будет двигаться в направлении применения силы, между тем, если применить силу к точке объекта, расположенной в стороне от центра масс, объект начнет вращаться.

4 Коллизии

Чтобы динамический объект взаимодействовал с другими объектами, для него необходимо создать коллизию. Она может быть разной формы. Для более точного определения столкновений она должна полностью обтекать объект, по своей форме максимально подходя к его форме.

Формы столкновения

Физический движок Box 2D обеспечивает обработку следующего списка форм коллизий:

1) Круг (Circle)
2) Полигон (только выпуклой формы) (Polygon)
3) Ребро (Edge)
4) Цепь (Chain)

Каждый SceneObject может содержать любое необходимое количество форм столкновений. Это позволяет создавать объекты с любым по сложности набором регионов коллизий. Их размеры определяются в метрах.

При изменении размеров объекта для соответствия его новым размерам его форму столкновений надо пересоздать, что накладывает на движок дополнительные вычисления, поэтому форма коллизия автоматически не меняет размер — не пересоздается для избежание дополнительной нагрузки на процессор, в виду чего, программист должен сам позаботиться о пересоздании формы коллизии, когда это станет необходимо.

Рассмотрим все 4 формы столкновений.

1) Круг — самая выгодная (в плане нагрузки на физический процессор) форма столкновения. Поскольку, вычисления столкновений между кругами, имеющими радиус больше 0, является вполне тривиальной задачей.

2) Полигон должен быть выпуклой формы и состоять не более чем из 8 вершин. Если потребуется, то максимальное количество вершин можно изменить, внеся модификации в исходный код движка, однако это делать не рекомендуется, так как повышение количества вершин вызовет замедление выполнения игры в связи с возросшей нагрузкой на движок, при этом выгода будет минимальна.

3) Ребро (или край) представляет собой одностороннее ребро, определяемое двумя точками (то есть представляет отрезок между ними). Оно может быть использовано в большом количестве случаев, например, представлять границы игрового поля.

4) Поведение цепи похоже на физическое взаимодействие с ребром, однако цепь (следуя своему названию) позволяет задать несколько продолжающихся отрезков столкновения.

Для создания разных форм коллизии у объекта класса SceneObject, а так же всех его потомков, имеется 5 методов:

```
1) createCircleCollisionShape( radius, [position] )
2) createPolygonCollisionShape( points )
3) createPolygonBoxCollisionShape(width, height,
[position, angle])
4) createChainCollisionShape( points,
[adjacentStartPoint], [adjacentEndPoint] )
5) createEdgeCollisionShape( startPoint, endPoint,
[adjacentStartPoint], [adjacentEndPoint] )
```

Каждый из них предназначен для создания определенной формы коллизии: первый создает коллизию в форме круга с заданным радиусом (передается в параметре); второй метод создает коллизию в форме полигона, в качестве параметров метод принимает координаты 4-х вер-

шин; третий метод тоже создает прямоугольную коллизию, принимает 2 параметра: длину и ширину; четвертый метод создает коллизию в виде цепи; пятый — создает ребро, принимая координаты его начала и конца.

Все перечисленные выше функции, в случае успеха возвращают уникальный и больший 0 индекс коллизии, а в случае неудачи: -1. Индекс коллизии имеет независимый от идентификаторов объектов счет. Индекс коллизии используется для обращения и управления коллизией.

Для удаления коллизии используется метод deleteCollisionShape, в качестве параметра он принимает индекс коллизии, которую надо удалить. Тем не менее коллизию совсем не обязательно удалять специально, она удалится автоматически вместе с материнским объектом SceneObject, для которого была создана.

У коллизии имеются дополнительные параметры, кроме ее формы: Density (плотность), Friction (трение), Restitution (восстановление). Для установки/возврата их значений у объекта класса SceneObject имеются специальные методы:

```
setCollisionShapeDensity(index, density)
getCollisionShapeDensity(index)
setCollisionShapeFriction(index, friction)
getCollisionShapeFriction(index)
setCollisionShapeRestitution(index, restitution)
getCollisionShapeRestitution(index)
setCollisionShapeIsSensor(index, isSensor?)
getCollisionShapeIsSensor(index)
```

Каждому из них на месте параметра index передается индекс коллизии, свойства которой надо изменить/получить. По умолчанию указанные выше свойства имеют такие значения:

Density = 1.0

Friction = 0.2

Restitution = 0.0

У разных по форме коллизий есть большой набор методов для возвращения определенных сведений о коллизии, каждый из них в качестве единственного параметра принимает индекс коллизии.

С помощью метода getCircleCollisionShapeRadius можно узнать радиус круговой коллизии, а с помощью getCircleCollisionShapeLocalPositi on — ее локальные координаты. Методом getPolygonCollisionShapePoin tCount возвращается количество точек, составляющих полигон. Методом getPolygonCollisionShapeLocalPoint получаем позицию полигонной коллизии. С помощью функции getChainCollisionShapePointCount воз-

вращается число точек, входящих в цепь; getChainCollisionShapeLocalPoint — получаем позицию последней.

Функция getCollisionShapeType, принимающая индекс коллизии, возвращает тип коллизии. Как мы помним: существует 4 типа коллизии, тип возвращается в виде строки:

— circle
— polygon
— edge
— chain

Есть возможность скопировать одну или все формы коллизий от одного SceneObject к другому. С помощью метода copyAllCollisionShapes(%targetSceneObject, [clearTargetShapes?]) происходит копирование всех коллизий в пользу указанного в первом параметре объекта. А метод copyCollisionShape(%sourceShapeIndex, %targetSceneObject) копирует только коллизию под индексом, указанным в первом параметре для объекта — во втором параметре.

Управление столкновениями

Когда сталкиваются 2 объекта, происходит «контакт». «Контакт» создается в момент первого соприкосновения объектов и уничтожается, когда они расходятся. Настройка столкновений заключается в фильтрации объектов, которые могут контактировать друг с другом. Эта фильтрация выполняется путем присвоения номеров экранных групп и слоев, а так же слоев столкновения. Каждый объект SceneObject имеет определенный экранный слой — SceneLayer с номером от 0 до 31, ровно, как и экранную группу (SceneGroup). Номер SceneLayer — это порядок визуализации — номер слоя отдаленности от камеры, то есть находящиеся на 0-м слое объекты расположены ближе к камере, отображаются поверх объектов, расположенных на других слоях, — перекрывают их. Соответственно, объекты, находящиеся на 31-м слое — это самые дальние объекты. Помещение объектов в одну экранную группу (SceneGroup с одним и тем же номером) верный способ группировки объектов. Это свойство может быть использовано при выборе объектов SceneObject из сцены (Scene). Вдобавок у объекта класса SceneObject есть свойство: группы коллизии — CollisionGroups и равнозначное ему: слои коллизии — CollisionLayers. Номера обеих групп находятся в диапазоне от 0 до 31.

После создания объект находится в 0-м экранном слое и в 0-й экранной группе. В каждый момент времени объект может находиться только

в одной экранной группе и одном экранном слое. Но при этом находиться сразу в нескольких слоях и группах столкновения.

Для изменения групп и слоев для объекта SceneObject может быть использован следующий код:

```
%obj = new SceneObject();
// для экранного слоя
%obj.setSceneLayer(10);
// для экранной группы
%obj.setSceneGroup(10);
// для слоев коллизии
%obj.setCollisionLayers( 5, 6, 7 );
%obj.setCollisionLayers( "5 6 7" );
%obj.CollisionLayers = "5 6 7";
// для групп коллизии
%obj.setCollisionGroups( 20, 30 );
%obj.setCollisionGroups( "20 30" );
%obj.CollisionGroups = "20 30";
```

После создания SceneObject сталкивается с любым объектом, не обращая внимания на разницу в группах и слоях. Для этого существуют специальные режимы, они устанавливаются следующим образом:

```
%obj = new SceneObject();
// выбрать все слои коллизии
%obj.setCollisionLayers();
%obj.setCollisionLayers( all );
// выбрать все группы коллизии
%obj.setCollisionGroups();
%obj.setCollisionGroups( all );
// не выбирать ни одного слоя коллизии
%obj.setCollisionLayers( none );
%obj.setCollisionLayers( off );
// не выбирать ни одной группы коллизии
%obj.setCollisionGroups( none );
%obj.setCollisionGroups( off );
```

Можно свободно настраивать слои и группы коллизии, учитывая, что объекты, находящиеся в одной группе будут сталкиваться, а в разных, соответственно, не будут. Также можно использовать только CollisionGroup или только CollisionLayers.

События столкновений

Как мы упоминали в прошлом подразделе: при столкновении двух тел, физическая подсистема регистрирует «контакт», что является запи-

сью о столкновении, которая существует пока эти тела соприкасаются. Когда «контакт» создан, Box 2D генерирует событие. Чтобы включить для объекта функцию обратного вызова и последующую передачу управления в скриптовый код, надо выполнить команду:

```
%obj.setCollisionCallback(true);
```

После этого произошедшие в движке события будут вызывать обработчики событий из скриптового кода. Есть 2 вида обработчиков событий: в первом случае вызывается обработчик глобального объекта Scene, посмотрим на него:

```
function Scene::onSceneCollision( %this,
%sceneObjectA, %sceneObjectB, %collisionDetails )
{
}
```

Параметры: %this — ссылка на глобальный объект Scene, %sceneObjectA — первый объект — участник столкновения, %sceneObjectB — второй объект — участник столкновения, %collisionDetails — дополнительные сведения о столкновении.

Главной особенностью этого обработчика является то, что он один служит для обработки столкновений между двумя любыми объектами. В то же время это является и главным минусом такой организации обработки столкновений, поскольку объекты разных классов могут по разному реагировать на столкновения, но обработчик то у нас один, таким образом, до его вызова мы не знаем класс, к которому принадлежит объект, следовательно, не можем написать конкретную функцию, а один обработчик может разрастись до гигантских размеров.

С другой стороны, во втором случае обработчик вызывается для объекта SceneObject — инициатора столкновения. Прототип обработчика выглядит следующим образом:

```
function SceneObject::onCollision( %this,
%sceneObject, %collisionDetails )
{
}
```

Параметры: %this — ссылка на первый объект — участник столкновения, %sceneObject — ссылка на второй объект — участник столкновения, %collisionDetails — дополнительные сведения о столкновении.

Преимущество этого способа организации обработчика столкновений заключаются в том, что мы можем написать определенный обработчик для конкретного класса объектов. После возбуждения события Торк уже знает какой обработчик надо вызывать, а в конкретном обработчике уже написан код для обработки столкновений определенного класса объектов. Таким образом, мы можем создать новый класс:

```
%sceneObject = new Sprite()
{
    class = "Bullet";
};
```

И написать обработчик конкретно для него:

```
function Bullet::onCollision(%this, %sceneObject,
%collisionDetails)
{
}
```

Как и ожидается, этот обработчик вызовется, когда объект Bullet столкнется с другим объектом в зависимости от фильтрации — номера слоя. Это является главным преимуществом данного способа обработки столкновений: вызов обработчика происходит только для определенного объекта.

С другой стороны, недостатком этого способа является то, что если в столкновении участвуют 2 объекта одного и того же класса, тогда обработчик вызовется 2 или более раз, что может отрицательно сказаться на производительности, поэтому надо принимать взвешенные решения при написании кода для обработчика, чтобы избежать лишних действий.

Сведения о столкновении

В параметрах обоих обработчиков присутствует переменная %collisionDetails. В ней содержаться дополнительные сведения о столкновении. Для разных типов обработчиков она имеет разный формат — разное количество содержащихся данных:

1. Индекс коллизии объекта A (1-й объект)
2. Индекс коллизии объекта B (2-й объект)
3. Нормаль коллизии
4. Точка контакта в мировых координатах для 1-го объекта
5. Нормализованный импульс столкновения для 1-го объекта
6. Касательный импульс столкновения для 1-го объекта

Далее следуют параметры, содержащиеся в общей версии обработчика столкновения:

7. Точка контакта в мировых координатах для 2-го объекта
8. Нормализованный импульс столкновения для 2-го объекта
9. Касательный импульс столкновения для 2-го объекта

Рассмотрим эти данные:

Индексы коллизий указывают на формы коллизий, участвующие в столкновении. Можно использовать эти индексы, чтобы узнать о свойствах форм коллизий: трение, восстановление и т.д.

Нормаль относится к направлению, в котором импульс будет применен к разделению двух форм коллизии.

Точка контакта в мировых координатах — используется для вычисления нормали коллизии и импульса разделения.

Нормализованный импульс является импульсом силы, используемым для разделения объектов во время столкновения.

Касательный импульс является импульсом силы для вычисления трения.

Из всего этого набора записей гарантированно возвращаются только первые две. Например, в случае столкновения двух сенсорных объектов, Box 2D не возвращает дополнительных сведений, таких как: контактные точки. То есть в таком случае известны только индексы форм коллизий.

Завершение столкновения

В момент, когда объекты заканчивают соприкосновение — выходят из области друг друга и начинают отдаляться, в движке происходит вызов обработчика события конца столкновения. Как и при обработке начала столкновения, при обработке завершения столкновения имеется 2 способа. Один — для глобального объекта сцены — Scene:

```
function Scene::onSceneEndCollision( %this,
%sceneObjectA, %sceneObjectB, %collisionDetails )
{
}
```

и второй — для конкретного объекта SceneObject:

```
function SceneObject::onEndCollision( %this,
%sceneObject, %collisionDetails )
{
}
```

В остальном, эти обработчики аналогичны рассмотренным выше.

В действительности физическая подсистема не знает о том, что один объект столкнулся с другим, она знает: произошел контакт между двумя объектами, в результате чего для обоих объектов надо вызвать обработчики данного события.

5 Заключение

В подходящей к концу главе мы рассмотрели многие физические свойства и механизмы движка Torque 2D: систему координат физического движка Box 2D, силы, гравитацию, ускорения, коллизии, их формы, события, сведения о них и мн. др. Также мы увидели как все это соотносится с объектами движка, обсудив это на основе базового класса для игровых объектов SceneObject. Хотя для наших первых игр на движке Torque 2D знаний о реализации в нем физики достаточно, эта тема пока не закрыта, в будущих главах мы еще вернемся к физике и ее реализации для использования новых механизмов.

Со следующей главы мы начнем разрабатывать уже достаточно большие игры, мы будем оттачивать свое мастерство в программировании и использовании движка Torque 2D, который поможет нам в нашем деле — в разработке игр!

Глава 6. Разработка аркадной игры Asteroids

Оглавление

1 Введение

Мне всегда был интересен космос. Но с полетами в космическое пространство не сложилось, зато с помощью игр я могу путешествовать меж звезд сколько угодно. Или что, конечно же, более интересно, я могу разработать свою галактическую экспедицию! И это может осуществить любой игродел!

Я начал писать эту главу в День космонавтики, поэтому решил посвятить ее около космической теме.

Впервые выпущенная на игровых автоматах в далеком 1979-ом году аркадная игра Asteroids не только благополучно дожила до наших дней, имея порты под все возможные игровые платформы, но и по праву стала считаться классикой игр. В этой главе мы с вами разработаем клон этой захватывающей игры на движке Torque 2D изначально для платформы IBM PC, а впоследствии эту игру можно портировать под любую, поддерживаемую Торком, платформу. Мы с вами вместе осуществим такое портирование для планшета под управлением операционной системой Android. Но это все планы на будущее, а пока нам необходимо создать саму игру.

2 Подготовительный этап

Перво-наперво разработка игры начинается с написания дизайн документа. Как вы знаете: в игровой индустрии этим занимается гейм-дизайнер. Он придумывает концепцию будущей игры, ее правила, стиль, определяет жанр, восприятие игроков и другое. Все это описывается в дизайн-документе. По идее, мы тоже должны выделить для этого время, тем не менее, поскольку правила игры Asteroids широко известны, а так же потому, что читаемая вами книга посвящена, в общем-то, программированию, мы ограничимся кратким описанием.

Дизайн документ для игры Asteroids

Игрок управляет космическим кораблем, который может перемещаться по экрану/окну, движение корабля ограничено рамками окна. В арсенале корабля есть два вида оружия: пули и лазер. Если пули бесконечны, то лазер имеет ограничение на количество выстрелов, после чего ему надо «восполниться». В игре присутствуют два типа врагов: дрейфующие в космосе астероиды, имеющие разные размеры и скорости, и летающие тарелки, всегда преследующие космический корабль. Если

пулей попасть в астероид, то последний расколется на два более мелких куска, которые продолжат дрейфовать. Следующим попаданием маленькие кусочки уничтожаются. С другой стороны, если по астероиду попасть лазером, то первый будет уничтожен с одного выстрела.

Также для более глубокого погружения пользователя в игру, надо реализовать эффекты. Например, системы частицы, возникающие при взрыве, отобразят присущий ему эффект; текстовые надписи покажут набранные игроком очки, увеличение которых происходит в момент поражения астероидов и тарелок. Также нужна надпись для вывода количества лазерных снарядов, оставшихся у корабля.

Космический корабль будет терпеть крушение в двух случаях: при столкновении с астероидом и тарелкой. В таком случае должно появиться окно, в котором надо вывести набранные очки и дать выбор для игрока между стартом новой игры и выходом.

Такого описания нам будет достаточно, чтобы представить игру, которую мы хотим разработать. Что дальше?

У нас есть диз-док (иными словами: план для разрабатываемой игры), превосходный движок, но для создания игры этого не достаточно. И мы столкнулись перед самым сложным вопросом, появляющимся перед инди-разработчиком в начале разработки игры: где взять контент: изображения, текстуры, модельки, анимации, одним словом — арт? Можно попросить друзей — дизайнеров нарисовать нужный арт, но все это может затянуться на невыносимо длительный срок. Поэтому я предлагаю воспользоваться артом с сайта github.com, который можно свободно использовать в целях самообразования. Кроме того, этот арт можно взять среди материалов книги. И я рекомендую именно последний вариант, так как дополнительно к имеющемуся арту для своей игры я нарисовал еще несколько изображений, без которых картина игры Asteroids, по-моему, не полная.

3 Создание проекта Asteroids

Обращаю ваше внимание: для каждого нового проекта мы будем подготавливать папку с чистой версией движка Torque 2D 3.3. Например, можно скачивать движок с помощью Git Bash, как это описано в главе 2 в разделе 2 «Загрузка и установка движка». Только скачивать надо каждый раз в новую папку, иначе измененные вами файлы будут перезаписаны!

Подготовленный арт прекрасно подходит для планируемой игры, среди файлов есть: космический корабль, астероиды, космическая та-

релка, фон, представляющий космическое пространство, различные эффекты, в том числе: взрывы, частицы, снаряды, лазеры, специально подготовленный шрифт, а так же модули описания (*.taml) изображений, систем частиц и анимаций.

Итак, давайте начнем создавать наш проект. О том, как создать проект на движке Torque 2D с нуля я подробно рассказал в главе 3. Поэтому здесь я дам лишь короткие рекомендации. Первое — это создания папок и размещения в них арта. Обсудим итоговое размещение файлов.

В папке C:\Torque\ создайте подкаталог Asteroids, в котором создайте подпапку modules. Это то место, где будут находиться все вспомогательные для игры файлы, как то: арт и скрипты. На уровне выше находится только исполняемый модуль движка и файл main.cs, во время разработки еще будут находиться файлы Torsion-проекта, в случае использования данной среды программирования. Эти файлы можно взять из материалов к книге или подготовить самостоятельно, как об этом рассказывается в главе 3, поэтому здесь я не буду повторяться. Содержимое подкаталога AppCore папки modules вполне стандартно и, также подробно рассмотрено в вышеупомянутой главе. В подкаталоге ToyAssets располагается арт для игры вместе со скриптами. Они предназначены для описания графических материалов. Когда мы займемся настройкой игры, то рассмотрим их. Подпапка AsteroidsGame главным образом содержит геймплейные скрипты, описывающие поведение игровых объектов. Кроме того, в ней есть профиль пользовательского интерфейса и недостающее изображение, представляющее собой космический корабль, управляемый пользователем.

Для запуска разрабатываемой игры нам понадобится откомпилированный исполняемый модуль движка (exe — файл), он может называться любым именем, например, Asteroids.exe.

4 Кодирование

Настала пора перейти к основному этапу разработки — кодированию. Вы можете вводить код самостоятельно, вчитываясь в книгу, и, следя за разъяснениями автора, а можете постепенно копировать скрипты в свой проект, одновременно, читая текст и разбираясь в коде. Какой вариант работы с книгой вы бы не выбрали, чтобы не потерять нить повествования просматривайте код и выполняйте упражнения.

Я не буду повторять относительно кода заготовки проекта, поэтому, если вам нужна эта информация, загляните в 3-ю главу.

Для начала создайте файл Torsion-проекта. И откройте его в редакторе Torsion, кроме того скопируйте из нашей заготовки main.cs файл и исполняемый модуль движка. По идее, если заготовка выполнена правильно, на ней может быть основана любая игра, в том числе и та, которую мы разрабатываем в данный момент, поэтому скопируйте из нашей заготовки подпапку AppCore каталога modules в соответствующее место игры Asteroids. Подкаталог с артом (ToyAssets) уже, надеюсь, скопирован из материалов книги к игре Asteroids (об этом я упоминал выше). Переходим в ключевую подпапку разрабатываемой игры AsteroidsGame. В ней нужен taml-модуль, с которого начнется загрузка содержимого этой папки. Подобные файлы мы уже создавали в 3-й главе, поэтому повторяться не буду (для освежения памяти можно посмотреть материалы к данной главе). Дополнительно здесь нужен файл main.cs. Мы уже сталкивались с созданием такого файла для конкретного проекта myProject, но здесь — в конструкторе класса AsteroidsGame этого файла подключаются дополнительные файлы и вызываются инициализирующие функции для объектов новых классов. Ниже приведен данный конструктор:

```
function AsteroidsGame::create( %this )
{
  exec("./scripts/scenewindow.cs");
  exec("./scripts/scene.cs");
   exec("./gui/guiProfiles.cs");
   exec("./scripts/background.cs");
   exec("./scripts/spaceship.cs");
   exec("./scripts/asteroids.cs");
   exec("./scripts/controls.cs");
   exec("./scripts/bullet.cs");
   exec("./scripts/Explosion.cs");
   exec("./scripts/UFO.cs");
   exec("./scripts/ray.cs");
   exec("./scripts/TextLabs.cs");
   exec("./scripts/consts.cs");
   exec("./scripts/GameOverScreen.cs");

   setRandomSeed(getRealTime());
    createSceneWindow();
       createScene();
   mySceneWindow.setScene(myScene);
   createBackground();
   createSpaceShip();
   createAsteroids($asteroidsMaxCount);
   createUfo();
```

```
  new ScriptObject(InputManager);
  mySceneWindow.addInputListener(InputManager);
       InputManager.Init_controls();
   InputManager.dirControls();
}
```

Функцией setRandomSeed устанавливается значение для генератора случайных чисел, ей передается аргумент, возвращенный функцией getRealTime, который представляет количество миллисекунд, прошедших с момента загрузки операционной системы. Выше находятся подключения файлов с кодом, а ниже вызываются функции, составляющие каркас игры, а так же функции для создания объектов.

Функции createSceneWindow и createScene нам уже знакомы, так как они включены в заготовку проекта. Они создают, соответственно, окно вывода и игровую сцену. createBackground создает объект фона; createSpaceShip создает объект космического корабля, управляемый игроком; createAsteroids создает астероиды в количестве, переданном в параметре; createUfo создает объект вражеского НЛО.

Ниже по коду в строке: new ScriptObject(InputManager); создается объект InputManager класса ScriptObject. Этот класс позволяет создавать пользовательские объекты с дополнительной нагрузкой (свойства и методы). То есть, по сути, он представляет пустой объект-контейнер, не имеющий никакой функциональности. Затем ее добавляет программист, создавая свои классы на основе ScriptObject. Между тем этот класс реализует функции обратного вызова: onAdd и onRemove, вызываемые, соответственно, при создании и удаления объекта данного и/или унаследованного класса. Следующей строчкой объект InputManager устанавливается как прослушиватель событий ввода для главного окна вывода. Далее вызываются методы для инициализации средств ввода. Мы рассмотрим их позднее в соответствующем контексте.

Во время закрытия приложения разрушается главный объект игры AsteroidsGame, это осуществляется в его деструкторе:

```
function AsteroidsGame::destroy( %this )
{
   shipcontrols.pop();
   destroySceneWindow();
   InputManager.delete();
}
```

Здесь происходят 3 действия: из стека выталкивается карта контроллера — элемент управления космическим кораблем, другими словами:

происходит его уничтожение; вызывается функция уничтожения окна вывода, а последним действием удаляется менеджер управления.

А теперь рассмотрим игровые сущности подробно.

Константы и глобальные переменные

Прежде, чем перейти к игровым объектам нам необходимо взглянуть внутрь файла consts.cs, где определены все константы и глобальные переменные, используемые в игре. По сути, в Torque Script нет констант как таковых, однако я применяю это название с учетом того, что их значения никогда не будут изменены. По названиям каждой константы (или переменной) интуитивно понятно для чего она служит, какую роль играет и какое значение хранит. Если что-то сначала не понятно, то дальше в коде будут пояснения, где используются соответствующие константы и переменные.

```
$screenWidth = 100;
$screenHeight = 75;
$timeSchedule = 100;
$turnSpeed = 100;
$asteroidsMaxCount = 3;
$asteroidsCount = 0;
$asteroidsScore = 10;
$ufoScore = 30;
$maxRayAmmo = 10;
$rechargeRayDelay = 2000;
//Scene Groups
$asteroidGroup = 20;
$bulletGroup = 21;
$borderGroup = 22;
$spaceShipGroup = 19;
$ufoGroup = 23;
$rayGroup = 24;
//ufo
$ufoWidth = 128;
$ufoHeight = 64;
$ufoSpeed = 10;
$ufoCount = 0;
$ufoMaxCount = 3;
$ufoCreateDelay = 5000;
//labels in a game
$scoreLab = 0;
$ammoLab = 0;
$finalScoreLab = 0;
//GameOverScreen size
$GameOverScrWidth = 50;
```

```
$GameOverScrHeight = 37.5;
//turn triggers
$turnLeft = 0;
$turnRight = 0;
```

5 Фон

Объект — фон описан в отдельном скрипте, который подключается в конструкторе AsteroidsGame командой exec. Этот файл включает только одну функцию. Она служит для создания фона, давайте рассмотрим ее подробнее:

```
function createBackground()
{
    %background = new Sprite();
    %background.Class = "Background";
    %background.setBodyType( static );
    %background.Position = "0 0";
    %background.Size = $screenWidth SPC $screenHeight;
    %background.SceneLayer = 31;
    %background.Image = "ToyAssets:skyBackground";
    %background.createEdgeCollisionShape(-50, -37.5,
-50, 37.5);
    %background.createEdgeCollisionShape(50, -37.5,
50, 37.5);
    %background.createEdgeCollisionShape(-50, 37.5,
50, 37.5);
    %background.createEdgeCollisionShape(-50, -37.5,
50, -37.5);
  %background.setCollisionGroups($asteroidGroup,
$spaceShipGroup, $ufoGroup);
    %background.setSceneGroup($borderGroup);
    %background.setDefaultRestitution(0.5);
    myScene.add(%background);
}
```

В локальной переменной %background мы сохраняем идентификатор созданного спрайта, затем, используя этот идентификатор, мы настраиваем свойства для спрайта. По сути, фон представляет собой прямоугольник с наложенной текстурой и растянутый на весь экран (или окно, в случае оконного приложения). Задаем для фона отдельный класс: %background.Class = «Background»; он понадобится нам в будущем. Для прямоугольника — фона задаем статический тип взаимодействия с окружающими объектами (метод setBodeType с параметром static), устанавливаем фон по середине экрана (свойство Position), раз-

мер равный ширине и высоте области вывода (свойство Size). Помещаем фон (свойство SceneLayer) на самый последний 31-й (счет с нуля) слой визуализации, таким образом любой объект, находящийся на слое выше будет перекрывать фон, что и требуется. В качестве значения свойства Image мы передаем идентификатор графического ассета ToyAssets:skyBackground, который описан на языке TAML.

Давайте опишем данный ассет. Но сперва разместим нужный арт в положенном месте: в папку Asteroids\modules\ToyAssets\1\assets\images\ поместим файл skyBackground.png. Рядом создадим файл skyBackground.asset.taml. Откроем его в любимом текстовом редакторе (например, NotePad++), напишем такой код:

```
<ImageAsset
    AssetName="skyBackground"
    ImageFile="skyBackground.png" />
```

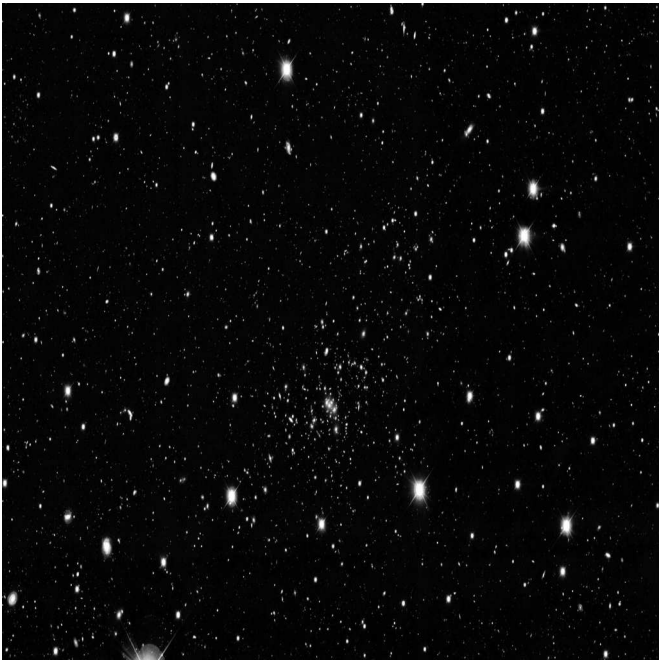

Рис. 6.1. Фон

Тем самым создается ссылка, которая указывает на файл skyBackground.png доступный для загрузки движком.

С помощью вызовов функции createEdgeCollisionShape создаются четыре границы. В параметрах функция получает координаты воображаемого отрезка (его вершин), представляющим границу. После этого функцией setCollisionGroups задается: какие группы объектов будут взаимодействовать с границами объекта — фона. В нашей игре имеется три группы объектов: в группу $asteroidGroup входят все астероиды, в $spaceShipGroup попадает пользовательский космический корабль, напоследок, в группу $ufoGroup входят вражеские НЛО. Все перечисленные группы объектов не должны покидать область обзора. Далее командой setSceneGroup мы присваиваем фону и границам, созданным для фона, номер группы экранных объектов, он передается в функцию параметром — глобальная переменная $borderGroup. Функцией setDefaultRestitution устанавливается сила взаимодействия внешних объектов с границами экрана, другими словами, степень «пружинистости» при соударении объектов с границами. Последняя строчка, рассматриваемого листинга, добавляет объект — фон в стек сцены. В будущем мы еще вернемся к этой функции, чтобы дополнить ее созданием других объектов.

Если сейчас запустить игру на выполнение, нажав F5 в Torsion , или, запустив исполняемый файл игры, то на фон будет наложена текстура космического пространства.

6 Космический корабль

Первым делом подготовим ассет космического корабля. В подпапку Asteroids\modules\AsteroidsGame\assets\ поместим файл ship4.png, а в сопровождающий текстовый файл ship4.asset.taml напишем:

```
<ImageAsset
    AssetName="LoRez_SpaceShip"
    ImageFile="ship4.png" />
```

Рис. 6.2. Космический корабль

Добавим в подпапку Asteroids\modules\AsteroidsGame\scripts\ файл spaceship.cs, он подключается в файле main.cs. В самом начале файла находится глобальная функция createSpaceShip, служащая для создания космического корабля. Эта функция для того и сделана глобальной, чтобы ее можно было вызвать тогда, когда объект корабля еще не создан.

```
function createSpaceShip()
{
    %spaceship = new Sprite(PlayerShip);
    %spaceship.setBodyType(dynamic);
    %spaceship.Position = "0 0";
    %spaceship.Size = "4 4";
    %spaceship.Image = "AsteroidsGame:LoRez_
SpaceShip";
    %spaceship.createPolygonBoxCollisionShape();
    %spaceship.setCollisionCallback(true);
    %spaceship.setFixedAngle(true);
    %spaceship.isThrusting = false;
    %spaceship.score = 0;
    %spaceship.rayAmmo = $maxRayAmmo;

    %spaceship.setCollisionGroups($borderGroup,
$spaceShipGroup,          $asteroidGroup, $ufoGroup);
    %spaceship.setSceneGroup($spaceShipGroup);
    %spaceship.setCollisionLayers(1);
    %spaceship.setSceneLayer(1);

    myScene.add(%spaceship);
}
```

Сначала мы создаем объект — Sprite, унаследованного класса PlayerShip. Делаем объект динамическим, даем ему возможность участвовать в физических взаимодействиях. Затем размещаем его в центре экрана. Устанавливаем размер 4 на 4 торковских метров. Изображение для корабля загружаем из ассета: «AsteroidsGame:LoRez_SpaceShip»; С помощью метода createPolygonBoxCollisionShape по периметру спрайта космического корабля создается квадратный объект коллизии, таким образом именно этот объект будет участвовать в столкновениях. Следующим методом включаем срабатывание процедур обратного вызова при возникновении событий столкновения. Устанавливаем использование закрепленных углов. Строкой %spaceship.isThrusting = false; добавляем к объекту класса PlayerShip новую булеву переменную-член, которая принимает истинное значение, когда космический корабль ускоряется — летит вперед. Также добавляем атрибут score, где будут

храниться набранные за игру очки. Третья переменная-член, которая нам нужна — rayAmmo будет хранить количество лазерных снарядов. Обращаю внимание: эти 3 переменные объявлены только в скриптах и могут использоваться таким же образом, как объявленные в движке, но только в рамках данного проекта, если, конечно, они не будут объявлены в другом проекте. С помощью метода setCollisionGroup мы объявляем группы объектов, с которыми может происходить физические взаимодействия — столкновения. Нужные группы перечисляются в качестве параметров через запятую. Следующим действием мы присваиваем номер группы экранных объектов. Метод setCollisionLayers устанавливает номер слоя коллизии. Метод setSceneLayer устанавливает: на каком экранном слое будет отображаться данный объект. В конце мы добавляем созданный и инициализированный объект на экран — в стек объектов для визуализации myScene.

Движения космического корабля

Следующий метод, который мы рассмотрим, вызывается при движении корабля вперед, то есть при ускорении:

```
function PlayerShip::accelerate(%this)
{
   %adjustedAngle = %this.Angle;
   %adjustedAngle += 90;

   if(%this.isThrusting) %ThrustVector = Vector2Direct
ion(%adjustedAngle, 35);
   else %ThrustVector = Vector2Direction(%adjustedAng
le, 95);
   %this.setLinearDamping(0.0);
   %this.setAngularDamping(2.0);
   %this.applyLinearImpulse(%ThrustVector, "0 0");
   %this.isThrusting = true;

   %this.thrustschedule = %this.schedule($timeSchedule,
accelerate);
}
```

В начале метода мы берем у объекта PlayerShip текущий угол поворота. Объект %this, передаваемый в метод в качестве параметра есть указатель на текущий объект. Затем мы прибавляем 90 градусов к текущему значению угла, что значит поворот против часовой стрелки на 90 градусов. Это нужно, чтобы при 0 градусов объект был направлен вверх. Так было раньше, но после очередного обновления движка, век-

тор, выходящий из 0 градусов, стал указывать направо. Следующим действием происходит проверка: включено ли ускорение, если включено, тогда амплитуда уменьшенная (2-й параметр функции) и вектор ускорения вычисляет с помощью функции: Vector2Direction(%adjustedAngle, 35); Если ускорение выключено, тогда амплитуда более широкая, и вектор ускорения вычисляется так: Vector2Direction(%adjustedAngle, 95); После этого применим силы: метод setLinearDamping — линейное уменьшение скорости (имеется в виду: скорость при движении), в данном случае в методе accelerate линейное затухание скорости выключается — параметр 0. Метод setAngularDamping понижает скорость вращения. Метод applyLinearImpulse применяет импульс, тем самым изменяет скорость движения и поворота. Принимает 2 параметра: вектор движения и координаты мировой позиции. На их основе происходит вычисления скорости движения и вращения. Под конец метода отмечаем флаг ускорения: %this.isThrusting = true; и планируем вызов этого же метода через $timeSchedule миллисекунд: %this.thrustschedule = %this.schedule($timeSchedule, accelerate); Метод планирования schedule принимает 2 параметра: количество миллисекунд и метод, который будет вызван через заданное количество миллисекунд. Последующие необязательные аргументы могут быть параметрами вызываемому методу. Планирование может происходить для конкретного объекта, как в данном случае, а может иметь место для глобальной функции — далее рассмотрим подробнее. Обратите внимание: метод планирования schedule возвращает дескриптор планирования, в данном случае он сохраняется в добавленную переменную-член thrustschedule объекта PlayerShip. В будущем воспользовавшись сохраненным дескриптором можно отменить запланированное выполнение метода.

Остановка (замедление) космического корабля происходит в методе stopthrust:

```
function PlayerShip::stopthrust(%this)
{
   %this.setLinearDamping(0.8);
   %this.setAngularDamping(0.0);
   cancel(%this.thrustschedule);
   %this.isThrusting = false;
}
```

В методе уменьшается скорость перемещения и выключается скорость вращения: без работающих двигателей космический корабль не может маневрировать. Отменяется последующий вызов запланирован-

ного метода ускорения — accelerate. Сбрасывается флаг isThrusting, показывающий включенное ускорение.

Перейдем к маневрам космического корабля, то есть к поворотам. Поворот влево осуществляется методом turnleft:

```
function PlayerShip::turnleft(%this)
{
      if (!$turnRight) {
      $turnLeft = 1;
      %this.gsetAnularVelocity(%this.
getAngularVelocity() + $turnSpeed);
      %this.turnschedule = %this.
schedule($timeSchedule, turnleft);
   }
}
```

Обратите внимание: глобальные булевы переменные $turnLeft и $turnRight нужны для того, чтобы пользователь не мог одновременно сделать повороты в обе стороны, в таком случае может возникнуть зацикливание. Поэтому, в случае поворота налево сначала происходит проверка, чтобы поворот вправо не был включен и тогда включается переменная $turnLeft, а объекту PlayerShip с помощью метода setAngularVelocity устанавливается скорость поворота, которая равна (значение передается в аргументе) текущей скорости поворота (узнаем посредством метода getAngularVelocity()) плюс скорость поворота (хранится в глобальной переменной $turnSpeed). Следующим действием запланируем выполнение этого метода через $timeSchedule миллисекунд.

Метод для поворота вправо аналогичен: только здесь мы проверяем состояние переменной $turnLeft, и, если она равна 0, то включаем переменную $turnRight, после чего устанавливаем скорость вращения корабля методом setAnularVelocity, только на этот раз из текущей скорости поворота мы вычитаем значение глобальной переменной $turnSpeed.

Остановка поворотов происходит в методе stopturn:

```
function PlayerShip::stopturn(%this)
{
   cancel(%this.turnschedule);
   %this.setAngularVelocity(0);
   $turnLeft = 0;
   $turnRight = 0;
}
```

Отменяем запланированный поворот; обнуляем скорость поворота; сбрасываем переменные — индикаторы поворота.

Управление космическим кораблем

Заготовка проекта уже содержит модуль реализации управления (см. главу 3), нам остается только дополнить его. В функции InputManager::Init_controls(this) создается карта активностей, в которую добавляются реакции на нажатия клавиш:

```
new ActionMap(shipcontrols);// создание карты
//активностей
```

Когда карта создана, в нее можно добавлять реакции на нажатия клавиш, это осуществляется с помощью метода bindCmd карты активностей:

```
shipcontrols.bindCmd(keyboard, "left", "if
(isObject(PlayerShip))PlayerShip.turnleft();",
"if (isObject(PlayerShip))PlayerShip.stopturn();");
```

Первым параметром метод принимается имя прослушиваемого устройства ввода, в данном случае — это клавиатура — keyboard; второй параметр — клавиша, нажатие которой ожидается («left» — стрелка влево); третий параметр — действие, которое происходит во время нажатия клавиши, определенной в предыдущем параметре. Здесь можно указать условный оператор, заключенный в кавычки, как в рассматриваемом примере:

```
"if (isObject(PlayerShip))PlayerShip.turnleft();"
```

если космический корабль существует (не подбит), то повернуть его влево. Четвертый параметр — действие, осуществляемое в момент отпускания клавиши, определенной во втором параметре:

```
"if (isObject(PlayerShip))PlayerShip.stopturn();"
```

то есть, в данном случае, остановить поворот корабля. Приведу оставшиеся определения действий:

```
shipcontrols.bindCmd(keyboard, "right", "if
(isObject(PlayerShip))PlayerShip.turnright();",
"if (isObject(PlayerShip))PlayerShip.stopturn();");
shipcontrols.bindCmd(keyboard, "up", "if
```

```
(isObject(PlayerShip))PlayerShip.accelerate();",
"if (isObject(PlayerShip))PlayerShip.stopthrust();");

shipcontrols.bindCmd(keyboard, "space", "if
(isObject(PlayerShip))PlayerShip.Fire(0);", "");
shipcontrols.bindCmd(keyboard, "lcontrol", "if
(isObject(PlayerShip))PlayerShip.Fire(1);", "");
```

Последним действием, карта активностей добавляется на обработку:

```
shipcontrols.push();
```

Метод Fire космического корабля мы рассмотрим несколько позднее.

Столкновения

Когда происходит столкновение объектов, в котором участвует наш космический корабль, из движка управление передается в скриптовый код в функцию обратного вызова onCollision объекта PlayerShip. Кроме указателя на текущий космический корабль, в качестве параметров метод принимает: %sceneObject — указатель на объект, с которым произошло столкновение, в объекте %collisionDetails передается список значений, дающий подробнейшую картину столкновения, среди них: индексы, нормали, координаты точек соприкосновения (соударения) и т.д. Пока такая подробная информация для нас не играет никакой роли. Ниже приведен код метода:

```
function PlayerShip::onCollision(%this, %sceneobject,
%collisiondetails)
{
   if (%sceneobject.getSceneGroup() == $asteroidGroup
   || %sceneobject.getSceneGroup() == $ufoGroup) {
       createExplosion(%this);
       destroyPlayerShip();
   }
}
```

В теле метода первым делом проверяется номер группы, которой принадлежит столкнувшийся объект. Если это группа астероидов или НЛО, то наш космический корабль поражен, тогда надо создать взрыв (класс взрывов мы рассмотрим в следующем разделе) с помощью функции createExplosion и уничтожить космический корабль, что осуществляется в функции destroyPlayerShip.

```
function destroyPlayerShip()
```

```
{
   GameOverScr::show(PlayerShip.score);
   PlayerShip.safeDelete();
}
```

Функция destroyPlayerShip выполняет только 2 действия: выводит окошко, оповещающее о конце игры и уничтожает космический корабль. Обратите внимание: как происходит удаление корабля: не с помощью метода delete, который выполняется немедленно, а с помощью метода safeDelete, который осуществляет операцию удаления безопасно, дожидаясь конца выполнения удаляемым объектом текущего действия.

7 Взрывы

Взрывы представляют собой игровые объекты, живущие непродолжительное время. Они образуются в месте крушения и/или столкновения игровых объектов.

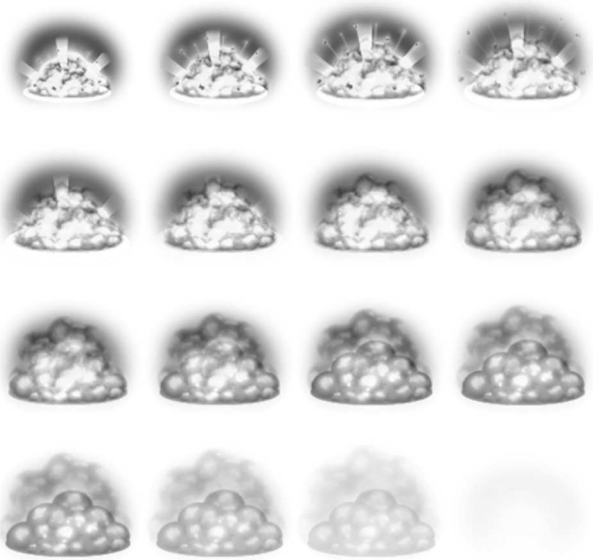

Рис. 6.3. Взрыв

Основное изображение для взрывов представлено в файле impactExplosion.png. Это последовательность, состоящая из 16 картинок, каждая из которых показывает определенный этап взрыва.

Исполняемый код на языке Torque Script для взрывов содержится в файле Explosion.cs и состоит только из одной функции, создающей отдельный взрыв:

```
function createExplosion(%sceneobject)
{
    %explosion = new ParticlePlayer();
    %explosion.Particle = "ToyAssets:impactExplosion";
    %explosion.setPosition(%sceneobject.
getPosition());
    %explosion.setSizeScale(2);
    myScene.add(%explosion);
}
```

В качестве параметра функция принимает объект, для чего она это делает — скоро увидим. В этой функции мы познакомимся с новым типом игровых объектов — классом ParticlePlayer. Как следует из названия: этот объект представляет источник (проигрыватель) частиц. Объект — испускатель частиц создается стандартным образом. Следующим действием свойству Particle мы присваиваем ассет, созданный на основе taml-файла, содержащего описание частиц на языке TAML:

```
<ParticleAsset
    AssetName="impactExplosion"
    Lifetime="1"
    LifeMode="KILL">
    <ParticleAssetEmitter
        EmitterName="debris"
        EmitterType="BOX"
        EmitterAngle="90"
        EmitterSize="2 1"
        AttachPositionToEmitter="1"
        AttachRotationToEmitter="1"
        FixedForceAngle="-90"
        OldestInFront="1"
        Image="@asset=ToyAssets:Asteroids"
        RandomImageFrame="1">
        <ParticleAssetEmitter.Fields>
            <Quantity
                Keys="0 250 0.1 150 0.11 0" />
            <LifetimeVariation
                Keys="0 1" />
            <EmissionArc
```

```
                Keys="0 90" />
            <Speed
                Keys="0 1.8" />
            <SizeXVariation
                Keys="0 2.1" />
            <FixedForce
                Keys="0 8" />
            <SizeXLife
                Keys="0 0 0.1 0.1" />
            <RandomMotion
                Keys="0 10" />
            <FixedForceVariation
                Keys="0 2" />
            <SpinVariation
                Keys="0 360" />
            <SpeedVariation
                Keys="0 0.3" />
            <AlphaChannel
                Keys="0 0 0.1 1 0.8 1 1 0" />
            <EmissionAngle
                Keys="0 90" />
        </ParticleAssetEmitter.Fields>
    </ParticleAssetEmitter>
    <ParticleAssetEmitter
        EmitterName="flames"
        EmitterType="LINE"
        EmitterSize="1 0"
        AttachPositionToEmitter="1"
        AttachRotationToEmitter="1"
        IntenseParticles="1"
        OldestInFront="1"
        Animation="@asset=ToyAssets:Impact_
ExplosionAnimation">
        <ParticleAssetEmitter.Fields>
            <Quantity
                Keys="0 20 0.1 20 0.11 0" />
            <Lifetime
                Keys="0 1" />
            <Speed
                Keys="0 0" />
            <SizeXVariation
                Keys="0 2" />
            <SizeXLife
                Keys="0 0 0.1 3" />
            <AlphaChannel
                Keys="0 0 0.1 1 0.8 1 1 0" />
        </ParticleAssetEmitter.Fields>
    </ParticleAssetEmitter>
</ParticleAsset>
```

Описание весьма обширно. Вкратце рассмотрим его. Сначала для ассета задается имя, затем время жизни одной частицы, действие, происходящее, когда время истечет: нам надо удалить частицу. Далее описываются параметры испускателя частиц: его имя, тип, угол поворота, размер, ассет-картинка; затем заполняются поля: количество частиц в определенном кадре, вариации времени жизни, дуга распространения, скорость, сила, размеры по разным осям, значение альфа-канала.

В одном файле может быть описано несколько объектов. К примеру, в этом же файле описан еще один проигрыватель частиц, имеющий другое имя. Не будем повторяться: по вышеприведенному описанию вы сможете разобраться с его параметрами без моей помощи.

После присвоения испускателя частиц для проигрывателя частиц мы устанавливаем его положение. А позиция как раз берется от полученного в параметре объекта. Затем устанавливаем для нашего проигрывателя размер и добавляем в стек сцены, то есть выводим на экран. На этом создание проигрывателя частиц завершается. После своего выполнения он самоуничтожится.

8 Астероиды

Астероиды представляют свободно дрейфующие в открытом космосе тела, с изначально заданными вектором, скоростью и углом вращения.

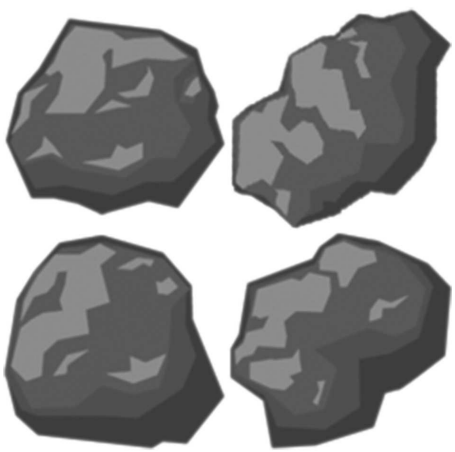

Рис. 6.4. Астероиды

Для астероидов заготовлено изображение asteroids.png, состоящее из 4-х картинок немного отличающихся «камней». Картинки расположены, как 2x2, то есть: 2 в верхнем ряду и 2 — в нижнем.

Описание ассета находится в файле asteroids.asset.taml. В нем содержится такой код:

```
<ImageAsset
    AssetName="Asteroids"
    ImageFile="asteroids.png"
    CellCountX="2"
    CellCountY="2"
    CellWidth="128"
    CellHeight="128" />
```

Описание этого ассета сравнительно проще, чем описание ассета, рассмотренного нами в предыдущем разделе данной главы. В первой строчке после открытия тэга ImageAsset мы присваиваем имя ассету, во второй — задаем графический файл для загрузки, далее определяем количество картинок по горизонтали и вертикали, наконец, двумя последними строчками задаем ширину и высоту каждой картинки, другими словами, кадра.

Скриптовый код, реализующий астероиды, находится в файле asteroids.cs. В нем содержатся только 2 функции: создающая и уничтожающая объекты астероидов. Ниже представлен код первой из них:

```
function createAsteroids(%nums, %size, %pos)
{
   for(%i=0; %i<%nums; %i++) {
       %asteroid = new Sprite();
       %asteroid.Class = "Asteroid";
       %asteroid.setBodyType(dynamic);
       if (%pos == 0) {
          %n = getRandom(0, 10);
          %m = getRandom(0, 10);
          if (%n < 5) %x = getRandom(-40, -45); else
          if (%n >= 5) %x = getRandom(40, 45);
          if (%m < 5) %y = getRandom(-25, -30); else
          if (%m >= 5) %y = getRandom(25, 30);
          %asteroid.Position = %x SPC %y;
       } else %asteroid.Position = %pos;
       if (%size == 0)
         %randomsize = getRandom(6,10);
       else %randomsize = %size;
       if (%size == 0 && %pos == 0)
         %asteroid.life = 1;//у астероида 2 жизни:
//сначала распад, затем гибель
```

```
       else if (%size != 0 && %pos != 0) %asteroid.
life = 0;
       %asteroid.Size = %randomsize;
       %asteroid.createCircleCollisionShape((
%randomsize*0.85)/2);
       %asteroid.setDefaultRestitution(
getRandom(0.5,1.1));
       %asteroid.setLinearVelocity(
getRandom(-10,10)*3, getRandom(-5,5)*3);
       %asteroid.setAngularVelocity(getRandom(5,25));
       %asteroid.SceneGroup = $asteroidGroup;
       %asteroid.Image = "ToyAssets:Asteroids";
       %asteroid.setImageFrame(getRandom(0,3));

       %asteroid.setCollisionGroups($borderGroup,
$asteroidGroup);
       %asteroid.setSceneGroup($asteroidGroup);
       %asteroid.setCollisionLayers(1);
       %asteroid.setSceneLayer(1);
       myScene.add(%asteroid);
       $asteroidsCount++;
   }
}
```

Данная функция принимает 3 параметра: %nums, %size, %pos, из которых обязательным является только один — %nums, он определяет количество создаваемых астероидов. Второй параметр — размер астероида, третий — позиция — пара значений, определяющие координаты по осям X и Y.

Дальше мы рассмотрим тело функции. В самом начале запускается цикл for, его счетчик выполняется от нуля до необходимого количества астероидов. Каждый астероид — это объект класса Sprite: %asteroid = new Sprite(); Все астероиды принадлежат классу Asteroid. Так как данные объекты должны взаимодействовать с внешней средой, присваиваем им динамический тип. Далее следует витиеватый механизм задания координат для вновь созданного астероида:

```
if (%pos == 0) {
          %n = getRandom(0, 10);
          %m = getRandom(0, 10);
          if (%n < 5) %x = getRandom(-40, -45); else
          if (%n >= 5) %x = getRandom(40, 45);
          if (%m < 5) %y = getRandom(-25, -30); else
          if (%m >= 5) %y = getRandom(25, 30);
          %asteroid.Position = %x SPC %y;
  } else %asteroid.Position = %pos;
```

В начале проверяется: не равен ли параметр %pos нулю, если равен, в таком случае, мы случайным образом в диапазоне от 0 до 10 последовательно выбираем 2 значения. Если первое значение — переменная %n меньше 5, тогда переменной %x мы присваиваем случайное число, выбранное из отрезка от -40 до -45, если же это значение больше или равно 5, тогда переменной %x мы присваиваем число от 40 до 45. Если второе значение — переменная %m меньше 5, тогда переменной %y мы присваиваем число от -25 до -30, в обратном случае, если значение %m больше или равно 5, то переменной %y присваиваем случайное число от 25 до 30. Последней строкой данного блока: %asteroid.Position = %x SPC %y; мы формируем и присваиваем пару координат свойству Position объекту класса Asteroid. Оператор SPC объединяет 2 строковые переменные, ставя между ними пробел.

Такие диапазоны значений координат выбраны не случайно. Как мы видели при рассмотрении заготовки, окно вывода представляет собой прямоугольник шириной 100 условных единиц и высотой 75, то есть 4:3. Учитывая, что начало координат располагается в центре — 0, тогда ширина окна вывода простирается от -50 до 50, а высота, соответственно, от -37,5 до 37,5. Значение выбирается так, чтобы астероид не оказался в середине сцены, и, чтобы не очутился вплотную к границам сцены.

С другой стороны, если в начале проверки переменная %pos не равна 0, тогда за место всех расчетов, ее значение напрямую присваивается свойству Position астероида.

Дальше происходит задание размера астероида:

```
if (%size == 0)
%randomsize = getRandom(6,10);
else
%randomsize = %size;
```

Если параметр %size равен 0, то переменная %randomsize получает случайное значение от 6 до 10. Иначе, когда параметр %size содержит значение, оно присваивается переменной %randomsize.

Следующий оборот весьма интересен. По замыслу, когда снаряд, выпущенный из пушки космического корабля, попадает по астероиду, последний взрывается и раскалывается надвое, и эти две части продолжают бороздить космическое пространство. Если игрок поражает один из осколков, последний взрывается и уничтожается. Следовательно, изначально астероид имеет 2 жизни, а после раскола каждая из частей только одну. Для лаконичности кода нам надо сделать так, чтобы все астеро-

иды создавались в одной функции. Мы снова можем воспользоваться параметрами, получаемыми функцией:

```
if (%size == 0 && %pos == 0)
%asteroid.life = 1;
else
if (%size != 0 && %pos != 0)
%asteroid.life = 0;
```

Здесь мы снова проверяем: если параметры %size и %pos равны 0, из чего следует, что создается начальный астероид, так как позиция и размер при таких значениях параметров выбираются случайно из диапазона, тогда свойству life объекта класса Asteroid присваивается значение 1, так как счет ведется от 0, значит, присваивается 2 жизни. В другом случае, когда параметры %size и %pos не равны 0, значит, параметры уже определены и создается осколок астероида, тогда последний получает одну жизнь.

Время создать коллизию, обхватывающую астероид. Под какую стандартную, реализованную в физическом движке форму коллизии подходит астероид? Несмотря на небольшие отличия, астероид вполне умещается в окружность. На некоторые различия вполне можно не обращать внимание, так как благодаря этой фигуре мы получим быстрый способ нахождения пересечений (столкновений) объектов. Окружность является стандартной фигурой, которая поддерживается физическим движком, поэтому для созданий круглой коллизии в Torque 2D есть функция: createCircleCollisionShape. Точнее, это метод класса Sprite. Он принимает один параметр — радиус охватывающего круга. После задания коллизии мы определяем степень «пружинистости» астероидов при столкновении (с помощью метода setDefaultRestitution). Затем для каждого астероида задаем линейную скорость перемещения — методом setLinearVelocity, выбирая для каждой из осей случайное приращение из определенного диапазона. Кроме того, для каждого астероида задается случайная скорость вращения, используя метод setAngularVelocity. Присваивается группа сцены (SceneGroup). Для выводимого изображения (свойство Image) выбирается ассет ToyAssets:Asteroids. Как мы знаем, этот ассет определяет 4 кадра, находящихся в одном изображении. Поэтому нам надо выбрать один кадр, сделаем это рандомно:

```
%asteroid.setImageFrame(getRandom(0,3));
```

Следующим действием методом setCollisionGroups устанавливаем для астероида группы коллизий, с которыми он может взаимодействовать, это (перечисляются через запятую) $borderGroup (группа границ экрана) и $asteroidGroup (группа астероидов, то есть они могут взаимодействовать между собой). Далее устанавливается группа сцены, слой коллизии и слой сцены. Наконец, астероид добавляется в стек сцены — выводится на экран: myScene.add(%asteroid); Последним действием мы увеличиваем глобальную переменную $asteroidsCount, как следует из названия, хранящую количество астероидов.

Кроме того, в этом файле находится функция destroyAsteroid, которая удаляет переданный в параметре объект астероида. Плюс она уменьшает значение глобальной переменной $asteroidsCount.

```
function destroyAsteroid(%asteroid)
{
   $asteroidsCount--;
   %asteroid.safedelete();
}
```

9 Оружие

Наш космический корабль имеет 2 типа оружия: пули и лазер. По замыслу: пули бесконечны, однако лазерная пушка во время выстрела нагревается. Если сделать 10 выстрелов, то лимит будет исчерпан, пушку нельзя будет использовать. В то же время она медленно остывает. И через некоторое время можно будет продолжить стрельбу. Остывание происходит медленно и постепенно, то есть, чем дольше промежуток времени для остывания был отведен, тем больше выстрелов можно совершить. Тем не менее максимальное количество выстрелов равно 10.

Лазер имеет перед пулями одно преимущество: если пулей попасть по астероиду, то он распадается надвое, с другой стороны если астероид поразить лазером, тогда астероид сразу уничтожается.

Пули

Пуля представляет собой шарик 20 × 20 пикселей, ее изображение хранится в папке ToyAssets в файле bullet.png. Код для загрузки пули — bullet.asset.taml — выглядит весьма просто:

```
<ImageAsset
    AssetName="bullet"
    ImageFile="bullet.png" />
```

Рис. 6.5. Пуля

Единственное что выполняет данный код — это загружает изображение из файла.

Перейдем к исполняемому скриптовому коду, к файлу bullet.cs. Создание пуль происходит в функции createBullet. Она принимает 3 параметра: %xpos — позиция по оси X для создаваемой пули; %ypos — позиция по оси Y; %angle — угол поворота вектора движения пули. Ниже приведен код функции createBullet:

```
function createBullet(%xpos, %ypos, %angle)
{
    %bullet = new Sprite();
    %bullet.class = "Bullet";
    %bullet.setBodyType(dynamic);
    %bullet.Position = %xpos SPC %ypos;
    %bullet.Size = "1 1";
    %bullet.Image = "ToyAssets:bullet";
    %bullet.setCollisionCallback(true);
    %bullet.setFixedAngle(true);
    %bullet.createCircleCollisionShape(1.5/2);

    %bullet.setCollisionGroups($borderGroup,
$asteroidGroup, $ufoGroup);
    %bullet.setSceneGroup($bulletGroup);
    %bullet.setCollisionLayers(1);
    %bullet.setSceneLayer(1);

    myScene.add(%bullet);

    %bullet.accelerate(%angle);
}
```

Сначала создаем спрайт (Sprite); присваиваем объекту принадлежность кастомному классу Bullet; делаем объект динамическим; координаты для позиции берем из переданных в функцию параметров; размер устанавливаем равный единице по обеим осям; в качестве изображения передаем ссылку на ассет Bullet; устанавливаем срабатывание вызова функции обратного вызова в момент, когда происходит столкновение; отмечаем использование установленных углов. Затем с помощью

метода createCircleCollisionShape задаем размер круглой коллизии для объекта пули. В параметрах метода setCollisionGroup перечисляем группы объектов, с которыми будет взаимодействовать пуля, это такие группы: $borderGroup (границы окна вывода, пуля не должна их покидать), $asteroidGroup (астероиды — с ними пуля вступает в непосредственное взаимодействие, взрывая их), $ufoGroup (группа для НЛО, с ними так же приходится взаимодействовать, так как при попадании пули, НЛО уничтожается). Далее последовательно устанавливаем для объекта пули группу сцены, слой столкновения, слой сцены, добавляем ее в стек сцены и вызываем для нее метод accelerate.

```
function Bullet::accelerate(%this, %angle)
{
   %adjustedAngle = %angle + 90;
   %ThrustVector = Vector2Direction(%adjustedAngle,
500);
   %this.applyLinearImpulse(%ThrustVector, "3 3");
}
```

Кроме указателя на обрабатываемый объект, метод получает параметр %angle — угол поворота, переданный из функции createBullet. Первым делом в методе происходит настройка угла поворота пули прибавлением к переменной %angle числа 90. Зачем это нужно вы можете прочитать в подразделе «Движения космического корабля». Затем мы вычисляем вектор движения, это делается посредством функции Vector2Direction. Она принимает два параметра: модифицированный угол поворота и амплитуду (подробнее см. в подразделе «Движения космического корабля»). Наконец, с помощью метода applyLinearImpulse применяем импульс для движения пули.

Последняя содержащаяся в этом файле функция обратного вызова служит для обработки столкновений onCollision. Она принимает 3 параметра: указатель на текущий объект, ссылку на объект, с которым произошло столкновение и массив данных о столкновении (более подробно см. в подразделе «Столкновения»). Ниже приведен исходный код метода:

```
function Bullet::onCollision(%this, %sceneobject,
%collisiondetails)
{
  if (%sceneobject.getSceneGroup() == $asteroidGroup)
{
    createExplosion(%sceneobject);
    %size = %sceneobject.getSize().X;
    %pos = %sceneobject.getPosition();
```

```
    %this.safedelete();
    if (%sceneobject.life == 1) {
       PlayerShip.score += $asteroidsScore;
       createAsteroids(2, %size / 2, %pos);
    }
    else if (%sceneobject.life == 0) {
         if ($asteroidsCount <= $asteroidsMaxCount)
            createAsteroids(1);
            PlayerShip.score += $asteroidsScore * 2;
    }
    destroyAsteroid(%sceneobject);
  } else
  if(%sceneobject.getSceneGroup() == $borderGroup){
     %this.safedelete();
  } else
  if(%sceneobject.getSceneGroup() == $ufoGroup) {
     %this.safedelete();
     createExplosion(%sceneobject);
     destroyUfo(%sceneobject);
     PlayerShip.score += $ufoScore;
  }
  $scoreLab.updateLabel("Score: " @ PlayerShip.score);
}
```

В методе происходит проверка: какой объект поражен пулей, если это астероид ($asteroidGroup), тогда с помощью метода createExplosion надо создать взрыв; затем получить размер и позицию пораженного объекта; после этого пулю можно удалить: %this.safedelete(); Осуществляется проверка: сколько жизней у астероида — если 1 (счет от нуля), тогда у космического корабля увеличивается значение свойства score (заработанные очки) и вызывается метод createAsteroids, которому в данном случае передается 3 параметра: 2 — количество создаваемых астероидов, %size / 2 — размер (в 2 раза меньше начального) и %pos — позиция. В другом случае если у астероида 0 (счет с нуля) жизней, то происходит проверка, чтобы количество находящихся на экране астероидов не превышало заданный лимит, если проверка успешна, создается один полноценный (не осколок) астероид (с помощью функции createAsteroids) со случайными значениями свойств. После этого условного оператора, когда астероид, в который попала пуля, больше не нужен, вызывается функция destroyAsteroid, которая уничтожает пораженный астероид.

Если объект, в который попала пуля, принадлежит группе границ — $borderGroup, нужно сразу уничтожить пулю, больше ничего выполнять не надо:

```
if (%sceneobject.getSceneGroup() == $borderGroup) {
     %this.safedelete();
}
```

Если тип объекта — НЛО ($ufoGroup), мы уничтожаем пулю, создаем взрыв, удаляем НЛО, добавив кораблю соответствующее вознаграждение в виде очков. В конце метода обработки столкновения, мы обновляем текстовую метку, отображающую очки космического корабля, соответствующим значением. Текстовые метки мы рассмотрим в разделе 11.

Лазерный луч

Второй тип оружия, имеющийся на космическом корабле — это лазерный луч или просто лазер. Во многом лазерный луч подобен пули, тем не менее есть небольшие отличия. Ассет лазера отличается только именем загружаемого файла — ray.png. В функции создания луча — CreateRay теперь используется свойство Angle, которое в объекте пули не использовалось, так как круг невозможно повернуть. Создание коллизии выполняется методом createPolygonBoxCollisionShape. Ускорение/перемещение происходит точно так же как и пули. Изменение в событии onCollision касаются поражения астероидов: даже первоначальные (не осколочные) уничтожаются с первого попадания (ниже приводится фрагмент обработчика события, реагирующий на столкновение с астероидом):

```
if (%sceneobject.getSceneGroup() == $asteroidGroup) {
    createExplosion(%sceneobject);
    %this.safedelete();
    destroyAsteroid(%sceneobject);
    createAsteroids(1);
    PlayerShip.score += $asteroidsScore;
} else
```

Реализация стрельбы космического корабля

Когда пользователь нажимает на пробел или левый CTRL, вызывается метод Fire с соответствующим параметром: 0 или 1, это позволяет различить, какой тип патрон использовать:

```
if (isObject(PlayerShip))PlayerShip.Fire(0);
if (isObject(PlayerShip))PlayerShip.Fire(1);
```

Перейдите в конец файла spaceship.cs, именно туда мы будем добавлять код. Первый метод, который мы добавим, будет Fire:

```
function PlayerShip::Fire(%this, %num)
{
   switch(%num) {
      case 0 : createBullet(%this.getPosition().X,
%this.getPosition().Y, %this.Angle);
      case 1 :
         if (PlayerShip.rayAmmo > 0) {
            createRay(%this.getPosition().X, %this.
getPosition().Y, %this.Angle);
            PlayerShip.rayAmmo--;
            $ammoLab.updateLabel("Ray Ammo: " @
PlayerShip.rayAmmo);
            if ($rechargeEvent != 0)
               cancel($rechargeEvent);
            $rechargeEvent = PlayerShip.
schedule($rechargeRayDelay, rechargeRay);
         }
   }
}
```

Метод получает номер используемых патрон. Внутри него в условном операторе switch выполняется проверка значения параметра %num, если оно равно 0, тогда вызывается метод createBullet, который создает пулю, ему передаются координаты по осям X и Y расположения космического корабля — текущего объекта %this, а так же его угол поворота. Если переданное в метод значение равно единице, тогда происходит создание луча лазера, затем на 1 уменьшается значение свойства rayAmmo объекта PlayerShip. После этого методом update объекта класса TextLab обновляется надпись, отображающая количество оставшихся выстрелов лазером. Следующим действием игра проверяет: если глобальная переменная $rechargeEvent, предназначенная для хранения указателя на запланированное событие, не равна 0, то есть в ней установлен указатель и событие запланировано, тогда это событие отменяется. После этого независимо от прошлого шага на выполнение планируется метод rechargeRay, а в переменную $rechargeEvent помещается указатель на это событие. Этот оборот осуществляется, чтобы во время стрельбы из лазерной пушки, не происходило ее остывание (или перезарядка).

Метод перезарядки или остывания пушки имеет следующий вид:

```
function PlayerShip::rechargeRay()
{
   if (PlayerShip.rayAmmo < $maxRayAmmo) {
      PlayerShip.rayAmmo++;
```

```
      $ammoLab.updateLabel("Ray Ammo: " @ PlayerShip.
rayAmmo);
      if ($rechargeEvent != 0)
               cancel($rechargeEvent);
      PlayerShip.schedule($rechargeRayDelay,
rechargeRay);
   }
}
```

Он не получает никаких параметров и вызывается постоянно через определенный интервал по умолчанию 2000 миллисекунд — 2 секунды. Сначала в нем осуществляется проверка: если значение свойства rayAmmo космического корабля меньше, чем лимит количества лазерных зарядов, тогда это значение увеличивается на один, затем происходит обновление надписи. Потом проверяется $rechargeAmmo подобно, как это происходит в методе Fire. После условного оператора планируется метод rechargeRay, только на этот раз указатель на запланированное событие не помещается в переменную, и его нельзя отменить.

10 НЛО

НЛО представляет собой второй вид противника. Если астероиды ведут себя пассивно, то НЛО нападают активно. Только появившись на сцене они всегда преследуют космический корабль игрока. При этом, чтобы подбить тарелку, нужна одна пуля или луч.

Для отображения НЛО используется файл ufo.png.

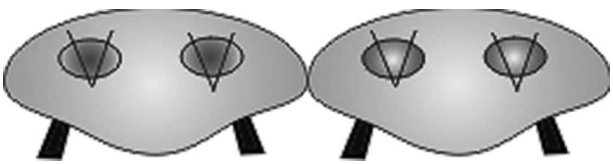

Рис. 6.6. Изображение НЛО

Этот файл описывается TAML-кодом из ufo.asset.taml:

```
<ImageAsset
```

```
    AssetName="ufo"
    ImageFile="UFO.png"
    CellCountX="2"
    CellCountY="1"
    CellWidth="128"
    CellHeight="64" />
```

Ассету присваиваем имя ufo; изображение берем из файла UFO.png, в котором, в одной строке находятся 2 картинки; одна картинка имеет ширину 128 и высоту 64 пикселя.

На его основе строится анимационный ассет — ufo_anim.asset.taml:

```
<AnimationAsset
    AssetName="ufo_anim"
    Image="@asset=ToyAssets:UFO"
    AnimationFrames="0 1"
    AnimationTime="1"
    AnimationCycle="1"
    />
```

Имя ассета — ufo_anim; изображение берется с уже созданного ассета UFO; имеется 2 кадра анимации: AnimationFrames=«0 1», кадры переключать через 1 секунду: AnimationTime=«1»; задаем последовательное повторение анимационного цикла: AnimationCycle=«1». «1» значит true.

Скриптовый код для НЛО находится в файле UFO.cs. Создание тарелки происходит в функции createUFO:

```
function createUfo()
{
   if ($ufoCount < $ufoMaxCount) {
       %ufo = new Sprite();
       %ufo.class = "UFO";
       %ufo.setBodyType(dynamic);
       %ufo.Position = getRandom(-50, 50) SPC
getRandom(-35, 35);
       %ufo.Size = "8 4";
       %ufo.Animation = "ToyAssets:ufo_anim";
       %ufo.createPolygonBoxCollisionShape();
       %ufo.setCollisionCallback(true);
       %ufo.setFixedAngle(true);

       %ufo.setCollisionGroups($borderGroup,
$spaceShipGroup, $ufoGroup);
       %ufo.setSceneGroup($ufoGroup);
       %ufo.setCollisionLayers(1);
       %ufo.setSceneLayer(1);
```

```
        // добавить спрайт на сцену
        myScene.add(%ufo);
        $ufoCount++;
        %ufo.Move();
    }
    $ufoPlan = schedule($ufoCreateDelay, 0, createUfo);
}
```

Первым делом производится проверка: если число уже созданных тарелок равно их лимиту, тогда выходим из функции, потому что ничего создавать в таком случае не надо. В обратном случае выполняются операции по созданию тарелки: создается спрайт, объекту присваивается кастомный класс UFO; динамический тип тела объекта; рандомная позиция в любой точке окна вывода; размер: 8 × 4; на этот раз за место изображения (свойство Image), подгружаем рассмотренную выше анимацию (свойство Animation) из ассета. Создаем полигональную коллизию. Указываем наше желание получать обратный вызов во время столкновения. НЛО может взаимодействовать со следующими группами объектов: границы экрана, космический корабль пользователя и группа НЛО. Устанавливаем группу сцены ($ufoGroup), слой коллизии и слой сцены; добавляем созданный и инициализированный объект в стек сцены. После этого увеличиваем значение глобальной переменной, содержащей количество созданных НЛО. Вызываем для готовой тарелки метод Move. После закрывающей скобки тела условного оператора мы планируем создание НЛО через условное количество миллисекунд (в данном случае: 5000):

```
$ufoPlan = schedule($ufoCreateDelay, 0, createUfo);
```

У функции планирования schedule 3 параметра: задержка в миллисекундах, указатель на объект (или 0, как в данном случае), имя функции, вызов которой должен произойти через указанное в первом параметре число миллисекунд. Функция возвращает идентификатор запланированного действия, с помощью которого выполнение действия можно отменить.

Метод Move объекта НЛО вызывается через каждые 500 миллисекунд, чтобы выровнять вектор движения, который всегда устремлен к позиции пользовательского космического корабля, и, поскольку последний не стоит на месте, направление движения нужно корректировать. Метод Move выглядит следующим образом:

```
function UFO::Move(%this)
{
```

```
   if (isObject(PlayerShip)) {
      %this.moveTo(PlayerShip.getPosition(),
$ufoSpeed);
      %this.schedule(500, Move);
   }
}
```

Здесь проверяется, чтобы объект космического корабля существовал, так как к моменту вызова этого метода, корабль может быть уже уничтожен, например, другой тарелкой или астероидом, если корабль существует, тогда для текущей тарелки (обратите внимание на использование указателя %this) вызывается метод moveTo, который направляет данный объект в позицию, координаты которой передаются в первом параметре, со скоростью, переданной во втором параметре. После этого с помощью метода schedule вызов метода Move планируется через 500 миллисекунд. Обратите внимание: метод schedule в отличие от одноименной функции принимает 2, а не 3 параметра, поскольку ID объекта, передаваемого во втором параметре функции, теперь является объектом вызова.

В текущем файле скриптового кода присутствует еще одна функция: destroyUfo, как следует из ее названия, она выполняет уничтожение НЛО, уменьшая при этом количество содержащихся на экране летающих тарелок:

```
function destroyUfo(%ufo)
{
   %ufo.safedelete();
   $ufoCount--;
   cancel($ufoPlan);
   $ufoPlan = schedule($ufoCreateDelay, 0, createUfo);
}
```

Скриптовый класс UFO не имеет обработчика столкновений, так как все возможные условия столкновения с этим типом объектов обрабатываются в других классах объектов.

11 Текстовые надписи

Теперь займемся визуально-декоративными элементами нашей игры.

Текстовые надписи в движке Torque 2D представлены классом ImageFont (до версии 3.3, находящейся в разработке). Он не использует стандартные шрифты из реестра операционной системы. Для него ну-

жен атлас, состоящий из всех используемых в игре символов. В первой главе я привел список программ для создания подобных атласов. По сути, атлас представляет собой изображение. В подпапке Font нашей игры находится файл ArialUnicode.png, который используется для рендеринга надписей в игре. Чтобы не создавать свой файл шрифта, можете воспользоваться моим.

Однако движку было бы не понятно, что это изображение есть атлас шрифта. Как всегда на помощь приходит описание изображение на языке TAML — оно содержится в файле ArialUnicode.asset.taml:

```
<ImageAsset
    AssetName="ArialUnicode"
    ImageFile="ArialUnicode.png"
    CellOffsetX="0"
    CellOffsetY="10"
    CellStrideX="31"
    CellStrideY="33"
    CellCountX="16"
    CellCountY="6"
    CellWidth="20"
    CellHeight="32" />
```

Описание происходит как обычного графического ассета: присваиваем ему имя; назначаем графический файл; дальше определяем отступы для ячейки по обеим осям; назначаем размер шага, через который находится следующая ячейка (буква); определяем количество ячеек по горизонтали и вертикали; устанавливаем ширину и высоту ячейки.

Класс TextLab находится в файле TextLabs.cs. Он содержит только 2 метода: createLabel и updateLabel:

```
function TextLab::createLabel(%x, %y)
{
    %lab = new ImageFont();
    %lab.Class = "TextLab";
    %lab.SceneLayer = 0;
    %lab.Image = "ToyAssets:ArialUnicode";
    %lab.Position = %x SPC %y;
    %lab.FontSize = "2 2";
    %lab.FontPadding = 0;
    %lab.TextAlignment = "Left";
    %lab.setBodyType(static);
    myScene.add(%lab);

    return %lab;
}
```

```
function TextLab::updateLabel(%this, %text)
{
   %this.Text = %text;
}
```

Первый является своего рода конструктором, он получает координаты для расположения создаваемой надписи. Как я сказал выше: надпись — это объект класса ImageFont; для этого объекта устанавливаем уже знакомые свойства, такие как: имя класса изображение (ассет ArialUnicode); позицию; тип тела (static); кроме того, у нового типа объектов присутствуют новые свойства: размер шрифта (FontSize) указывается по обеим осям; растяжка символов (FontPadding), выравнивание (TextAlignment). В конце метода он возвращает созданный объект вызываемому коду.

Второй метод только обновляет текст на надписи, получая новый текст в параметре.

Возвратимся в файл background.cs в функцию createBackground. В конце этой функции нам надо добавить код для создания надписей:

```
$scoreLab = TextLab::createLabel(-48, 35);
$scoreLab.updateLabel("Score: 0");
$ammoLab = TextLab::createLabel(-48, 33);
$ammoLab.updateLabel("Ray Ammo: " @ $maxRayAmmo);
```

Создаются 2 надписи, указатели на которые сохраняются в глобальных переменных, из чего следует, что мы всегда сможем обновить надписи. Мы уже видели, как происходит обновление очков, когда игрок поражает астероиды или летающие тарелки.

Вторая надпись отображает запас количества лазерных выстрелов.

Кроме того, здесь же для фона надо включить использование событий ввода:

```
%background.setUseInputEvents(true);
```

Пока нам не надо обрабатывать события для фона, тем не менее в будущем это пригодится.

Последним действием нам надо отсюда вызвать конструктор экрана меню, рассматриваемое в следующем разделе:

```
GameOverScr::create();
```

12 Экран меню

В игре Asteroids есть только один экран меню. Он появляется, когда главный космический корабль терпит поражение. На этом экране (окне) отображается счет, набранный игроком, и отображаются 2 кнопки: Start (начать игру с начала) и Exit (выйти и закрыть приложение).

Окно меню представляет собой спрайт, отображаемый поверх игровых объектов. Сверху на этом спрайте находятся еще 3 объекта: 2 спрайта — кнопки и надпись (объект класса TextLab, рассмотренном нами в предыдущем разделе). Логично назвать класс этого окна GameOverScr, он располагается в файле GameOverScreen.cs:

```
function GameOverScr::create()
{
   %win = new Sprite(GOScr)//создание спрайта для окна
   {
      Position = "0 0";
      Size = $GameOverScrWidth SPC $GameOverScrHeight;
      SceneLayer = 0;
      Image = "ToyAssets:GameOver";
      Visible = false;
      BodyType = static;
   };
   myScene.add(%win);

   %but = new Sprite(butStart)//спрайт для кнопки Start
   {
      Position = "-15 -12";
      Size = "10 5";
      SceneLayer = 0;
      Image = "ToyAssets:butStart";
      Visible = false;
      BodyType = static;
   };
   myScene.add(%but);

   %but = new Sprite(butExit)//спрайт для кнопки Exit
   {
      Position = "15 -12";
      Size = "10 5";
      SceneLayer = 0;
      Image = "ToyAssets:butExit";
      Visible = false;
      BodyType = static;
   };
   myScene.add(%but);
```

```
   $finalScoreLab = TextLab::createLabel(-10, 0);
//создание надписи
   $finalScoreLab.updateLabel("Score: ");
   $finalScoreLab.Visible = false;
}
```

Ничего нового в этом коде для нас нет, мы уже встречались с созданием спрайтов.

После создания окна оно делается невидимым; вызов конструктора класса GameOverScr происходит в конце функции createBackground. Метод для показа окна меню вызывается из функции destroyPlayerShip, в тот момент, когда разрушается пользовательский космический корабль. Вместе с тем в метод show класса GameOverScr в качестве параметра передается количество очков, которое набрал пользователь во время игры: `GameOverScr::show(PlayerShip.score);`

Сам метод show выглядит следующим образом:

```
function GameOverScr::show(%score)
{
   if (isObject(GOScr))
      GOScr.Visible = true;
   if (isObject($finalScoreLab)) {
      $finalScoreLab.updateLabel("Score: " @ %score);
      $finalScoreLab.Visible = true;
   }
   if (isObject(butStart))
      butStart.Visible = true;
   if (isObject(butExit))
      butExit.Visible = true;
}
```

В методе происходит отображение объектов, а так же обновление надписи, показывающей набранные игроком очки.

Обработка нажатий на визуальные элементы (кнопки)

В Torque 2D узнать о нажатии на кнопку можно двумя способами. Первый состоит в том, чтобы объявить для элемента функцию обратного вызова, другими словами, обработчик события, который будет срабатывать в момент, когда пользователь нажимает кнопку. Второй способ заключается в том, что мы проверяем объекты, находящиеся под курсором во время нажатия, если среди них оказалась какая-то кнопка, значит, на ней было совершено нажатие, поэтому надо выполнить соответствующую функцию.

У нас есть объект — InputManager, отвечающий за обработку ввода, он прослушивает ввод со всего окна; этого мы добились, сделав его прослушивателем основного окна внутри конструктора игры — AsteroidsGame:

```
mySceneWindow.addInputListener(InputManager);
```

Теперь в обработчике события нажатия левой клавиши мыши мы можем написать код для нахождения объекта под курсором и вызова соответствующей функции:

```
function InputManager::onTouchDown(%this, %touchId,
%worldposition)
{
   //проверить все кнопки
   %picked = myScene.pickPoint(%worldposition);
   for(%i = 0; %i < %picked.count; %i++) {
      %myobj = getWord(%picked, %i);
      if (%myobj != 0 && %myobj.visible == true) {
         switch$(%myobj.getName()) {
            case "butExit" : quit();
            case "butStart" :
               GameOverScr::hide();
               destroyGame();
               startGame();
               break;
         }
      }
   }
}
```

С помощью метода pickPoint мы получаем все объекты, находящиеся под курсором, для этого в качестве параметра мы передаем координаты курсора, находящиеся в переменной %worldposition, полученной из параметра. Обычно возвращается список объектов, он выглядит как идентификаторы объектов, разделенные пробелом. Затем мы в цикле перебираем все элементы возвращенного списка, сохраняем в переменной %myobj тот идентификатор по счету, на который указывает счетчик цикла; проверяем, чтобы %myobj не была равна 0 и, чтобы объект, на который указывает %myobj был в видимом состоянии; далее получаем имя данного объекта — кнопки, если это butExit, то закрываем приложение (вызываем функцию quit), если это butStart, то сначала разрушаем текущую игру (destroyGame), а потом — создаем новую (startGame).

Рассмотрим функции конца и начала игры (destroyGame, startGame), дополним файл control.cs:

```
function destroyGame()
```

```
{
   while (myScene.getCount())
      myScene.getObject(0).delete();
   destroyScene();

   $asteroidsCount = 0;
   $ufoCount = 0;
   cancel($ufoPlan);
}
function StartGame()
{
   createScene();
   mySceneWindow.setScene(myScene);
   createBackground();
   InputManager.dirControls();
   createSpaceShip();
   createAsteroids($asteroidsMaxCount);
   createUfo();
}
```

В начале первой функции мы очищаем стек объектов сцены — myScene, удаляем саму сцену, обнуляем глобальные переменные, отменяем запланированное действие.

Во второй функции создаем игровые объекты: сцену, делая ее главной, фон, элементы управления, пользовательский космический корабль, астероиды, НЛО, вызывая соответствующие функции.

13 Запускаем и отлаживаем игру

Все готово: арт подготовлен, весь код написан! Теперь можно запустить нашу игру. Это можно сделать из среды Torsion, нажав клавишу F5, или, в том случае, если вы не используете Torsion, то надо сохранить все файлы со скриптовым кодом и запустить исполняемый файл движка. Он соберет все скриптовые файлы и преобразует код на языке Torque Script, находящийся в них, в двоичный код, сохранив результат в оперативную память. Если ошибок и опечаток не допущено, то в обоих случаях на экране монитора появится окно с игрой Asteroids на движке Torque 2D!

Если присутствуют ошибки и/или опечатки, запуск игры прервется, а для их исправления удобно воспользоваться средой программирования Torsion. Она выводит во вкладку Output, находящуюся внизу окна информацию обо всех ошибках, замеченных движком, а так же всю остальную диагностическую информацию: об инициализации графиче-

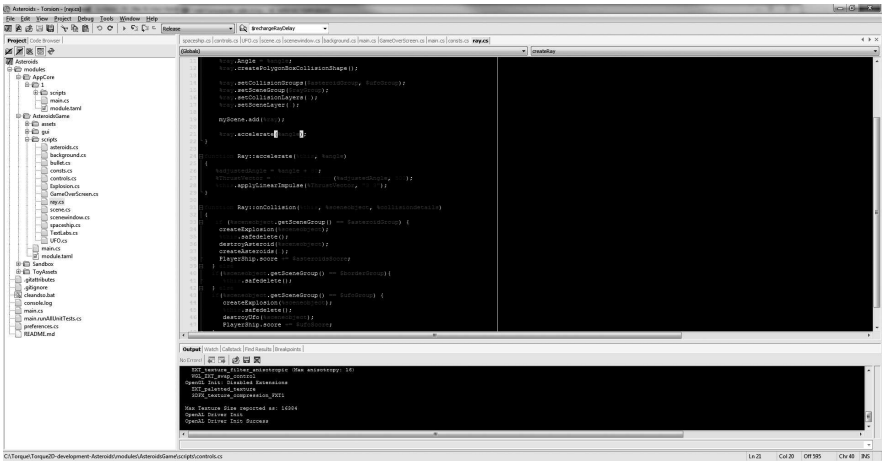

Рис. 6.7. Окно редактора Torsion

ских, аудио устройств и т.д. Дважды щелкнув на какой-либо ошибке, Torsion сразу переведет курсор на ее место в коде (рис. 6.7).

Рис. 6.8. Файл console.log

Когда Torsion не используется, эти же сведения можно обнаружить в файле console.log, создаваемом движком (рис. 6.8).

Можно увидеть вывод движка, но нельзя интерактивно использовать его для навигации по коду, поэтому Torsion незаменимый инструмент разработчика на Torque!

Лучшая практика при разработке игр, это, закончив написание класса, сразу протестировать получившуюся игру. Пускай она не готова, зато можно проверить последний добавленный класс.

14 Распространение игры

Когда игра разработана, свой труд, возможно, захочется кому-то показать: маме, папе, сестре, девушке, удаленным друзьям, а может вы захотите выложить ее на свой веб-сайт. В главе 2 мы рассмотрели содержимое папки с движком. А сейчас мы увидим: какие файлы нужны для запуска и работы определенной игры на компьютере с операционной системой Windows пользователя, не являющимся разработчиком данной игры. Для примера продолжим рассмотрение нашей игры Asteroids.

Можно создать отдельный каталог, который назвать, к примеру, Asteroids-BUILD, и в него скопировать всю папку modules, где находятся: арт, ассеты и скрипты нашей игры, исполняемый файл Asteroids.exe и файлы динамических библиотек: Leap.dll — содержит код для работы с сенсором Leap Motion, даже если ваша игра не использует данную функциональность, эта библиотека все равно должна присутствовать, так как без нее игра не запустится. Файл Leapd.dll нужен для работы отладочной версии приложения. Еще нам нужна динамическая библиотека OpenAL, она содержит код для работы с аудиоданными посредством одноименного интерфейса программирования приложений. Если в игре не используется воспроизведение музыки/звуков, рекомендуется включить этот файл, и, хотя без него игра успешно запуститься, в случае его отсутствия, движок выведет в лог сообщение об отсутствии файла. Для работы новых версий движка Torque 2D динамическая библиотека unicows.dll не нужна, поскольку из современных версий Torque 2D исключена поддержка старых операционных систем семейства Windows (Windows 9x). А динамическая библиотека unicows.dll нужна для поддержки Unicode в этих операционных системах. Также нужен скриптовый файл main.cs. Когда пользователь запускает игру — кликнув на ее исполняемом файл, она в первую очередь читает установки, прописанные в файле main.cs.

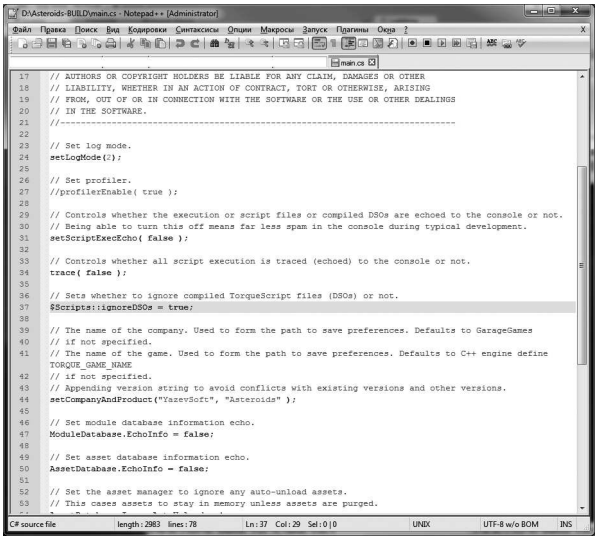

Рис. 6.9. Содержимое файла main.cs

Однако не торопитесь пока распространять свою игру! Нужно совершить еще один подготовительный этап. Весь скриптовый код на языке Torque Script, который мы написали, хранится в открытом виде. Поэтому любой пользователь вашей игры может его прочесть, изменить и вывести игру из рабочего состояния, кроме того, этот пользователь может украсть ваши коммерческие и интеллектуальные тайны. Другое дело, если вы распространяете игру с открытым исходным кодом, в ином случае, вам надо преобразовать исходный код в нечитаемый формат — Торковский двоичный код — dso. Для этого в каком-либо текстовом редакторе откройте файл main.cs. В нем найдите строчку:

```
$Scripts::ignoreDSOs = true;
```

Она сообщает движку игнорировать dso файлы. Ее следует закомментировать (в стиле Torque Script), после чего не забудьте сохранить файл. Теперь запустите игру, в результате она запустится как обычно. Между тем внутри всех подпапок папки modules рядом с *.cs файлами образуются файлы *.cs.dso, содержащие двоичный код первых. По функциональности они полностью аналогичны *.cs файлам.

Обратите внимание: файл main.cs, находящийся в корневом каталоге игры, не претерпел конвертации в dso формат. Этого требует Torque, данный файл должен иметь исходный вид.

Перед распространением игры файлы с исходным кодом надо удалить. Перед этим убедитесь, что у вас сделана резервная копия игры! Иначе исходный код будет нельзя восстановить!

В папке modules создайте текстовый файл с именем cleanCS.bat. Введите в него такую строчку:

```
del /s *.cs
```

Затем сохраните файл. Тем самым у нас получился обычный командный файл операционной системы Windows. Команда, введенная в файл, производит удаление всех *.cs файлов в текущем и всех дочерних каталогах! Будьте осторожны при ее использовании! После ее использования командный файл можно удалить.

Теперь можно спокойно распространять свою игру, не беспокоясь о своих наработках на языке Torque Script!

Рис. 6.10. Готовая игра Asteroids на PC

15 Заключение

В этой главе мы с вами совершили большой шаг вперед и разработали свою игру на движке Torque 2D. Мы использовали множество инструментов: графических, текстовых редакторов и редакторов кода (такие как Torsion). Мы выполнили не простую задачу: создали законченную игру с главным персонажем, препятствиями, врагами, оружием, диагностирующими надписями и меню (рис. 6.10)!

В следующей главе мы рассмотрим взаимодействия движка Torque 2D с мобильными платформами!

Сходите, отдохните, попейте чаю с тортиком, вы это заслужили и возвращайтесь к чтению и разработке видео игр!

Рис. 6.11. Проигрыш — меню

Глава 7. Torque 2D и мобильные платформы

Оглавление

В прошлой главе мы разработали аркадную игру Asteroids для платформы PC и операционной системы Windows. В текущей главе мы рассмотрим кроссплатформенные возможности движка Torque 2D.

1 OS X

Начнем с операционной системы для настольных компьютеров фирмы Apple — OS X. Поддержка этой системы унаследована еще от платной версии двумерного движка от GarageGames — iTorque 2D.

В сентябре 2016-го года Apple выпустила новую версию операционной системы для своих компьютеров — macOS Sierra. Однако для портирова-

ния игры я использовал предыдущую версию операционной системы — OS X El Capitan, поэтому ниже я буду рассказывать о ней. В новой версии системы различия при осуществлении портирования отсутствуют.

Скопируем на компьютер макинтош всю папку с движком, если ваша игра находится в другой папке, тогда ее тоже надо скопировать. Откройте среду программирования Xcode. Я использую версию 7.1, хотя сейчас доступна версия 8.1, при портировании игры в ней не будет различий. Файл проекта Torque2D.xcodeproj находится в подкаталоге /engine/compilers/Xcode папки с движком.

Если вы используете Torque 2D версии 3.2, смело открывайте файл для загрузки проекта в Xcode (рис. 7.1).

Запустите компиляцию. Процесс сборки должен завершиться без ошибок. Но в итоге получиться большое количество предупреждений. На выходе создается исполняемый файл для операционной системы OS X — Torque2D.app.

Если вы используете Torque 2D 3.3, в таком случае нужны предварительные приготовления. Иначе во время линковки проекта поя-

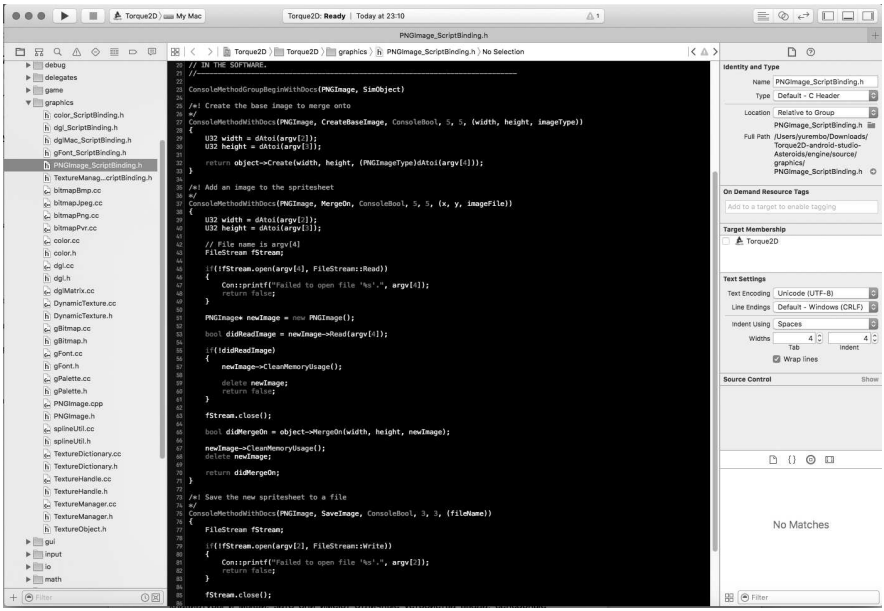

Рис. 7.1. Проект Torque 2D, открытый в среде программирования Xcode

вится ошибка, и исполняемый файл не будет создан. В сообщение об ошибке указано, что линковщик не может найти библиотеку libogg.dylib. Проблема заключается в том, что в Torque 2D 3.3 была добавлена поддержка проигрывание звука с помощью аудио библиотеки Vorbis. Она не включена в репозиторий, поэтому ее установка происходит отдельно от движка. Установка Vorbis выполняется с помощью HomeBrew. Это менеджер недостающих пакетов для OS X, как написано на сайте разработчика. Следовательно, для начала надо установить сам HomeBrew. Он находится на сайте: http://brew.sh/index_ru.html, для его установки достаточно ввести в терминале следующую команду (в одну строчку):

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.
githubusercontent.com/Homebrew/install/master/
install)"
```

После инсталляции менеджера пакетов HomeBrew можно переходить к установке аудио библиотеки Vorbis. Для этого введите в терминал:

```
brew install libogg libvorbis --universal
```

Эта команда установит необходимые файлы и пропишет ссылки, таким образом, Xcode будет знать, куда установлены нужные библиотеки.

Если вы используете версию 3.3 движка из ветки development (самую последнюю) и приведенное выше решение не помогло (во время компиляции возникают ошибки), попробуйте скачать и воспользоваться Torque 2D 3.3 из ветки master.

Скопируйте исполняемый файл движка — Torque2D.app в папку с нашей игрой Asteroids, там переименуйте его в Asteroids.app и запустите. Игра

Рис. 7.2. Терминал в OS X

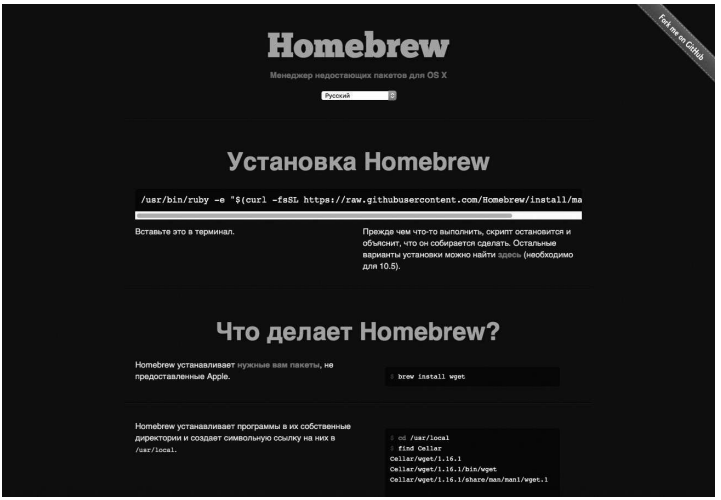

Рис. 7.3. Сайт HomeBrew

должна работать так же, как в операционной системе Windows на платформе PC! При этом, чтобы добиться этого нам надо было только перекомпилировать движок: скрипты и ассеты остались без изменения (рис. 7.4)!

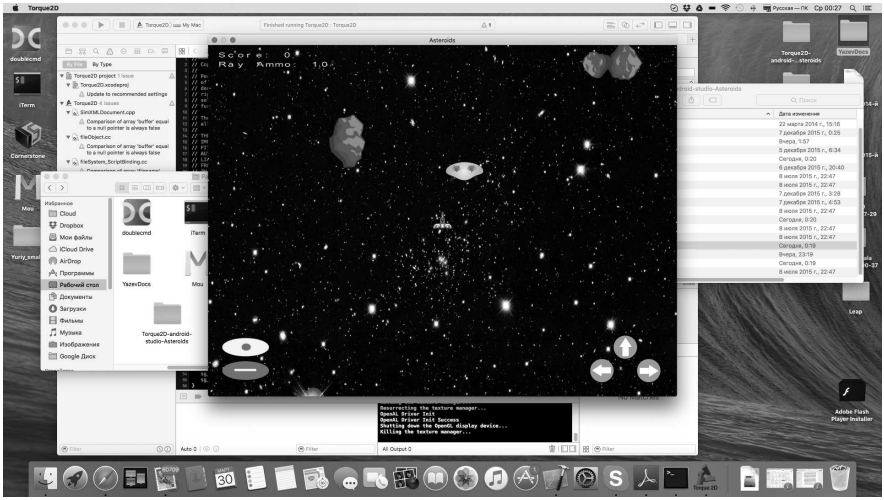

Рис. 7.4. Asteroids в OS X

2 Доработка игры для запуска и работы на мобильных платформах

Мобильные платформы: смартфоны, планшеты и т.д. в отличие от настольных компьютеров и ноутбуков для своей миниатюрности лишены клавиатуры и мыши, зато они имеют отличные устройства ввода: сенсорные экраны. От того, что у мобильных устройств нет клавиатуры, теряется способ управления космическим кораблем в нашей игре. Один из способов реализации управления аркадными играми на мобильных устройствах с сенсорным экраном является создание виртуальных кнопок — виртуального джойстика. То есть кнопки представляются объектами движка (спрайтами) на экране устройства. Взаимодействие с ними и реакция на их нажатие происходит так же, как с реальными клавишами и кнопками. Создадим такой способ управления для нашей игры. Для начала нарисуем кнопки в любом удобном графическом редакторе. Также изображения можно взять из материалов к книге. Они находятся в подкаталоге \modules\AsteroidsGame\assets\controlButs\ игры (рис. 7.5).

Изображения со стрелками являются аналогами клавиш со стрелками на клавиатуре. Разместим их в правом нижнем углу экрана. Изображения с кружком и лучом означают оружие, соответственно, пулю и лазерный луч. Поместим эти две кнопки в левый нижний угол.

Код для загрузки всех 5-ти картинок примерно одинаковый, отличается только именами загружаемых файлов:

```
<ImageAsset
   AssetName="butLeftArrow"
   ImageFile="butLeftArrow.png" />
```

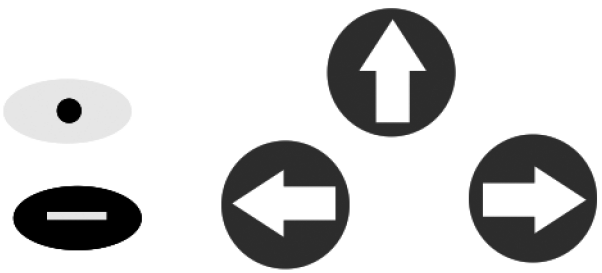

Рис. 7.5. Изображения виртуальных кнопок

В операционной среде Windows откроем Torsion (в том случае, если успели его закрыть), для редактирования откроем файл controls.cs. Дополним его следующим кодом для создания спрайтов, представляющих кнопки:

```
function InputManager::dirControls(%this)
{
    %but = new Sprite(butUp)
    {
   Class = "butUpArrow";
   Position = "39 -27";
   Size = "5 5";
   SceneLayer = 1;
   Image = "AsteroidsGame:butUpArrow";
   Visible = true;
   BodyType = static;
   UseInputEvents = true;
    };
    myScene.add(%but);

    %but = new Sprite(butLeft)
    {
   Class = "butLeftArrow";
   Position = "34 -32";
   Size = "5 5";
   SceneLayer = 0;
   Image = "AsteroidsGame:butLeftArrow";
   Visible = true;
   BodyType = static;
   UseInputEvents = true;
    };
    myScene.add(%but);

    %but = new Sprite(butRight)
    {
   Class = "butRightArrow";
   Position = "44 -32";
   Size = "5 5";
   SceneLayer = 0;
   Image = "AsteroidsGame:butRightArrow";
   Visible = true;
   BodyType = static;
   UseInputEvents = true;
    };
    myScene.add(%but);

    %but = new Sprite(butBulletObj)
    {
```

```
   Class = "butBullet";
   Position = "-42 -27";
   Size = "10 4";
   SceneLayer = 0;
   Image = "AsteroidsGame:butBullet";
   Visible = true;
   BodyType = static;
   UseInputEvents = true;
    };
    myScene.add(%but);

    %but = new Sprite(butRayObj)
    {
   Class = "butRay";
   Position = "-42 -32";
   Size = "10 4";
   SceneLayer = 0;
   Image = "AsteroidsGame:butRay";
   Visible = true;
   BodyType = static;
   UseInputEvents = true;
    };
    myScene.add(%but);
}
```

Заметьте, при создании каждой кнопки ее булево свойство UseInputEvents устанавливается в true. Это означает, что данный игровой элемент (в данном конкретном случае Sprite) будет отвечать на события ввода. Мы будем использовать «прикосновение», неважно, чем оно выполнено: курсором мыши или пальцем на сенсорном экране. Вот еще одно замечательное свойство Торка! Одно событие реагирует на ввод с разных устройств.

Следующим действием нам надо написать обработчики нажатия этих кнопок:

```
function butUpArrow::onTouchDown(%this, %touchId,
%worldposition)
{
   if (isObject(PlayerShip))PlayerShip.accelerate();
}

function butUpArrow::onTouchUp(%this, %touchId,
%worldposition)
{
   if (isObject(PlayerShip))PlayerShip.stopthrust();
}
```

```
function butLeftArrow::onTouchDown(%this, %touchId,
%worldposition)
{
   if (isObject(PlayerShip))PlayerShip.turnleft();
}

function butLeftArrow::onTouchUp(%this, %touchId,
%worldposition)
{
   if (isObject(PlayerShip))PlayerShip.stopturn();
}

function butRightArrow::onTouchDown(%this, %touchId,
%worldposition)
{
   if (isObject(PlayerShip))PlayerShip.turnright();
}

function butRightArrow::onTouchUp(%this, %touchId,
%worldposition)
{
   if (isObject(PlayerShip))PlayerShip.stopturn();
}

function butBullet::onTouchDown(%this, %touchId,
%worldposition)
{
   if (isObject(PlayerShip))PlayerShip.Fire(0);
}

function butRay::onTouchDown(%this, %touchId,
%worldposition)
{
   if (isObject(PlayerShip))PlayerShip.Fire(1);
}
```

Так как для каждой кнопки мы присвоили свой класс, то, чтобы написать обработчик события определенной кнопки мы вводим:

function <имя класса определенной кнопки>::onTouchDown (параметры)

или, чтобы написать обработчик события отпускания кнопки — вводим:

function <имя класса определенной кнопки>::onTouchUp(параметры)

Все вызываемые действия мы уже рассматривали в главе 6 при реализации ввода с клавиатуры, здесь используются те же самые вызовы.

Поскольку рассматриваемые кнопки не физические, мы не можем контролировать отпускание конкретной кнопки, пользователь может нажать какую-то кнопку, а палец отпустить, выйдя за ее пределы; поэтому нам надо реализовать возможное отпускание в любой точке экрана. Весь экран (или окно) в нашей игре занимает фон — объект класса Background, следовательно, нам надо для него обработать событие отпускания касания — onTouchUp, в котором остановить вращение и ускорение космического корабля. В Torsion откройте файл backgrounds.cs и в его конец добавьте следующий обработчик события:

```
function Background::onTouchUp(%this, %touchId,
%worldposition)
{
   if (isObject(PlayerShip)){
      PlayerShip.stopthrust();
      PlayerShip.stopturn();
   }
}
```

На этом доработка завершена. В ее процессе главная преследуемая цель заключалась в изменение способа управления игрой для сенсорных экранов мобильных устройств, все остальные заботы по созданию кроссплатформенных игр Torque 2D берет на себя!

3 iOS

Поддержка мобильной операционной системы от компании Apple iOS была изначально включена в Torque 2D еще со времен его предка — платного продукта от GarageGames — iTorque 2D.

В среде разработки Xcode установите последние версии эмуляторов для iOS 9.1 (или 10.1 в случае, если у вас Xcode 8.1). Так как у меня нет физического устройства под управлением операционной системы iOS, для тестирования игр и приложений для iPhone/iPad я использую эмуляторы. Из папки с движком откройте файл /engine/compilers/Xcode_iOS/Torque2D.xcodeproj.

После загрузки проекта перекомпилируйте его. Успех или неудача равнозначны с описанием из раздела 1 данной главы. Для вспоминания рекомендую обратиться к этому разделу. Ошибки никто не отменял, поэтому надо следить, как за обновлениями Torque 2D, так и инструментами разработки от Apple.

Если построение завершилось успешно, можно выкладывать игру в App Store!

4 Android

Поддержка платформы Android была добавлена только, начиная с третьей версии движка, наравне с добавленными в нее поддержками операционной системы Linux и отображением графического представления на web-страницах.

До версии Torque 2D 3.2 для компиляции игр под Android в репозитории движка присутствовал только проект, ориентированный на среду разработки Eclipse. Однако еще в 2013-м году компания Google на основе открытой среды разработки от JetBrains IntelliJ IDEA разработала систему программирования Android Studio для создания приложений под Android. Теперь Google рекомендует использовать именно её. Это естественно, поскольку Android Studio гораздо удобнее, чем Eclipse, и первая специально предназначена для проектов под Android.

Между тем в версию Torque 2D 3.2 был включен проект для Android Studio, и теперь стало возможным компилировать движок под платформу Android с помощью этой среды программирования!

Установка Android Studio

Для начала, если у вас не установлена виртуальная машина Java, ее необходимо установить. Вся операционная система Android основана на Java. Однако Android выполняет приложения на особой виртуальной Java-машине — Dalvik. Последняя использовалась до 5-й версии операционной системы, а начиная с 5-й версии, виртуальная машина Dalvik была заменена на ART (Android Runtime). Таким образом, программист разрабатывает прикладные приложения, используя стандартный пакет инструментов Java Development Kit от Oracle, а при подготовке приложения для распространения на Android, инструменты из Android SDK (пакета разработчика от Google) преобразуют откомпилированные java-программы из стандартного байт-кода в формат, пригодный для виртуальной машины Android. Рекомендуется использовать Java SE Development Kit 7 (http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html).

Установка не представляет ничего сложного: после запуска инсталляции (путем двойного щелчка по скачанному с указанной странице исполняемому файлу) мастер установки проведет вас по необходимым шагам.

Чтобы сторонние программы могли использовать установленный пакет для компиляции Java-кода необходимо добавить в системные пути

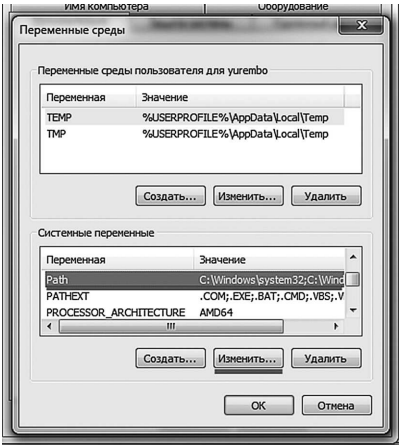

Рис. 7.6. Окно «Переменные среды»

ссылки на каталоги пакета Java-инструментов. Для этого откройте свойства Компьютера (пункт находится в контекстном меню, вызываемом щелчком правой кнопкой мыши по значку или элементу меню «Компьютер»), перейдите на «Дополнительные параметры системы», в открывшемся окне на вкладке «Дополнительно» щелкните по кнопке «Переменные среды». В появившемся окне в списке «Системные переменные» выделите переменную Path, ниже нажмите кнопку «Изменить» (рис. 7.6).

В появившемся окне «Изменение системной переменной» текст поля «Значение переменной:» через точку с запятой надо дополнить двумя путями к подпапкам bin папок jdk<номер версии> и jre<номер версии> каталога Java, разделенные так же точкой с запятой — «;». В моем случае это такая строка, включающая оба пути: «C:\Program Files\Java\jdk1.7.0_79\bin;C:\Program Files\Java\jre7\bin».

Чтобы проверить: правильно ли прописаны пути у вас в системе, откройте «Командную строку» и последовательно введите туда команды:

```
java –version
```

нажмите Enter, затем:

```
javac –version
```

Enter.

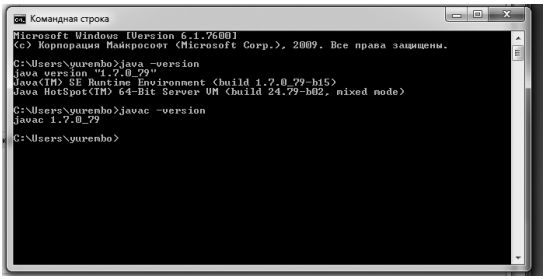

Рис. 7.61. Командная строка в действии

После каждого нажатия Enter должна отобразиться версия указанной программы (рис. 7.61).

Если за место версии выводится текст: ««имя программы» не является внутренней или внешней командой, исполняемой программой или пакетным файлом», значит, пути прописаны некорректно, и вам надо еще раз их перепроверить.

Теперь установим Android Studio: http://developer.android.com/intl/ru/sdk/index.html. В установке студии так же нет ничего сложного.

Дополнительно для разработки под Android нам понадобятся соответствующие SDK и NDK. Чтобы их установить надо в запущенной Android Studio выбрать пункт меню: Tools -> Android -> SDK Manager (рис. 7.7).

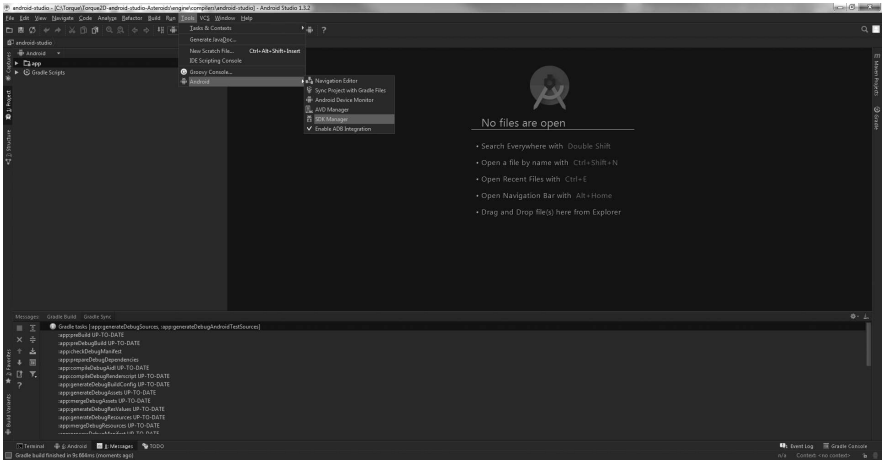

Рис. 7.7. меню

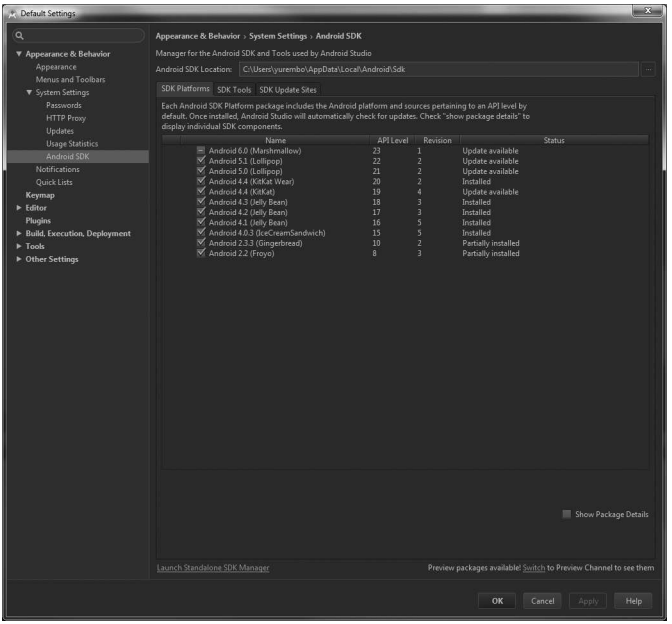

Рис. 7.8. окно настроек — sdk manager

Откроется окно настроек, где будет предложено задать путь для установки sdk. Обратите внимание, чтобы в пути к папке с заголовками и дополнительными библиотеками (SDK и NDK) отсутствовали пробелы. Если для Eclipse это было не существенно, то для Android Studio важно. Ниже находятся 3 вкладки: на первой — SDK Platforms предлагается выбрать устанавливаемые платформы — нужные версии операционных систем, для которых надо установить пакеты разработки. Следует выбрать целевые платформы. Можно выбрать все, чтобы иметь возможность создавать приложения для всех версий операционной системы Android. На второй вкладке — SDK Tools выбираются дополнительные инструменты, устанавливаемые в комплекте с SDK. Это такие инструменты: средства построения для Android, документация по SDK, USB драйвер, Web драйвер, эмулятор от Intel, Android NDK и другое. На вкладке SDK Update Sites указываются сайты и их адреса, где содержатся обновления для инструментов разработки (рис. 7.8).

После установки SDK и NDK потребуется добавить для них системные переменные в окне «Переменные среды», рассмотренном нами ранее.

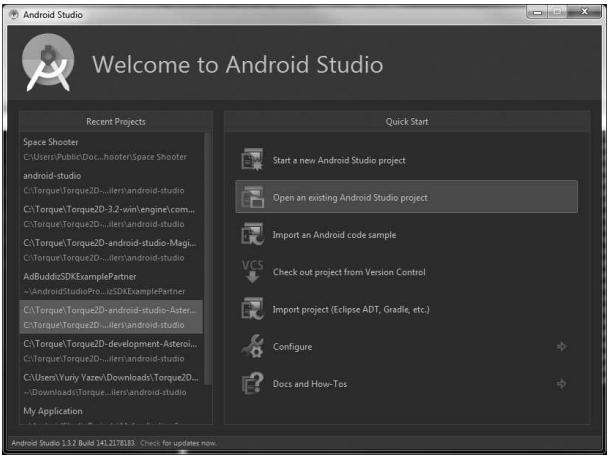

Рис. 7.9. Android Studio Launcher

Для SDK переменная должна называться ANDROID_SDK_ROOT, для NDK надо добавить 2 переменные с одинаковым значением: ANDROID_NDK_PATH и NDK_ROOT. Значениями этих переменных должны быть пути к папкам расположения соответствующих инструментальных пакетов. На моем компьютере — это, соответственно: C:\Users\yurembo\AppData\Local\Android\Sdk и C:\Users\yurembo\AppData\Local\Android\Sdk\ndk-bundle.

Компиляция Asteroids под Android

Запустите Android Studio. В появившемся окне выберите пункт «Open an existing Android Studio project» (рис. 7.9).

В новом окне будет предложено выбрать каталог, в котором находится проект Android Studio конкретной игры, в моем случае окно и путь выглядят следующим образом: C:\Torque\Torque2D-android-studio-Asteroids\engine\compilers\android-studio (рис. 7.10).

После нажатия OK, выбранный проект загрузится в студию. Можно начинать компиляцию и построение движка, дополнительных настроек в текущей версии Torque 2D выполнять не нужно. Хотя возможно надо уменьшить выставленный по умолчанию номер версии API, требуемый для работы приложения. Эта настройка находится в файле app/manifests/AndroidManifest.xml. Строки, которые нам нужно изменить находятся в начале файла:

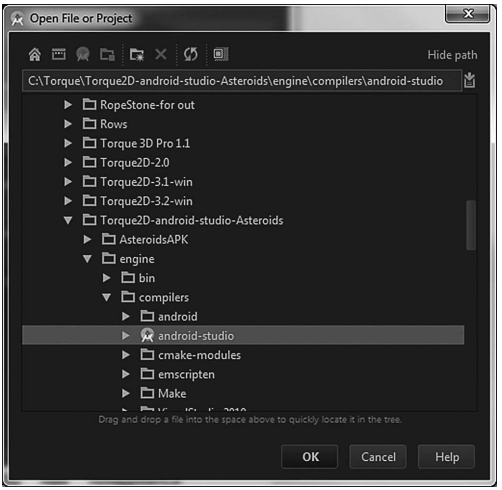

Рис. 7.10. открытие проекта

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.garagegames.torque2d"
   android:versionCode="1"
   android:versionName="3.0" xmlns:android="http://
schemas.android.com/apk/res/android">

   <uses-permission android:name="android.permission.
INTERNET" />
   <uses-permission android:name="com.android.vending.
CHECK_LICENSE" />
   <uses-permission android:name="android.permission.
ACCESS_WIFI_STATE"/>
   <uses-permission android:name="android.permission.
VIBRATE"/>

   <uses-sdk
      android:minSdkVersion="14"
      android:targetSdkVersion="19" />
```

Рассмотрим подробнее интересующие нас строчки:

```
android:versionCode="1"
```

в этой строке указывается номер версии приложения, этот номер очень важен для магазина Google Play, поскольку, если вы решите заме-

нить уже загруженное приложение его новой версией и не поменяете эту цифру, магазин откажет в загрузке;

```
android:versionName="3.0"
```

версия движка, декоративный номер;

```
android:minSdkVersion="14"
android:targetSdkVersion="19"
```

в этих строчках указывается, соответственно, минимальная и целевая версии API необходимые для функционирования игры; по умолчанию во второй из этих строк стоял номер 23: `android:targetSdkVersion="23"`. Так как у меня планшет имеет версию API под номером 19, Android Studio отказался запускать игру на моем устройстве. Тогда я изменил номер целевой версии на 19, и Android Studio пропустил мою игру, которая впоследствии прекрасно стала работать на моем планшете. Тем не менее, если у вас более новое устройство, соответствующее начальным требованиям, вам ничего менять не надо (рис. 7.11).

Также проверьте строки в начале файла build.gradle (Module: app), он находится в свертке Gradle Scripts:

```
android {
```

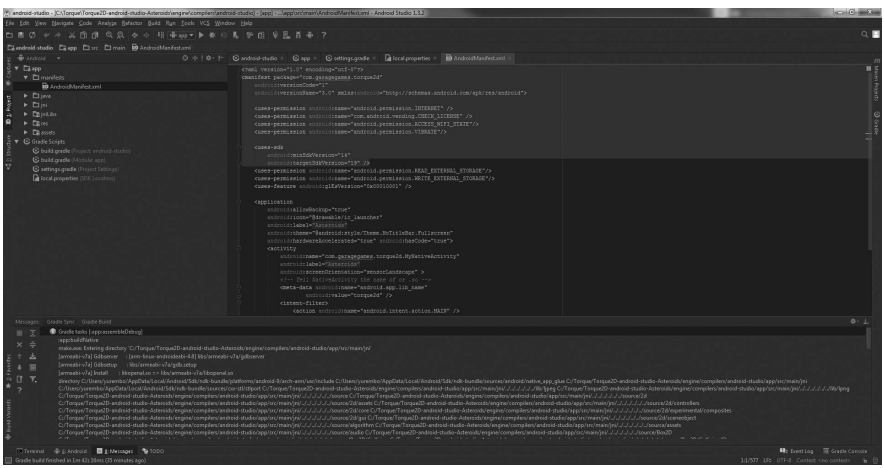

Рис. 7.11. Манифест

```
    compileSdkVersion 23
    buildToolsVersion "23.0.2"

    defaultConfig {
        applicationId "com.garagegames.torque2d"
        minSdkVersion 23
        targetSdkVersion 23
```

Числа 23 (во 2-й и 2-х последних строчках) надо заменить на номер имеющегося на вашем устройстве уровня API, в моем случае — это 19:

```
android {
    compileSdkVersion 19
    buildToolsVersion "23.0.2"

    defaultConfig {
        applicationId "com.garagegames.torque2d"
        minSdkVersion 19
        targetSdkVersion 19
```

Замечу еще один момент. В версии движка Torque 2D 3.2 до обновления (27 декабря 2015-го года) перед началом процесса компиляции надо исправить некоторые параметры для системы построения Gradle. Таким образом, чтобы осуществить компиляцию движка из-под ОС Windows надо в файле build.gradle (Module: app) строки $ndkDir/ndk-build дополнить расширением .cmd . Это нужно по той причине, что в OS X и Unix-системах командные файлы не имеют расширений, следовательно, путь к ним не содержит расширения. С другой стороны в Windows командные файлы имеют расширения cmd (или bat), поэтому нам необходимо исправить путь к ним, иначе система построения Gradle не найдет командные файлы.

Однако после указанной даты в ветке development появилось обновление, где скрипт построения проекта с помощью системы Gradle автоматически определяет операционную систему, в которой выполняется, и использует соответствующие имена файлов. Кроме того в этом обновление исправлено множество ошибок и предупреждений, появляющихся во время компиляции Android-проекта. Естественно из-за них происходил сбой в процессе компиляции. Тем не менее, при построении для других платформ, в частности, для Windows компиляция проходила без проблем. Между тем, как сказано выше теперь в ветке development имеются решения для всех проблем.

Дополнительно надо выполнить такую настройку. Так как наше приложение не имеет активности, то, следовательно, его запускать не надо

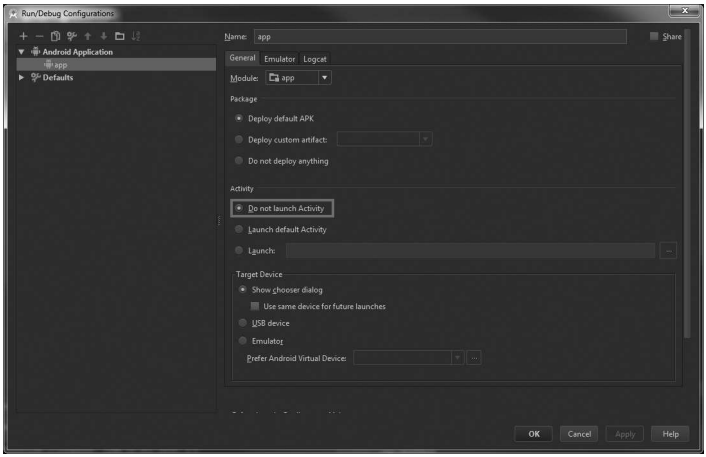

Рис. 7.12. Отключение запуска активности

и об этом надо сообщить Android Studio. Откройте меню: Run -> Edit Configurations. В открывшемся окне в разделе Activity поставьте переключатель рядом с пунктом: «Do not launch Activity» (рис. 7.12). Тем самым мы сообщили Android Studio, что никакую активность запускать не надо.

При работе с Android Studio перед запуском проекта иногда может всплывать подобное окно с выбором активности. Всегда выбирайте пункт «Do not launch Activity», так как игра на движке Torque 2D не имеет активности.

Для начала построения проекта нажмите кнопку Run (зеленый треугольник, повернутый вправо) (рис. 7.13).

В результате запустится процесс компиляции и построения движка для платформы Android. На мобильном устройстве с этой операционной системой могут быть установлены различные микропроцессоры, например: Intel X86 или ARM. В любом случае построение проекта должно завершиться без ошибок. Для трансляции C/C++ кода используется компилятор GCC. Его можно сменить, например, на Intel C/C++.

В итоге компиляции будет предложено запустить готовый продукт на выполнение (рис. 7.14).

В появившемся окне будет предложено выбрать среди созданных эмуляторов и подключенных к компьютеру реальных устройств. Перед началом процесса построения проекта я подключил свой планшет

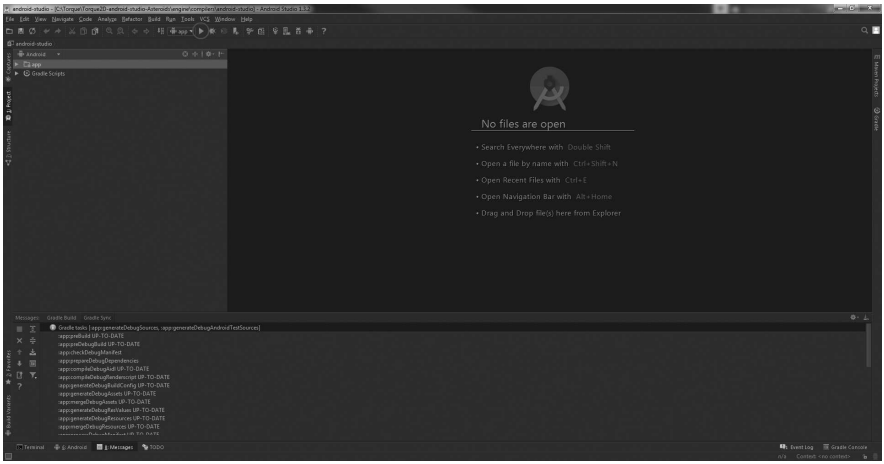

Рис. 7.13. Начало построения проекта

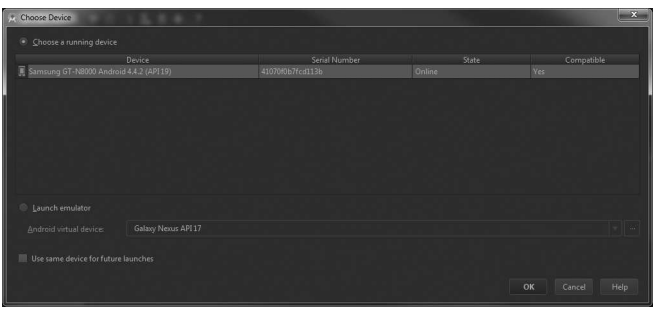

Рис. 7.14. Запуск приложения на выполнение

Samsung GT-N8000 с операционной системой Android 4.4.2 (API 19) с процессором ARM. Эмуляторы для Android-устройств работают очень медленно, и, похоже, на них не получится запустить игру. Поэтому для запуска я выбираю реальное устройство.

В нижней части Android Studio (на вкладке Run) будут отображены команды, выполняемые для отправки и установки нашей игры на подключенное устройство. Затем на его экране отобразится стандартная заставка (рис. 7.15).

Рис. 7.15. Заставка Torque 2D

После загрузки приложения на экране появится наша игра! Управление космическим кораблем осуществляется посредством виртуальных кнопок (рис. 7.16).

Примечание. Возможно, при запуске Торковской игры на Android-устройстве за место некоторых спрайтов отобразится оранжевый пря-

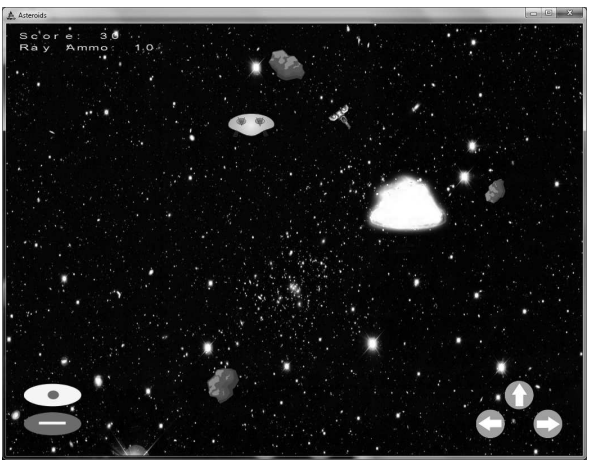

Рис. 7.16. Asteroids с виртуальными кнопками

моугольник с надписью CANNOT RENDER. Чтобы решить эту проблему скопируйте из корневой папки с движком из подкаталога modules директорию Sandbox в подпапку modules с вашей игрой. Каталог Sandbox занимает всего половину мегабайта и включает все необходимое для решения описанной выше проблемы. Он подхватится движком автоматически.

Для повторного запуска игры на Android-устройстве надо выполнить построение проекта, но не его компиляцию. Для этого удалите содержимое подкаталога: \engine\compilers\android-studio\app\src\main\ каталога с движком. После этого в Android Studio запустите компиляцию, студия обнаружит, что все файлы уже скомпилированы, а изменений нет, поэтому запустит процесс линковки откомпилированных исходников вместе с тем возьмет новое содержимое каталога с игрой (с добавленной папкой Sandbox), поместит это содержимое в указанный выше подкаталог, из чего соберет APK файл — исполняемый файл для операционной системы Android. Итоговый файл будет находиться в поддиректории: \engine\compilers\android-studio\app\build\outputs\apk\. Там находятся как отладочные, так и релизные версии игры, в зависимости от настроек построения. Отладочные и релизные версии игры требуют разных процессов компиляции.

5 Windows Phone

Windows Phone входит в тройку самых популярных операционных систем для мобильных устройств. Тем не менее, к сожалению, Torque 2D не имеет поддержки этой платформы. Этому есть веская причина: визуализатор Torque 2D построен на базе OpenGL ES (расширенная версия OpenGL, предназначенная для встраиваемых/мобильных систем), тогда как Windows Phone, в отличие от iOS и Android, визуализирует с помощью DirectX. Последнее утверждение касается Windows Phone 8 и Windows 10 Mobile, однако Windows Phone 7 отображает графику с помощью XNA, другими словами, управляемом DirectX. Тем не менее, WP7 можно пренебречь, так как это устаревшая мобильная операционная система. Зато WP8 и Windows 10 Mobile очень важные системы, они визуализируют с помощью, соответственно, DirectX 11 и DirectX 12. Отсюда следует, чтобы добавить в движок Torque 2D поддержку платформы Windows Phone надо написать новый визуализатор на основе DirectX. Это очень сложное решение, так как движок целиком пронизан поддержкой OpenGL.

ANGLE

Тем не менее, есть решение лучше! В 2010-м году Google начала открытый проект по переносу графических приложений с OpenGL ES на DirectX, этот проект получил название ANGLE. В 2013-м году подразделение Microsoft Open Technologies включилось в этот проект. Результатом этого стало ответвление репозитория на GitHub, поддерживаемого Microsoft: https://github.com/MSOpenTech/angle.

Проект ANGLE — Almost Native Graphics Layer Engine — движок почти нативного графического слоя позволяет пользователям Windows бесшовно запускать OpenGL ES приложения на аппаратуре, работающей с DirectX 11. Это достигается путем преобразования вызовов с OpenGL ES API на DirectX 11 API.

ANGLE полностью поддерживает следующие 3 типа приложений:

1) Универсальные приложения для Windows 10 (Universal Windows apps)

2) Приложения для Windows 8.1 и для Windows Phone 8.1

3) Классические приложения для рабочего стола Windows (Windows desktop applications)

Поддержка разных версий OpenGL ES зависит от оборудования, на котором выполняется приложение.

Уровень возможностей оборудования	Пример устройства	Какие версии OpenGL ES поддерживает ANGLE?
11_1 11_0 10_1 10_0	Современные настольные компьютеры Планшет Microsoft Surface	OpenGL ES 2.0 Частично OpenGL ES 3.0
9_3	Смартфоны Windows Phone	OpenGL ES 2.0 (с некоторыми исключениями)
9_2 9_1	Планшеты Microsoft Surface RT	OpenGL ES 2.0 (через программную эмуляцию)
None	Raspberry Pi 2	OpenGL ES 2.0 (через программную эмуляцию)

«Уровень возможностей оборудования» соответствует способности видеоадаптера в поддержке определенной версии DirectX. Таким образом, им определяется набор возможностей графического процессора.

Воспользоваться ANGLE можно тремя способами:

1) Через заготовки приложений в MS Visual Studio 2015;

2) Скачать скомпилированные бинарники через NuGet;

3) Скачать исходный код с GitHub репозитория и самостоятельно скомпилировать ANGLE;

Для использования ANGLE в зависимости от версии операционной системы вам понадобятся:

1) Если операционная система Windows 10, то вам понадобится Visual Studio 2015 для возможности разработки универсальных приложений Windows;

2) При Windows 8.1 и/или Windows Phone 8.1 понадобится Visual Studio 2013 с четвертым обновлением;

3) Для разработки классических приложений подойдет Windows 7 вместе с Visual Studio 2013 Community;

Подробное рассмотрение работы с такой большой технологией, как ANGLE выходит за рамки данной книги. Я коротко рассказал о ней в виду того, как может быть реализована поддержка Windows Phone в движке Torque 2D без полного переписывания визуализатора.

6 Заключение

В этой главе мы узнали о кроссплатформенных возможностях движка Torque 2D, его выполнении в разных операционных системах для настольных компьютеров и разных мобильных платформах. Мы так же увидели возможное направление развития движка и применимые для этого инструменты.

Портировав игру Asteroids на Android, мы, при этом не написали ни строчки кода на языке Java, в итоге все для нас сделал Torque 2D и его замечательная заготовка проекта для Android! Мы только использовали язык Torque Script, который одинаково интерпретируется движком под все поддерживаемые платформы.

В этой главе мы рассмотрели построение проекта нашей игры для платформы Android. Мы проделали довольно много работы, адаптировали PC-версию нашей игры для управления с сенсорных экранов, разобрались с мобильными платформами, скомпилировали и запустили нашу игру на устройстве под управлением операционной системы Android.

В следующей главе речь пойдет об автоматизации создания Торковских проектов.

Глава 8. Менеджер проектов для Torque 2D

Оглавление

Чтобы создать независимую от SandBox'а игру на движке Torque 2D нужно проделать утомительные операции по созданию папок, копированию файлов и их изменению для соответствия имени нового проекта (я рассказывал об этой последовательности операций в главе 3). Для создания каждого нового проекта эти действия приходится выполнять снова и снова. Выполнение одних и тех же операций оказывает на программиста демотивирующий эффект. Поэтому то, что можно автоматизировать, надо автоматизировать. Созданием новых проектов должен заниматься менеджер проектов. Так как в Torque 2D таковой отсутствует, я решил его разработать. В первую очередь для своих собственных нужд. А так же для участников торковского сообщества и всех остальных пользователей движка. Кроме того я выложил на GitHub все исходники, поэтому любой программист может скачать код и модифицировать его под свои цели. Для ускорения разработки я написал Менеджер проектов на C#, воспользовавшись dotNet 4.5.

Я очень надеюсь, что менеджер проектов снизит порог вхождения для новых пользователей Torque 2D, которым сходу мало что понятно, в том числе, как создать новый проект. Менеджер проектов создает минимальное торковское приложение, в котором, кроме инициализации канвы содержатся объекты: Scene, SceneWindow, объект управления — InputManager и, для примера, спрайт с натянутой текстурой, размещенный в центре экрана. Дополнительно менеджер проектов может создать Torsion-проект для данной конкретной игры, который впоследствии можно открыть с помощью Torsion IDE и удобно редактировать код на Torque Script.

1 Обзор Project Manager: создание проекта

Менеджер проектов представляет классическое Windows-приложение с минималистическим интерфейсом (рис. 8.1).

Ничто так не вдохновляет, как релиз новой версии Torque 2D, что отражено в заголовке окна, но на самом деле менеджер проектов будет работать с любой версией движка.

Итак, для того чтобы создать новую игру на Torque 2D с помощью менеджера проектов надо:

1) в поле «Project name» ввести имя будущей игры;

2) в поле «Company name» ввести имя компании-разработчика;

3) в поле «Torque 2D root folder» ввести путь к папке, куда установлен движок Torque 2D, так же можно нажать кнопку «Select folder» и выбрать нужную папку в диалоге;

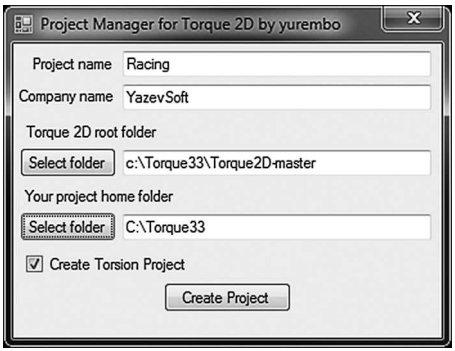

Рис. 8.1. Менеджер проектов

4) в поле «Your project home folder» ввести путь к папке, в которой будет создан проект пользовательской игры, кроме того можно нажать кнопку «Select folder», располагающейся рядом с этим полем и выбрать нужную папку с помощью диалога; в ней будет создана подпапка с именем проекта — «Project Name»;

5) ниже можно отметить или снять (по умолчанию отмечен) флажок «Create Torsion Project», в случае, если он отмечен, тогда будет создан Torsion проект, который можно открыть с помощью Torsion IDE и редактировать скриптовый код на Torque Script с помощью последней;

6) после нажатия кнопки Create Project, проект новой минимальной игры на движке Torque 2D будет создан в выбранном каталоге, о чем известит появившееся внизу окна надпись;

Некоторые файлы менеджер копирует (и изменяет) из корневой папки Torque 2D, некоторые файлы он создает сам.

За место выбора корневой директории с Torque 2D, пользователь может поместить исполняемый файл менеджера проектов в один каталог с движком. Тогда при запуске менеджера, он сам определит свое месторасположение и заполнит соответствующее текстовое поле (Torque 2D root folder).

В результате создания проекта с помощью менеджера получается довольно незамысловатая игра, представляющая собой фундамент для дальнейшего развития (рис. 8.2).

Скачать менеджер проектов для Torque 2D можно с моей страницы GitHub: https://github.com/yurembo/ProjectManager или взять из материалов к книге.

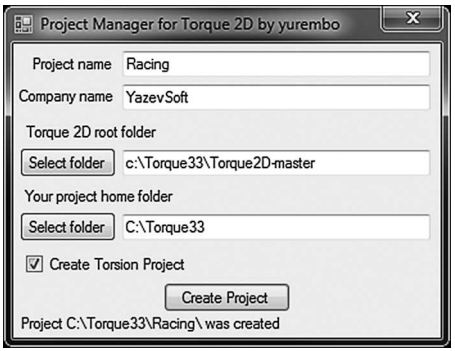

Рис. 8.2. Проект Racing создан

Рис. 8.3. Содержимое папки проекта

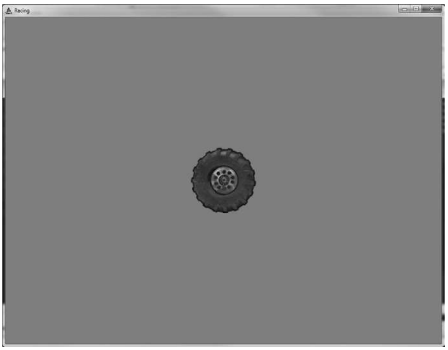

Рис. 8.4. Минимальная Torque 2D игра

2 Обзор исходного кода Project Manager

Как отмечено выше: я разработал Project Manager на языке C# в среде Visual Studio 2013 Community с использованием .NET 4.5. Откроем в студии проект ProjectManager.sln, который можно взять или из материалов к книге, или скачать с GitHub (по ссылке, приведенной выше). Если вы хотите создать свой менеджер проектов, то в моем случае это приложение типа Windows Forms Application.

Вначале откройте визуальное представление формы — файл Form1.cs (рис. 8.5).

На форме находятся 5 объектов класса Label из пространства имен System.Windows.Forms, к слову, все остальные визуальные компоненты принадлежат этому же пространству имен. Также на форме расположены 4 поля ввода (объекты класса TextBox) для внесения имени проекта, имени компании — разработчика, корневой папки с движком и папки с создаваемым проектом. Ниже находится переключатель (CheckBox) «Create Torsion Project». Кроме того на форме расположены 3 кнопки (объекты класса Button). Две из них служат для открытия диалога выбора папки: для корневой папки движка и домашней папки создаваемого проекта, а третья — с надписью «Create Project» служит для создания проекта.

Посмотрим на исходный код. Первое на что стоит обратить внимание — это обработчик события щелчка на кнопке buttonCreateProject_Click:

```
private void buttonCreateProject_Click(object sender,
EventArgs e)
```

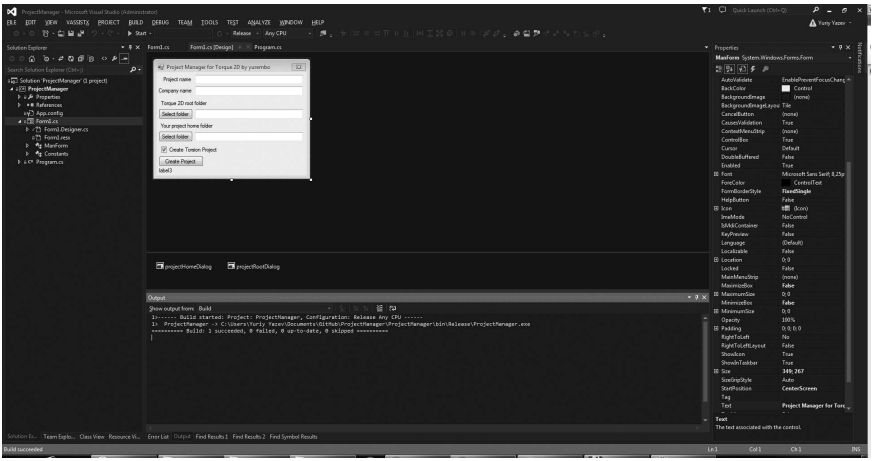

Рис. 8.5. Visual Studio 2013 Community Edition

```
{
   labelTotal.Visible = false;
   projDir = projectHomeDialog.SelectedPath;
   rootDir = textRootFolder.Text + '\\';
   projDir += '\\' + textProjectName.Text + '\\';
    projName = textProjectName.Text;
    if (rootDir != "" && projDir != "" && projName !=
""")
    {
       createT2DProject();
    }
    else MessageBox.Show("Check text fields");
}
```

Сначала мы убеждаемся, что надпись внизу окна, повествующая нам о том, что предыдущий проект создан, невидима, другими словами, делаем ее невидимой. Затем в глобальную переменную projDir (где хранится путь к каталогу нашего проекта) типа string мы сохраняем выбранный с помощью объекта projectHomeDialog класса FolderBrowserDialog путь к домашнему каталогу проекта. Чтобы закончить этот путь мы добавляем в его конец обратный слеш '\' (экранированный еще одним обратным слешем, чтобы компилятор не принял его за контрольный символ), название проекта (введенное пользователем в текстовое поле textProjectName) и ставим еще один обратный слеш. В глобальную переменную rootDir типа string (здесь хранится путь к каталогу с движком)

помещаем текст из поля ввода textRootFolder дополнительно, добавив в конец обратный слеш. Глобальной переменной projName присваиваем содержимое поля ввода textProjectName. После этого проверяем, чтобы все 3 заполненные переменные не были пусты, в таком случае вызываем функцию createT2DProject. В ином случае функцией Show объекта MessageBox выводим сообщение: «Check text fields» (проверьте текстовые поля).

Функция createT2DProject не получает и не возвращает никакие аргументы:

```
 private void createT2DProject()
{
   if (Directory.Exists(projDir))
   {
      MessageBox.Show("Directory exists");
   }
   else
   {
      Directory.CreateDirectory(projDir);
      labelTotal.Text = "Project " + projDir + " was
created";
      Directory.SetCurrentDirectory(projDir);
      if (copyFiles())
          labelTotal.Visible = true;
   }
}
```

В начале функции проверяется, существует ли директория по пути, ранее сохраненному в глобальной переменной projDir, если ответ положительный, чего, по сути, быть не должно, мы выводим сообщения о существовании папки и заканчиваем выполнение. Ведь, если папка соответствующая переданному пути существует, значит, пользователь, скорее всего, по ошибке или не внимательности создает проект повторно, и этот момент нам необходимо предотвратить. Если же такой каталог отсутствует, тогда с помощью статического метода CreateDirectory класса Directory мы создаем папку, путь к которой указан в переменной projDir. После этого присваиваем текстовой метке, отображаемой внизу окна, текст: «Project » + projDir + « was created». Далее, с помощью метода SetCurrentDirectory класса Directory делаем каталог projDir текущим (рабочим). После этого вызываем метод copyFiles. Он возвращает булево значение: если оно истинное, значит, метод успешно отработал, и мы можем показать надпись о создании проекта.

Как мы увидели выше: метод copyFiles возвращает булево значение, при этом ничего не принимает:

```
private bool copyFiles()
{
   if (File.Exists(rootDir + Constants.exeName))
   {
       File.Copy(rootDir + Constants.exeName, projDir
+ Constants.exeName, true);
       File.Move(projDir + Constants.exeName, projDir
+ projName + ".exe");
    }
    else
    {
        notExistFile(Constants.exeName);
        return false;
    }
    if (File.Exists(rootDir + Constants.leapName))
    {
        File.Copy(rootDir + Constants.leapName,
projDir + Constants.leapName, true);
    }
    else
    {
        notExistFile(Constants.leapName);
        return false;
    }
        if (File.Exists(rootDir + Constants.
openALName))
        {
           File.Copy(rootDir + Constants.openALName,
projDir + Constants.openALName, true);
        }
        else
        {
            notExistFile(Constants.openALName);
            return false;
        }
/* устаревший код, в версии Torque 2D 3.3 данного
файла не существует
        if (File.Exists(rootDir + Constants.uniName))
        {
           File.Copy(rootDir + Constants.uniName,
projDir + Constants.uniName, true);
        }
        else
        {
            notExistFile(Constants.uniName);
```

```
            return false;
        }
*/
        if (File.Exists(rootDir + Constants.script_
main))
        {
            File.Copy(rootDir + Constants.script_main,
projDir + Constants.script_main, true);
            modifyMainFile(projDir + Constants.script_
main);
        }
        else
        {
            notExistFile(Constants.script_main);
            return false;
        }
        if (flag_CreateTorsionProj.Checked)
createTorsionProject();
        Directory.CreateDirectory("modules");
        Directory.SetCurrentDirectory("modules");
        Directory.CreateDirectory("AppCore");
        projDir += "modules";
        DirectoryInfo di = new DirectoryInfo(Path.
Combine(rootDir, "modules", "AppCore", "1"));
        FileInfo[] files = di.GetFiles();
        foreach (FileInfo file in files)
        {
            string destFile = Path.Combine(projDir,
"AppCore", file.Name);
            file.CopyTo(destFile, true);
            if (file.Name == "main.cs")
            {
                modifyProjMainFile(destFile);
            }
        }
        di = new DirectoryInfo(Path.Combine(rootDir,
"modules", "AppCore", "1", "scripts"));
        files = di.GetFiles();
        Directory.CreateDirectory(Path.
Combine("AppCore", "scripts"));
        foreach (FileInfo file in files)
        {
            string destFile = Path.Combine(projDir,
"AppCore", "scripts", file.Name);
            file.CopyTo(destFile, true);
        }
        Directory.CreateDirectory(projName);
        createProjMainCS();
        createProjModuleTAML();
```

```
        Directory.CreateDirectory(Path.
Combine(projName, "gui"));
        createProjGuiProfiles();
        Directory.CreateDirectory(Path.
Combine(projName, "assets"));
        if (File.Exists(Path.Combine(rootDir,
"modules\\ToyAssets\\1\\assets\\images\\tires.png")))
        {
            File.Copy(rootDir + "modules\\
ToyAssets\\1\\assets\\images\\tires.png", projDir +
"\\" + projName + "\\assets\\tires.png", true);
        }
        else
        {
            notExistFile("tires.png");
            return false;
        }
        if (File.Exists(Path.Combine(rootDir,
"modules\\ToyAssets\\1\\assets\\images\\tires.asset.
taml")))
        {
            File.Copy(rootDir + "modules\\
ToyAssets\\1\\assets\\images\\tires.asset.taml",
projDir + "\\" + projName + "\\assets\\tires.asset.
taml", true);
        }
        else
        {
            notExistFile("tires.asset.taml");
            return false;
        }
   Directory.CreateDirectory(Path.Combine(projName,
"scripts"));
   createProjScriptsControlCS();
   createProjScriptsSceneCS();
   createProjScriptsSceneWindowCS();
   createProjScriptsSpriteCS();
   return true;
}
```

Разберем код метода. Первое, что бросается в глаза — это класс Constants. На самом деле этот статический класс является своего рода контейнером для констант:

```
public static class Constants
   {
   //public const string rootDir = "c:\\Torque
// \\Torque2D-development-3-2\\";
```

```
    public const string exeName = "Torque2D.exe";
    public const string leapName = "Leap.dll";
    public const string openALName = "OpenAL32.dll";
    public const string uniName = "unicows.dll";
    public const string script_main = "main.cs";
   }
```

Константы представляют имена файлов, к которым будет обращаться приложение.

В начале метода copyFiles, с помощью статического метода Exist класса File происходит проверка на существование файла Constants.exeName (по умолчанию исполняемый файл движка — Torque2D.exe) в директории, путь к которой лежит в глобальной переменной rootDir (как вы помните, в ней хранится путь к корневому каталогу с движком). Если файл имеет место, мы копируем его из папки с движком в папку со своим проектом. Это действие осуществляется с помощью статического метода Copy класса File: в 1-м параметре методу передается путь к файлу источнику, во 2-м — путь к файлу — приемнику. Затем с помощью метода Move мы переименовываем целевой файл. Этот метод так же получает 2 параметра: прежнее имя файла, новое имя файла. Как вы заметили по названию, этот метод используется для перемещения файлов между папками, но, если операция производится в одном и том же каталоге, то файл переименовывается. В случае если файл отсутствует, вызывается метод notExistFile. Ему передается имя несуществующего файла, после чего происходит прекращение выполнения метода copyFiles с возвращением значения false, что знаменует о неудачном выполнении метода.

Метод notExistFile имеет следующий вид:

```
private void notExistFile(string fileName)
{
  MessageBox.Show("File is not exists " + fileName);
}
```

Он принимает имя файла, а занимается только тем, что выводит сообщение об отсутствии указанного в параметре файла.

Эта последовательность операций выполняется еще над четырьмя файлами:

1) Leap.dll — служит для работы с устройством Leap Motion — является обязательным;

2) OpenAL32.dll — содержит код для работы с аудио библиотекой OpenAL — является обязательным;

3) unicows.dll — позволяет работать с Unicode в устаревших системах Windows — является устаревшим, поэтому удален из движка версии 3.3; менеджер проектов так же обновлен для соответствия развитию движка;

4) main.cs — главный скриптовый файл, с него начинается выполнение игры — является обязательным;

Только в случаях этих фалов не надо производить переименование. Между тем после копирования файла main.cs происходит вызов метода modifyMainFile. Этот метод осуществляет модификацию данного скриптового файла, так как содержимое последнего является простым текстом. Код метода представлен так:

```
private void modifyMainFile(string fileName)
{
   string[] lines = File.ReadAllLines(fileName);
   if (lines.Length > 2)
   {
  for (int i = 2; i < lines.Length; ++i)
  {
        lines[i] = lines[i].Replace("\"Torque 2D\"",
'"' + projName + '"');
        lines[i] = lines[i].Replace("\"GarageGames\"",
'"' + textCompanyName.Text + '"');
  }
        File.Delete(fileName);
        File.WriteAllLines(fileName, lines);
   }
}
```

В параметре метод принимает имя файла для редактирования. В первой строчке с помощью статического метода ReadAllLines класса File открывается текстовой файл, заданный параметром fileName, из файла считываются все строки, помещаются в массив строк — string[] lines, затем файл закрывается. Когда массив строк заполнен, мы проверяем, чтобы в нем находилось больше двух элементов. Если условие совпадает, тогда начинает выполняться цикл, который перебирает все строки (содержимое массива), начиная с третьей (счет от нуля) и заканчивая последней; в каждой строке слово «Torque2D» вместе с кавычками заменятся на название нашего проекта, сохраненного в переменной projName. Это выполняется посредством метода Replace объекта класса string, метод получает 2 параметра: заменяемую строку и подставляемый за место нее текст. Следующим действием в этой же строке, в случае присутствия, заменяется слово «GarageGames» (вместе с кавычками) на название нашей компании, введенное в текстовое поле

textCompanyName. Почему мы начинаем проверять массив текста, начиная с третьей строчки? Во второй строке находится имя компании разработчика движка, я решил его не изменять, если вы считаете иначе, можете легко изменить наш алгоритм. После того как мы проверили весь массив текста, мы удаляем исходный файл с помощью статического метода Delete класса File, передав ему имя удаляемого файла. Затем на место удаленного файла мы записываем новый, используя для этого статический метод WriteAllLines класса File. Этот метод принимает 2 параметра: путь + имя нового файла (мы передаем путь и имя прежнего — удаленного файла), массив строк, предназначенных для записи. После создания нового файла и записи в него переданных данных метод закрывает его.

Копированию подлежат только стандартные файлы, но мы хотим добавить дополнительную функциональность. Например, если пользователь пожелает (и установит в окне соответствующий флажок), в корневой папке пользовательского проекта надо создать torsion-проект, чтобы дать пользователю возможность удобно работать с исходным кодом. Эта проверка происходит в следующей строчке:

```
if (flag_CreateTorsionProj.Checked)
createTorsionProject();
```

Итогом успешной проверки вызывается метод createTorsionProject. Мы уже знаем (см. главу 3 о создании заготовки игры), что Torsion-проект описан на языке XML, мы также знаем какими сведениями нужно заполнить описание проекта. Поэтому, нам не составит труда автоматизировать этот процесс.

Итак, в менеджере проектов метод createTorsionProject выглядит следующим образом:

```
private async void createTorsionProject()
{
   StringBuilder sb = new StringBuilder();
   sb.AppendLine("<TorsionProject>");
   sb.AppendLine("<Name>" + projName + "</Name>");
   sb.AppendLine("<WorkingDir/>");
   sb.AppendLine("<EntryScript>main.cs</
EntryScript>");
   sb.AppendLine("<DebugHook>dbgSetParameters( #port#,
\"#password#\", true );</DebugHook>");
   sb.AppendLine("<Mods>");
   sb.AppendLine("<Folder>modules</Folder>");
   sb.AppendLine("</Mods>");
```

```
   sb.AppendLine("<ScannerExts>cs; gui</
ScannerExts>");
   sb.AppendLine("<Configs>");
   sb.AppendLine("<Config>");
   sb.AppendLine("<Name>Release</Name>");
   sb.AppendLine("<Executable>" + projName + ".exe</
Executable>");
   sb.AppendLine("<Arguments/>");
   sb.AppendLine("<HasExports>true</HasExports>");
   sb.AppendLine("<Precompile>false</Precompile>");
   sb.AppendLine("<InjectDebugger>true</
InjectDebugger>");
   sb.AppendLine("<UseSetModPaths>false</
UseSetModPaths>");
   sb.AppendLine("</Config>");
   sb.AppendLine("<Config>");
   sb.AppendLine("<Name>Debug</Name>");
   sb.AppendLine("<Executable>" + projName + "_debug.
exe</Executable>");
   sb.AppendLine("<Arguments/>");
   sb.AppendLine("<HasExports>false</HasExports>");
   sb.AppendLine("<Precompile>false</Precompile>");
   sb.AppendLine("<InjectDebugger>true</
InjectDebugger>");
   sb.AppendLine("<UseSetModPaths>false</
UseSetModPaths>");
   sb.AppendLine("</Config>");
   sb.AppendLine("</Configs>");
   sb.AppendLine("<SearchURL/>");
   sb.AppendLine("<SearchProduct>main</
SearchProduct>");
   sb.AppendLine("<SearchVersion>HEAD</
SearchVersion>");
sb.AppendLine("<ExecModifiedScripts>true</
ExecModifiedScripts>");
   sb.AppendLine("</TorsionProject>");
   using (StreamWriter outfile = new StreamWriter(Path.
Combine(projDir, projName + ".torsion"), true))
   {
      await outfile.WriteAsync(sb.ToString());
   }
}
```

Сперва создается объект sb типа StringBuilder. Класс StringBuilder представляет собой контейнер редактируемых строк. Таким образом, мы можем использовать этот класс для добавления и сохранения множества строк, каждую из которых мы в любой момент можем изменить. После создания объекта мы посредством его метода AppendLine добав-

ляем в него строки, из которых состоит Torsion-проект. Метод принимает параметр — добавляемую строку. Обратите внимание, в тех местах, где нужно поместить имя нашей игры и/или исполняемого файла, а этот файл называется так же, как наша игра, мы ставим значение глобальной переменной projName, например:

```
sb.AppendLine("<Executable>" + projName + ".exe</
Executable>");
```

В остальном, исходник Torsion-проекта такой же, как в главе 3. После помещения в объект класса StringBuilder всех необходимых строк нам надо сохранить его содержимое в файл. Это осуществляется с помощью такого куска кода:

```
using (StreamWriter outfile = new StreamWriter(Path.
Combine(projDir, projName + ".torsion"), true))
{
   await outfile.WriteAsync(sb.ToString());
}
```

Конструкция using гарантирует, что после ее завершения открытый внутри нее файл будет закрыт. А внутри нее создается файловый поток StreamWriter, который позволяет записать в файл символы в выбранной кодировке. Конструктор объекта в качестве параметров получает:

1. путь + имя файла для записи. В данном случае для формирования пути мы используем статический метод Combine класса Path. Этот метод принимает произвольное количество строковых параметров, представляющих собой имена папок. В результате своей работы метод выдает строку — путь к файлу со специфическими для операционной системы, в которой в данный момент выполняется программа разделителями между именами каталогов;

2. булево значение: если оно положительно, и одноименный файл уже создан, тогда он будет дополнен.

Конструктор класса StringBuilder имеет 7 перегруженных версий.

После создания и открытия файла в него асинхронно (с помощью метода WriteAsync) записывается содержимое объекта sb. Обратите внимание: этот оператор предваряет ключевое слово await. С помощью него родительский метод не возвращает значение, пока не завершится выполнение асинхронной задачи. В таком случае родительский метод должен быть объявлен с использованием ключевого слова async:

```
private async void createTorsionProject()
```

Возвратимся в метод copyFiles. После вызова createTorsionProject происходит создание папки modules (метод CreateDirectory), переход в нее (метод SetCurrentDirectory) и создание подпапки AppCore. Заметьте: мы находимся в папке modules.

Создадим объект di типа DirectoryInfo. Этот класс содержит методы для работы с папками и их содержимым. Его конструктор поучает путь к целевой папке. Для формирования пути мы снова используем метод Combine класса Path, передавая параметры: путь к корневой папке с движком — rootDir, плюс в каждом следующем параметре вложенная подпапка. В итоге имеем \modules\AppCore\1. Следующим действием получим из этого каталога все файлы с помощью метода GetFiles объекта класса DirectoryInfo, заполним ими массив типа FileInfo и последовательно скопируем каждый файл. Это осуществляется в цикле foreach. Таким образом, из массива последовательно выбирается каждый файл, далее, происходит формирование пути и имени целевого файла так же посредством Path.Combine. Затем для файла (объекта класса FileInfo) вызывается метод copyTo, он принимает путь к целевому файлу и булево значение, если оно — true, тогда методу разрешается перезаписать файл. Метод не принимает исходный файл, так как им является вызывающий объект. Если мы натыкаемся на файл с именем main.cs (такой файл должен присутствовать), то после копирования мы редактируем целевой файл. Этим занимается метод modifyProjMainFile. Он похож на рассмотренный выше метод modifyMainFile, поэтому для экономии пространства мы не будем его рассматривать, только отмечу: за место Torque 2D ставится имя проекта (projName), за место loadGroup — loadExplicit, а вместо gameBase так же имя проекта.

Затем весь вышеописанный процесс осуществляется для содержимого подпапки scripts папки AppCore. Для экономии книжного пространства мы не будем его обсуждать.

Как вы помните, мы находимся в папке modules. Начиная оттуда, мы создаем папку «имени нашей игры» — projName. Затем с помощью методов:

```
createProjMainCS();
createProjModuleTAML();
```

мы создаем управляющие скрипты. Можно сказать, что они уникальные для нашей заготовки.

Метод createProjMainCS выглядит следующим образом:

```
private async void createProjMainCS()
{
   StringBuilder sb = new StringBuilder();
   sb.AppendLine("function " + projName +
"::create(%this)");
   sb.AppendLine("{");
   sb.AppendLine("      exec(\"./gui/guiProfiles.
cs\");");
   sb.AppendLine("      exec(\"./scripts/scene.
cs\");");
   sb.AppendLine("      exec(\"./scripts/scenewindow.
cs\");");
   sb.AppendLine("      exec(\"./scripts/InputManager.
cs\");");
   sb.AppendLine("      exec(\"./scripts/sprite.
cs\");");
   sb.AppendLine("      createSceneWindow();");
   sb.AppendLine("      createScene();");
   sb.AppendLine("      mySceneWindow.
setScene(myScene);");
   sb.AppendLine("      createSprite();");
   sb.AppendLine("      new
ScriptObject(InputManager);");
   sb.AppendLine("      InputManager.Init_
controls();");
   sb.AppendLine("}");
   sb.AppendLine("");
   sb.AppendLine("function " + projName +
"::destroy(%this)");
   sb.AppendLine("{");
   sb.AppendLine("      destroySceneWindow();");
   sb.AppendLine("      InputManager.delete();");
   sb.AppendLine("}");
   using (StreamWriter outfile = new StreamWriter(Path.
Combine(projDir, projName, Constants.script_main),
true))
   {
      await outfile.WriteAsync(sb.ToString());
   }
}
```

Как мы видим: он создает файл, в котором описывается основной геймплейный класс игры. В конструкторе подключаются скриптовые файлы, составляющие основу (заготовку) игры:

— guiProfiles.cs — файл описания профиля для интерфейса;

— scene.cs — содержит описание игровой сцены (создание, разрушение);

— scenewindow.cs — содержит описание класса окна приложения;

— InputManager.cs — описание менеджера управления;
— sprite.cs — определение класса спрайта;

Затем происходит вызов функций для создания окна, создания игровой сцены, спрайта, менеджера управления, его инициализации. В деструкторе происходит уничтожение всех объектов. В конце концов подготовленный скрипт записывается в файл (выше мы подробно рассмотрели процесс записи).

Следующим вызывается метод createProjModuleTAML:

```
public async void createProjModuleTAML()
{
    StringBuilder sb = new StringBuilder();
    sb.AppendLine("<ModuleDefinition");
    sb.AppendLine("ModuleId=\"" + projName + "\"");
    sb.AppendLine("VersionId=\"1\"");
    sb.AppendLine("Description=\"" + projName + "
folder\"");
    sb.AppendLine("");
    sb.AppendLine("ScriptFile=\"main.cs\"");
    sb.AppendLine("CreateFunction=\"create\"");
    sb.AppendLine("DestroyFunction=\"destroy\">");
    sb.AppendLine("");
    sb.AppendLine("<DeclaredAssets");
    sb.AppendLine("Path=\"assets\"");
    sb.AppendLine("Extension=\"asset.taml\"");
    sb.AppendLine("Recurse=\"true\"/>");
    sb.AppendLine("");
    sb.AppendLine("</ModuleDefinition>");
    using (StreamWriter outfile = new
StreamWriter(Path.Combine(projDir, projName, "module.
taml"), true))
    {
       await outfile.WriteAsync(sb.ToString());
    }
}
```

Метод создает taml-скрипт, который направляет движок на выполнение в скрипт main.cs. Также в нем указываются конструктор и деструктор.

Далее, в папке «имени проекта» создается подкаталог gui, в нем с помощью метода createProjGuiProfiles создается профиль пользовательского интерфейса:

```
private async void createProjGuiProfiles()
{
```

```
    StringBuilder sb = new StringBuilder();
    sb.AppendLine("if(!isObject(GuiDefaultProfile)) new
GuiControlProfile (GuiDefaultProfile)");
    sb.AppendLine("{");
    sb.AppendLine("Modal = true;");
    sb.AppendLine("};");
    using (StreamWriter outfile = new
StreamWriter(Path.Combine(projDir, projName, "gui",
"guiProfiles.cs"), true))
    {
       await outfile.WriteAsync(sb.ToString());
    }
}
```

Затем в одноименной с проектом директории создается подпапка assets. В нее копируются изображение и описывающий ее ассет из папки с движком:

```
if (File.Exists(Path.Combine(rootDir, "modules\\
ToyAssets\\1\\assets\\images\\tires.png")))
{
    File.Copy(rootDir + "modules\\ToyAssets\\1\\
assets\\images\\tires.png", projDir + "\\" + projName
+ "\\assets\\tires.png", true);
}
else
{
    notExistFile("tires.png");
    return false;
}
if (File.Exists(Path.Combine(rootDir, "modules\\
ToyAssets\\1\\assets\\images\\tires.asset.taml")))
{
    File.Copy(rootDir + "modules\\ToyAssets\\1\\
assets\\images\\tires.asset.taml", projDir + "\\" +
projName + "\\assets\\tires.asset.taml", true);
}
else
{
    notExistFile("tires.asset.taml");
    return false;
}
```

В случае проблем с копированием, выводим окошко, которое извещает, какой файл не удалось скопировать (метод notExistFile).

После того как необходимая графика будет добавлена в нашу заготовку, нам остается добавить геймплейные скрипты. Для этого в той же пап-

ке создадим подкаталог scripts. В нем нам надо создать 4 уникальных (отсутствующих в корневом каталоге с движком) геймплейных скрипта. Для этого служат 4 метода:

```
createProjScriptsControlCS();
createProjScriptsSceneCS();
createProjScriptsSceneWindowCS();
createProjScriptsSpriteCS();
```

Первый из них создает скрипт — менеджер управления:

```
private async void createProjScriptsControlCS()
{
 StringBuilder sb = new StringBuilder();
 sb.AppendLine("function InputManager::Init_
controls(%this)");
 sb.AppendLine("{");
 sb.AppendLine("new ActionMap(" + projName +
"Input);");
 sb.AppendLine("   " + projName + "Input.
bindCmd(keyboard, \"escape\", \"quit();\", \"\");");
 sb.AppendLine("   " + projName + "Input.push();");
 sb.AppendLine("}");
 sb.AppendLine("");
 sb.AppendLine("function InputManager::destroy()");
 sb.AppendLine("{");
 sb.AppendLine("   " + projName + "Input.delete();");
 sb.AppendLine("}");
 using (StreamWriter outfile = new StreamWriter(Path.
Combine(projDir, projName, "scripts", "InputManager.
cs"), true))
 {
    await outfile.WriteAsync(sb.ToString());
 }
}
```

В создаваемом скрипте образуются 2 метода: конструктор и деструктор менеджера управления. В первом создается карта действий — ActionMap (которую мы назвали: projName + «Input») для управления, в нее помещается прослушиватель нажатия клавиши Escape, в результате чего происходит выход. Деструктор просто удаляет карту действий. Менеджер в заготовке реагирует только на одну реакцию, это сделано для демонстрации возможностей, расширять которые предстоит разработчику для своей игры.

Следующий метод createProjScriptsSceneCS занимается генерацией скрипта, который создает и уничтожает игровую сцену:

```
private async void createProjScriptsSceneCS()
{
   StringBuilder sb = new StringBuilder();
   sb.AppendLine("function createScene()");
   sb.AppendLine("{");
   sb.AppendLine(" if ( isObject(myScene) )");
   sb.AppendLine(" destroyScene();");
   sb.AppendLine(" new Scene(myScene);");
   sb.AppendLine("}");
   sb.AppendLine("");
   sb.AppendLine("function destroyScene()");
   sb.AppendLine("{");
   sb.AppendLine(" if ( !isObject(myScene) )");
   sb.AppendLine(" return;");
   sb.AppendLine(" while(myScene.getCount()) {");
   sb.AppendLine(" %obj = myScene.getObject(0);");
   sb.AppendLine(" if (isObject(%obj)) {");
   sb.AppendLine(" myScene.remove(%obj);");
   sb.AppendLine(" %obj.delete();");
   sb.AppendLine("}");
   sb.AppendLine("}");
   sb.AppendLine("myScene.clear();");
   sb.AppendLine("myScene.delete();");
   sb.AppendLine("}");
   using (StreamWriter outfile = new StreamWriter(Path.
Combine(projDir, projName, "scripts", "scene.cs"),
true))
   {
      await outfile.WriteAsync(sb.ToString());
   }
}
```

Функция createScene проверяет: существует ли сцена, если — да, тогда ее необходимо уничтожить (с помощью функции destroyScene) и создать новую: new Scene(myScene);

По сути, объект класса Scene представляет собой стек, в который помещаются другие объекты, которые необходимо визуализировать. Поэтому, перед уничтожением самого объекта Scene, нам надо удалить каждый содержащийся в нем объект.

Скрипт, созданный в методе createProjScriptsSceneWindowCS, служит для создания (внутри функции createSceneWindow) окна вывода приложения, подключения профиля пользовательского интерфейса, задания контента канвы, чем, как раз и является окно вывода.

```
private async void createProjScriptsSceneWindowCS()
{
```

```
   StringBuilder sb = new StringBuilder();
   sb.AppendLine("$screenWidth = 100;");
   sb.AppendLine("$screenHeight = 75;");
   sb.AppendLine("function createSceneWindow()");
   sb.AppendLine("{");
   sb.AppendLine(" if ( !isObject(mySceneWindow) )");
   sb.AppendLine("{");
   sb.AppendLine(" new SceneWindow(mySceneWindow);");
   sb.AppendLine(" mySceneWindow.Profile =
GuiDefaultProfile;");
   sb.AppendLine(" Canvas.setContent(mySceneWindow);");
   sb.AppendLine("}");
   sb.AppendLine("");
   sb.AppendLine(" mySceneWindow.
setCameraPosition(0,0);");
   sb.AppendLine(" mySceneWindow.
setCameraSize($screenWidth, $screenHeight);");
   sb.AppendLine(" mySceneWindow.setCameraZoom(1.0);");
   sb.AppendLine(" mySceneWindow.setCameraAngle(0);");
   sb.AppendLine(" mySceneWindow.setUseObjectInputEvent
s(true);");
   sb.AppendLine("}");
   sb.AppendLine("");
   sb.AppendLine("function destroySceneWindow()");
   sb.AppendLine("{");
   sb.AppendLine(" if ( !isObject(mySceneWindow) )");
   sb.AppendLine(" return;");
   sb.AppendLine(" mySceneWindow.delete();");
   sb.AppendLine("}");
   using (StreamWriter outfile = new StreamWriter(Path.
Combine(projDir, projName, "scripts", "scenewindow.
cs"), true))
   {
      await outfile.WriteAsync(sb.ToString());
   }
}
```

Кроме того, в функции создания окна к нему подключается камера, для нее устанавливается позиция, размер, масштаб и угол поворота. Последней строчкой активируем окно на возможность получения событий ввода:

```
mySceneWindow.setUseObjectInputEvents(true);
```

Когда окно вывода уничтожается, в методе destroySceneWindow, мы вызываем для него деструктор.

Последний метод — createProjScriptsSpriteCS генерирует скрипт, содержащий только один метод для создания спрайта:

```
private async void createProjScriptsSpriteCS()
{
   StringBuilder sb = new StringBuilder();
   sb.AppendLine("function createSprite()");
   sb.AppendLine("{");
   sb.AppendLine(" %sprite = new Sprite();");
   sb.AppendLine(" %sprite.setBodyType( static );");
   sb.AppendLine(" %sprite.Position = \"0 0\";");
   sb.AppendLine(" %sprite.Size = \"15 15\";");
   sb.AppendLine(" %sprite.SceneLayer = 31;");
   sb.AppendLine(" %sprite.Image = \"" + projName +
":tires\";");
   sb.AppendLine(" myScene.add( %sprite );");
   sb.AppendLine("}");
   using (StreamWriter outfile = new StreamWriter(Path.
Combine(projDir, projName, "scripts", "sprite.cs"),
true))
   {
      await outfile.WriteAsync(sb.ToString());
   }
}
```

После того как спрайт создан, для него устанавливается статический тип, нулевая позиция (0 0), размер 15 × 15, он помещается на 31-й слой, для него из ассета загружается картинка. Наконец, инициализированный спрайт добавляется в стек сцены.

Деструктор в данном случае не нужен, так как удаление спрайта осуществляется при очистке сцены.

Под конец рассмотрим 3 обработчика событий. Первый возникает при загрузке приложения менеджера проектов:

```
private void ManForm_Load(object sender, EventArgs e)
{
  buttonCreateProject.Left = ClientSize.Width / 2 -
buttonCreateProject.Width / 2;
  string folder = Directory.GetCurrentDirectory();
  if (File.Exists(folder + "\\Torque2D.exe"))
            textRootFolder.Text = folder;
}
```

В первой строчке центрируем кнопку «Create Project». Во второй — в переменной folder сохраняем путь к текущей папке, далее, проверяем, если в ней находится файл Torque2D.exe, значит, мы находимся в корне-

вой папке движка, поэтому мы можем поместить путь в соответствующее текстовое поле на форме.

Оба обработчика событий нажатия на кнопки для выбора каталогов движка и пользовательского проекта играют похожую роль: они считывают из соответствующего диалога выбранный путь и помещают его в соответствующие поля редактирования:

```
private void buttonSelectRootFolder_Click(object sender,
EventArgs e)
{
  if (projectRootDialog.ShowDialog() == DialogResult.OK)
  {
    textRootFolder.Text = projectRootDialog.
SelectedPath;
  }
}
private void
buttonSelectHomeFolder_
Click(object sender, EventArgs
e)
{
  if (projectHomeDialog.
ShowDialog() == DialogResult.OK)
  {
    textHomeFolder.Text =
projectHomeDialog.SelectedPath;
  }
}
```

Стандартный диалог выбора папки в операционной системе Windows выглядит следующим образом — рис. 8.6.

Рис. 8.6. Диалог выбора папки

3 Project Manager для OS X

Через некоторое время после разработки и использования менеджера проектов под Windows мне понадобился он для создания проектов в операционной системе OS X, чтобы разрабатывать игры для iOS. Как я упоминал в начале главы, я разработал менеджер с помощью фреймворка .Net 4.5, воспользовавшись содержащимся в нем интерфейсом программирования приложений WinForms. Последний служит для создания пользовательского интерфейса системы Windows. Для операционной среды OS X существует свободная реализация фреймворка .NET под названием Mono. Инициатором и основным разработчиком Mono

является компания Xamarin. Для визуализации элементов пользовательского интерфейса WinForms в Mono используется открытый графический тулкит GTK#. Поэтому мы можем портировать наш менеджер проектов из Windows на OS X, и он так же будет хорошо работать в ней.

Для решения этой задачи у нас есть 2 способа. Рассмотрим оба. Вы сможете применить тот, который вам ближе.

Итак, первый вариант. Сначала установим среду Mono. Для этого перейдите на сайт проекта Mono в раздел Download (http://www.mono-project.com/download/). Для начала скачивания пакета установки для операционной системы OS X выберите слева вкладку Mac OS X (открывается автоматически) и нажмите кнопку Download Mono 32-bit (рис. 8.7).

В результате на жесткий диск вашего мака скачается стандартный установщик — пакет pkg, инсталляция из которого протекает стандартным образом.

После установки среды выполнения надо установить систему программирования Xamarin Studio Community (https://www.xamarin.com/download). После перехода на этот сайт появится форма регистрации, заполнив которую, станет доступна кнопка Download Xamarin Studio

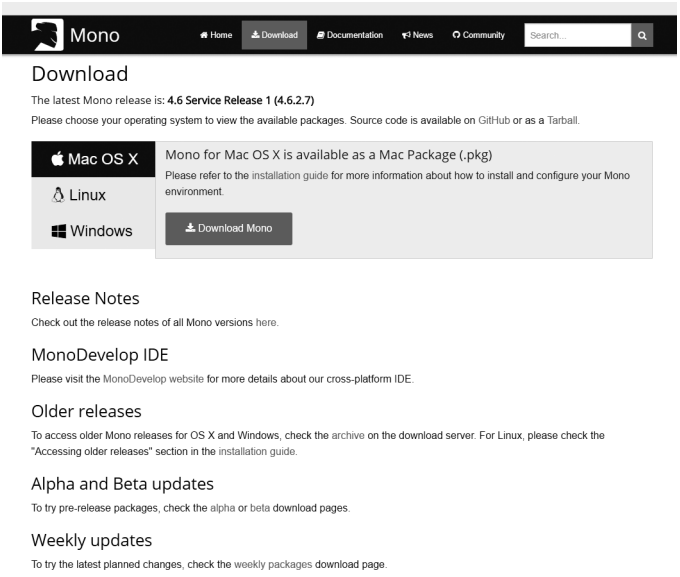

Рис. 8.7. Сайт проекта Mono

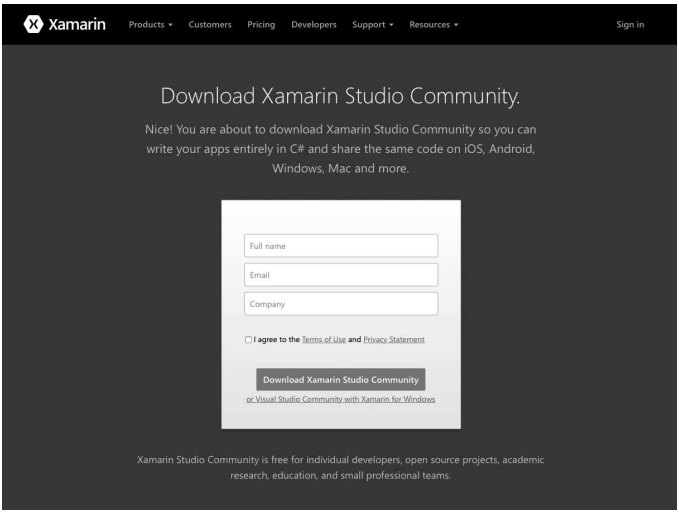

Рис. 8.8. Скачивание Xamarin Studio

Community. В результате нажатия на ней начнется скачивание установщика (рис. 8.8). Инсталляция студии ничем особенным не отличается.

Теперь можно переходить к изменению исходного кода менеджера проектов под Windows для соответствия требованиям OS X. Исходный проект для Windows можно скачать с моего аккаунта GitHub (https://github.com/yurembo/ProjectManager).

Сделайте двойной щелчок на файле ProjectManager.sln, входящем в содержимое проекта менеджера, скачанного с GitHub. В результате решение будет открыто в Xamarin Studio. Сделайте перечисленные ниже модификации. Во-первых, надо изменить символ-разделитель между каталогами с обратного слеша '\', принятого в Windows на слеш '/', принятый в Unix-подобных системах. Это удобно сделать с помощью диалога «Заменить в файлах» (рис. 8.9).

Хотя в некоторых местах кода я применял статический метод Combine класса Path, который принимает неограниченное количество параметров — имен папок через запятую, а возвращает файловый путь, между папками содержащий символ, присущий данной конкретной операционной системе, в которой выполняется приложение. Тем не менее в некоторых местах кода я поленился использовать данную конструкцию,

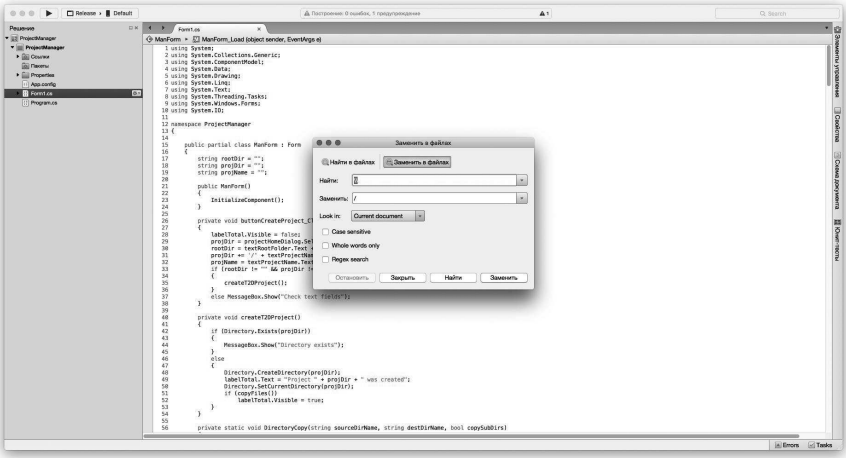

Рис. 8.9. Диалог «Заменить в файлах»

поэтому там жесткий хард-код, который придется заменять для соответствия правилам OS X.

Во-вторых, поскольку в OS X нет динамических библиотек, их копировать не придется. В самом низу файла с исходным кодом есть статический класс, в котором объявлены константы — имена файлов — динамических библиотек:

```
public static class Constants
{
//public const string rootDir = "c:/Torque/Torque2D-
development-3-2/";
public const string exeName = "Torque2D.app";
public const string leapName = "Leap.dll";
public const string openALName = "OpenAL32.dll";
public const string uniName = "unicows.dll";
public const string script_main = "main.cs";
}
```

Файл unicows.dll, начиная с версии Torque 2D 3.3 для Windows не копируется, поскольку был изъят из дистрибутива движка. Вместе с тем надо закомментировать и/или удалить код для копирования файлов Leap.dll и OpenAL32.dll. В операционной системе OS X код этих библиотек помещается в исполняемый объект.

В-третьих, везде измените Torque2D.exe на Torque2D.app, а так же *.exe на *.app.

В-четвертых, если в Windows исполняемым объектом является файл с расширением exe, то в OS X исполняемым объектом является каталог с расширение app кроме кода запуска включающий код дополнительных библиотек, картинки, иконки и др., что только позволит фантазия программиста. Однако набивать этот каталог не рекомендуется. Поэтому, если в менеджере проектов версии для Windows мы просто копировали исполняемый файл движка из корневой папки Торка в директорию с нашим новым проектом, то в OS X нам надо копировать каталог app со всем содержимым. Для этого служит статичная рекурсивная функция DirectoryCopy, принимающая 3 параметра: папка — источник, папка — приемник, булевый флаг — копировать ли поддиректории:

```
private static void DirectoryCopy(string
sourceDirName, string destDirName, bool copySubDirs)
{
DirectoryInfo dir = new DirectoryInfo(sourceDirName);
if (!dir.Exists)
{
throw new DirectoryNotFoundException("Папка не
существует: " + sourceDirName);
}
DirectoryInfo[] dirs = dir.GetDirectories();
// If the destination directory doesn't exist, create
//it.
if (!Directory.Exists(destDirName))
{
Directory.CreateDirectory(destDirName);
}
// Get the files in the directory and copy them to the
//new location.
FileInfo[] files = dir.GetFiles();
foreach (FileInfo file in files)
{
string temppath = Path.Combine(destDirName, file.Name);
file.CopyTo(temppath, false);
}
// If copying subdirectories, copy them and their
//contents to new location.
if (copySubDirs)
{
foreach (DirectoryInfo subdir in dirs)
{
string temppath = Path.Combine(destDirName, subdir.
Name);
```

```
DirectoryCopy(subdir.FullName, temppath, copySubDirs);
}
}
}
```

Внутри тела функции первым делом создаем объект директории-источника и сохраняем его в переменной dir класса DirectoryInfo. Если в результате объект dir нулевой, тогда папка-источник не существует, выводим соответствующее сообщение. Если проверка завершилась успешно, т.е. объект dir инициализирован, тогда в массив объектов DirectoryInfo — dirs помещаем все подпапки. Далее, выполняем проверку на существование папки приемника, если ее нет, то создаем ее. Затем в массив files объектов класса FileInfo сохраняем список файлов, находящихся в папке-источнике. После этого в цикле foreach копируем каждый файл из папки источника в папку-приемник. Далее, если флаг copySubDirs имеет положительное значение, тогда копируем содержимое каждой подпапки из массива dirs с помощью рекурсивных вызовов рассматриваемой функции — DirectoryCopy.

В-пятых, там, где раньше были операторы File.Exist, проверяющие существование исполняемого файла exe, теперь должны быть Directory. Exist, соответственно, проверяющие существование папки — исполняемого объекта с расширением app.

На этом изменения исчерпали себя. Проект готов к эксплуатации.

Теперь нам надо пересобрать проект для операционной системы OS X (или macOS). В главном меню Xamarin Studio Community выберите пункт «Сборка -> Пересобрать все». В итоге, если вы не сделали никаких ошибок, проект будет перекомпилирован и собран заново. Если вы сейчас откроете папку Debug или Release, смотря проект какого типа вы только что построили, то обнаружите там исполняемый файл для Windows — ProjectManager.exe.

Вместе с новой версией Xamarin Studio устанавливается Xamarin Profiler. Если раньше (со старой версией Xamarin Studio) по построенному с помощью Mono файлу дважды щелкнуть, OS X сообщит, что она не поддерживает файлы такого типа. Это было в порядке вещей, в этой операционной системе исполняемые объекты — это папки с расширением app. Xamarin Studio строит универсальные исполняемые файлы для среды Mono, с помощью которой мы можем запустить exe файлы из OS X. Для этого в терминале с помощью команды cd осуществлялся пе-

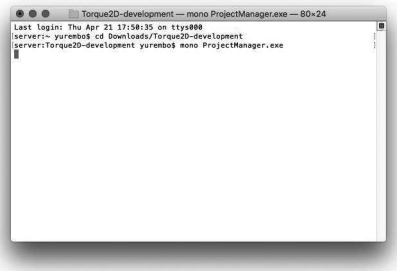

Рис. 8.10. Запуск менеджера проектов с помощью среды Mono

реход в каталог, где находился менеджер проектов и выполнялась команда (рис. 8.10):

```
mono ProjectManager.exe
```

В результате в оконном режиме запускался менеджер проектов, ровно как в Windows только по правилам OS X.

Если сейчас (с новой версией Xamarin Studio) запустить exe файл, сначала загрузится Xamarin Profiler (рис. 8.11). Он позволяет задать опции профилирования запускаемого приложения.

Выберите опцию All Instruments и нажмите кнопку Start Profiling. Запустится профайлер вместе с нашим приложением. Плюс от этого в том,

Рис. 8.11. Запуск профайлера

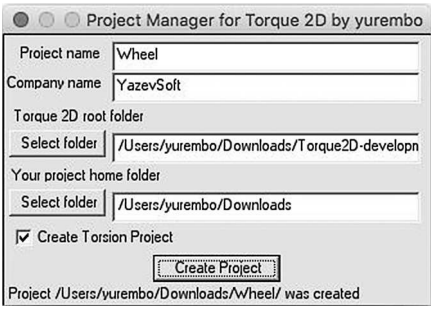

Рис. 8.12. Менеджер проектов с заполненными полями создал мини-игру

что больше не надо запускать mono-приложения с помощью терминала, достаточно дважды нажать кнопку мыши на исполняемом exe файле, и он будет работать, как в системе Windows. Правда, пока внутри профайлера. Работа с профилировщиком выходит за рамки данной книги.

Между тем вы по-прежнему можете запустить приложение с помощью командной строки, используя mono. В таком случае не придется использовать профилировщик.

Теперь давайте проведем эксперимент. Скопируйте исполняемый exe файл из каталога с проектом в папку с движком. В моем случае движок находится в директории Downloads/Torque2D-development. В терминале перейдите в вышеуказанную папку, туда, куда вы скопировали исполняемый файл менеджера проектов. И запустите его. В результате он будет работать так же, как в операционной среде Windows. О чем подробнее можно прочитать в начале текущей главы.

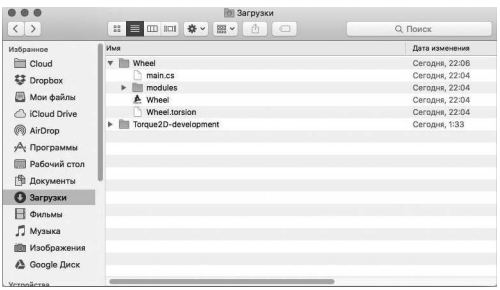

Рис. 8.13. Папка Downloads с движком и созданным проектом Wheel

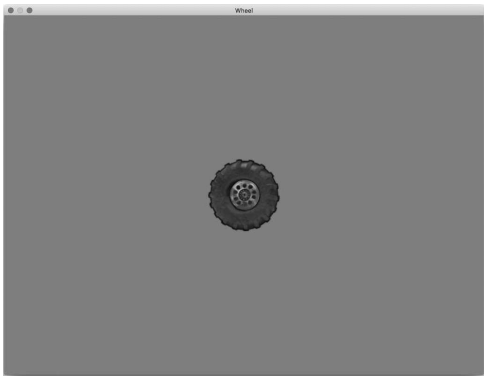

Рис. 8.14. Созданная с помощью менеджера проектов мини-игра для OS X на движке Torque 2D

Если вам не охота все это делать собственноручно, можете скачать полный репозиторий ProjectManager для OS X с моего аккаунта GitHub (*https://github.com/yurembo/ProjectManager-OS-X*).

Visual Studio for Mac

Второй способ портирования Project Manager для OS X заключается в использовании интегрированной среды разработки Visual Studio for Mac. Ее предварительная версия вышла как раз в то время, когда я работал над вторым изданием, поэтому я решил протестировать ее на реальном проекте — Project Manager.

Чтобы скачать Visual Studio for Mac перейдите на страницу: *https://www.visualstudio.com*. Выберите иконку Visual Studio for Mac Preview.

Начнется процесс скачивания.

В результате на ваш Мак будет залит стандартный установщик VisualStudioForMacPreviewInstaller.dmg. Установка запускается стандартно — двойным щелчком по файлу. Процесс инсталляции тоже вполне обычный. Вас попросят выбрать устанавливаемые компоненты (рис. 8.16).

Так как Microsoft купила компанию Xamarin, многие компоненты производства последней: Mono Framework, Xamarin.iOS, Xamarin.Android, Xamarin.Mac и другие. Обратите внимание в процессе установки Visual Studio for Mac, также устанавливается среда выполнения Mono.

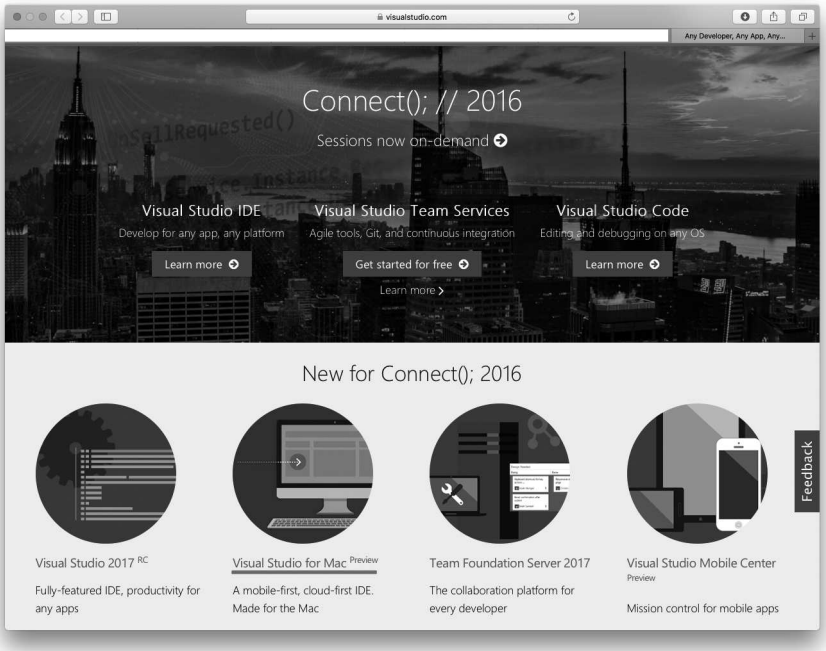

Рис. 8.15. Скачивание Visual Studio for Mac Preview

После установки появится окно с кнопкой для запуска Visual Studio for Mac. Среда имеет привычный для пользователей VS интерфейс, однако, заточенный для OS X (рис. 8.16). С помощью среды VS можно про-

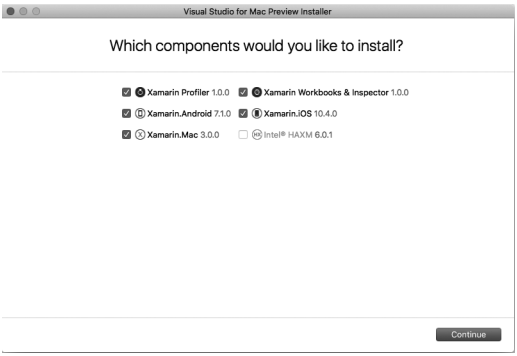

Рис. 8.16. Выбор компонентов для установки

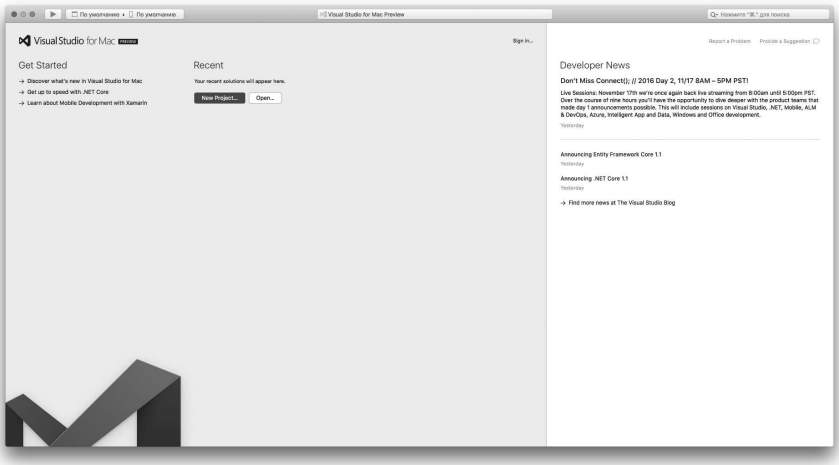

Рис. 8.17. Visual Studio for Mac Preview

вести такой же процесс редактирования, что описан выше в предыдущем разделе.

Когда редактирование будет завершено (рис. 8.18), можно откомпилировать и запустить менеджер проектов. После построения получается такой же исполняемый файл с расширением .exe.

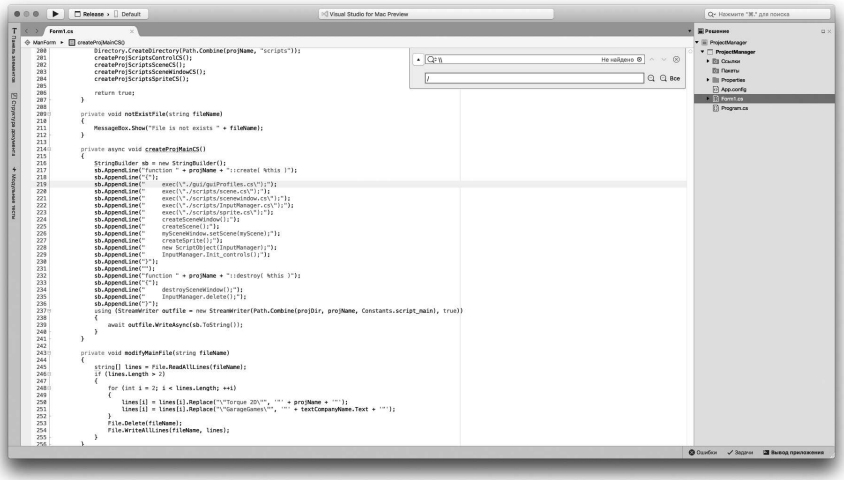

Рис. 9.18. Редактирование в Visual Studio for Mac Preview

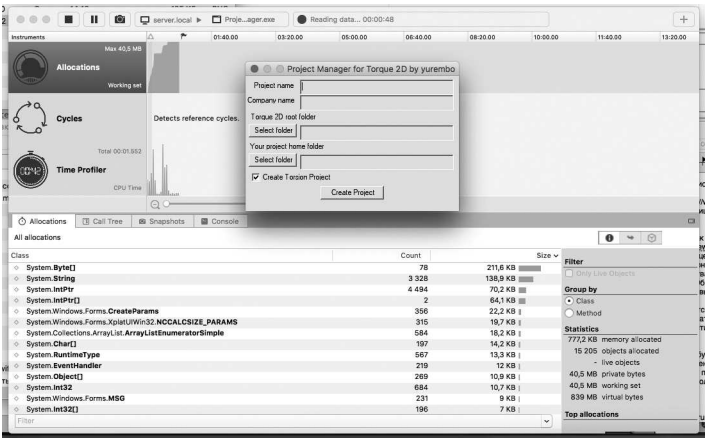

Рис. 8.19. Запущенный вместе с профайлером Project Manager

4 Заключение

На этом наш экскурс в исходный код менеджера проектов завершен. Мы узнали: каким образом менеджер проектов подготавливает заготовку новой игры. Также мы рассмотрели способы портирования Windows-приложений, созданных с использованием фреймворка .Net для операционной системы OS X (macOS).

Теперь, зная, как устроен менеджер проектов, вы можете скачать его исходный код с моей страницы GitHub (ссылка приведена в начале главы) и модифицировать его под свои цели: расширить, добавить функциональность и другое.

Уже следующую главу мы начнем с создания заготовки для игры с помощью менеджера проектов.

Глава 9. Простой физический эксперимент

Оглавление

1 Соединения

В главе 5 «Физические свойства и взаимодействия» мы рассмотрели много физических компонентов, свойств и способов взаимодействия Торковских объектов между собой.

Между тем Torque 2D унаследовал от Box 2D еще один очень мощный тип физических объектов — соединения. Они используются для соединения двух физических объектов класса SceneObject или его наследников, например, Sprite. В Torque 2D реализованы 10 типов соединений:

- DistanceJoint
- FrictionJoint
- WeldJoint
- RopeJoint
- WheelJoint
- PulleyJoint
- TargetJoint
- PrismaticJoint
- MotorJoint
- RevoluteJoint

В этом разделе мы поговорим об общих чертах всех соединений, о том, как они создаются, работают, и как ими управлять программисту, использующему их в своей игре.

Соединения управляются глобальным объектом сцены (типа Scene). То есть сцена создает соединения, уничтожает и манипулирует ими. После создания соединений они слабо поддаются манипуляциям. Но их легко пересоздать с новыми начальными параметрами. Они взаимодействуют с другими объектами под влиянием тех же свойств: плотность, трение, восстановление.

Во время создания соединения методом построения соединения возвращается его уникальный идентификатор (jointID), в дальнейшем его можно использовать для обращения к этому конкретному соединению, подобно тому, как используется идентификатор объектов. Для примера рассмотрим метод createRevoluteJoint:

```
%jointID = Scene.createRevoluteJoint(sceneobjectA,
sceneobjectB, localAnchorA, localAnchorB);
```

Возвращенный id сохранен в переменную для последующего обращения. Хотя id объекта и id соединения похожи, на самом деле, они отличаются: если id объекта — это уникальное случайное число, то id со-

единения — это уникальное число из упорядоченной последовательности от 0. В остальном они похожи.

Кроме создания соединений объект класса Scene может получить количество всех соединений, находящихся в сцене, с помощью метода getJointCount:

```
Scene.getJointCount();
```

Узнать тип определенного соединения (все типы перечислены выше) можно с помощью метода getJointType, который в качестве параметра принимает идентификатор соединения:

```
Scene.getJointType(%jointID);
```

Также можно удалить соединение, вызвав метод deleteJoint, передав ему id соединения:

```
Scene.deleteJoint(%jointID);
```

Еще раз взглянем на приведенный выше метод:

```
createRevoluteJoint(sceneobjectA, sceneobjectB,
localAnchorA, localAnchorB);
```

1-й его параметр — это 1-й объект для соединения, 2-й параметр, соответственно, 2-й объект для создания соединения. 3-й и 4-й параметры представлены парами значений — координат по 2-м осям: X и Y. Эти координаты в локальной системе координат определенного объекта указывают положение якорной точки для этого объекта.

Какая бы скорость не была применена к привязанному объекту, теперь он будет реагировать и действовать по другому, его будет ограничивать связь. Например, если к независимому (не привязанному) объекту применить ускорение — Linear velocity, он будет двигаться в соответствии с указанными направлением и силой движения. Или, если к нему применить угловое ускорение, тогда он будет вращаться вокруг своей оси. С другой стороны, если эти силы применить к объекту, привязанному соединением RevoluteJoint, тогда он будет ограничен движением и вращением вокруг парного объекта привязки.

2 Типы соединений

В этом разделе мы поговорим: какие типы соединений бывают, чем они отличаются, для чего служат и какой достигается эффект от их использования.

DistanceJoint

Если это соединение создано между двумя объектами, то между ними будет удерживаться минимальное расстояние. Если для свойства frequency (частота) установить значение отличное от 0, тогда соединение будет сжиматься и расширяться при попытке удержания определенной дистанции.

DistanceJoint создается с помощью метода createDistanceJoint объекта класса Scene. Кроме первых стандартных 6-ти параметров, обсужденных нами в прошлом разделе, метод получает:

— distance — расстояние по умолчанию между двумя объектами;

— frequency — «пружинная» частота измеряется в Герцах (Hz):

— значение выше нуля позволяет соединению сокращаться и резко выпрямляться — эффект пружины;

— значение равное 0 означает жесткое взаимодействие;

— значение равное 1 означает эффект, когда оба объекта связаны эластичной лентой;

— DampingRatio — количественное затухание, примененное к мягкости пружины; значения распространяются от 0 до 1, где 0 означает без ослабления, 1 означает абсолютную жесткость;

— CollideConected — булево значение, если имеет положительное значение, это позволяет обеим якорным точкам сталкиваться друг с другом;

Дополнительные функции позволяют установить и вернут значения свойств соединения независимо от момента создания:

— Scene.setDistanceJointDampingRatio(%jointID, DampingRatio); — устанавливает количественное затухание пружины: 1-й параметр: идентификатор соединения; 2-й — значение затухания;

— Scene.setDistanceJointFrequency(%jointID, Frequency); — изменяет частоту, значение которой передается во 2-м параметре;

— Scene.setDistanceJointLength(%jointID, Length); — изменяет дистанцию между двумя соединенными объектами;

— Scene.getDistanceJointDampingRatio(%jointID); — возвращает значение затухания пружины;

— Scene.getDistanceJointFrequency(%jointID); — возвращает значение частоты;

— Scene.getDistanceJointLength(%jointID); — возвращает расстояние между объектами;

FrictionJoint

«Соединение трения» ограничивает движения. Если один из объектов генерирует больше силы, чем значение свойства MaximumForce, 2-ой объект не будет следовать за 1-м, а будет находиться позади него, вместо того, чтобы быть притянутым. В противоположность этому, если 1-й объект движется медленно, 2-й объект будет тянуться в соответствии с движением.

Параметр MaximumTorque следует той же самой логике, при этом, используя силы вращения.

FrictionJoint создается с помощью функции createFrictionJoint:

```
Scene.createFrictionJoint(...)
```

Кроме 6-ти общих параметров для всех соединений при создании данного, создающая его функция получает 3 дополнительных:

— MaximumForce — максимальная сила в Ньютонах. Когда движение одного из объектов превосходит максимальную силу (MaximumForce) для 2-го объекта, движение перестает на него влиять.

— MaximumTorque измеряется в Ньютон-метр. Когда вращательное движение одного из объектов превосходит максимальную вращательную силу (MaximumTorque) для 2-го объекта, вращательное движение перестает на него влиять.

— CollideConected — булево значение, если имеет положительное значение, то обе якорные точки будут сталкиваться друг с другом;

Дополнительные функции позволяют установить и вернут значения свойств соединения независимо от момента создания:

— Scene.setFrictionJointMaxForce(jointID, MaxForce); — устанавливает максимальную силу (MaxForce), выраженную в Ньютонах;

— Scene.setFrictionJointMaxTorque(jointID, MaxTorque); — устанавливает максимальную вращательную силу (MaxTorque), выраженную в Ньютон-метр;

— Scene.getFrictionJointMaxForce(jointID); — возвращает максимальную силу;

— Scene.getFrictionJointMaxTorque(jointID); — возвращает максимальную силу вращения для определенного соединения;

WeldJoint

«Сварной шов» ограничивает все движение между объектами. Функция для создания данного типа соединения выглядит следующим образом:

```
Scene.createWeldJoint(...)
```

Кроме 6-ти вышеупомянутых параметров функция принимает следующие 3:

— frequency — «пружинная» частота измеряется в Герцах (Hz):

— значение выше нуля позволяет соединению сокращаться и резко выпрямляться — эффект пружины;

— значение равное 0 означает жесткое взаимодействие;

— значение равное 1 означает эффект, когда оба объекта связаны эластичной лентой;

— DampingRatio — количественное затухание, примененное к мягкости пружины; значения распространяются от 0 до 1, где 0 означает без ослабления, 1 означает абсолютную жесткость;

— CollideConected — булево значение, если имеет положительное значение, это позволяет обеим якорным точкам сталкиваться друг с другом;

Дополнительные методы позволяют установить и узнать значения приведенных выше свойств.

RopeJoint

«Веревочное соединение» действует так, как будто оба объекта связаны сегментной веревкой. Создание данного типа соединения осуществляется с помощью функции:

```
Scene.createRopeJoint(...)
```

Функция имеет 2 дополнительных параметра:

— MaxLength — максимальная длина веревки;

— CollideConected — булево значение, если имеет положительное значение, это позволяет обеим якорным точкам сталкиваться друг с другом;

Дополнительных функций только две:

— для установки длины веревки:

```
Scene.setRopeJointMaxLength(jointID, MaxLength);
```

— для получения длины веревки:

```
Scene.getRopeJointMaxLength(jointID);
```

WheelJoint

Соединение в виде колеса заключает точку объекта sceneObjectB в линии на объекте sceneObjectA. Этот тип соединения так же позволяет сделать натянутую пружину.

Создание WheelJoint осуществляется с помощью функции:

```
Scene.createWheelJoint(...)
```

Вдобавок к 6-ти стандартным параметрам, функция принимает еще 2:

— WorldAxis — мировые оси координат в большинстве случае имеют значение «0 1»;

— CollideConected — булево значение, если имеет положительное значение, это позволяет обеим якорным точкам сталкиваться друг с другом;

Среди дополнительных функций есть включающая двигатель соединения:

```
Scene.setWheelJointMotor(jointID, enableMotor,
MotorSpeed, MotorMaxTorque);
```

Двигатель использует значение MotorSpeed, как константную силу. Вращательный момент мотора ограничен значением MotorMaxTorque и никогда не превышает его. Данная функция принимает такие параметры:

— jointID — идентификатор соединения;

— enableMotor — булево значение: включить или выключить мотор;

— MotorSpeed — сила, которая будет применена, измеряется в градусах в секунду;

— MotorMaxTorque — используется для ограничения верхнего предела скорости соединения мотора;

Функция Scene.setWheelJointFrequency(jointID, frequency); устанавливает значение частоты;

Функция Scene.setWheelJointDampingRatio(jointID, dampingRatio); устанавливает значение затухания;

Функция Scene.getWheelJointMotor(); возвращаем сразу 3 значения, разделенные пробелом: включен или выключен мотор, скорость мотора, максимальная вращательная скорость мотора;

Функция Scene.getWheelJointFrequency(jointID); возвращает частоту;

Функция Scene.getWheelJointDampingRatio(jointID); возвращает фактор затухания.

PulleyJoint

Соединение PulleyJoint является одним из самых интересных типов соединений в Torque 2D. Оно представляет собой целую законченную физическую конструкцию, иными словами, шкив. Для ясности картины устройства предлагаю взглянуть на следующую иллюстрацию (рис. 9.1).

Объект sceneObjectA привязан к объекту worldGroundAnchorA, а объект sceneObjectB — к объекту worldGrountAnchorB. Оба объекта worldGroundAnchor представляют собой пустые объекты, которые не реагируют на внешние силы. Единственное, что имеет значение, это их расположение. Если прибавить lengthA и lengthB, то мы получим полную длину веревки. Расстояние между точками worldGroundAnchor (верхний отрезок) не влияет на баланс.

По умолчанию, когда один из объектов sceneObject находится на одну единицу дистанции ниже, противоположный sceneObject тянется вверх на одну единицу измерения.

Система выходит из равновесия — становится более сложной, когда изменяются стандартные коэффициенты между объектами, находящиеся на обеих сторонах шкива.

Коэффициент определяет изменение частоты lengthA относительно lengthB во время влияния ограничивающей силы. Это означает, что с ко-

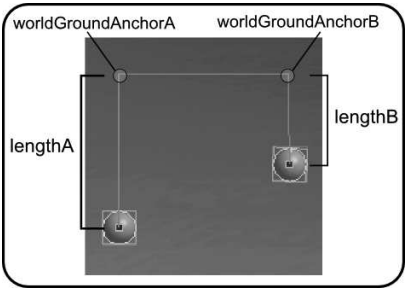

Рис. 9.1. Шкив

эффициентом равным 2, в каждый момент времени sceneObjectA будет тянуться вниз на 1 условную единицу, в то же время sceneObjectB будет тянуться вверх на 2 единицы. Таким образом, ограничивающая сила lengthA будет равна половине силы lengthB. Другими словами, когда lengthA полностью вытянута, она будет в 2 раза длиннее, чем полностью вытянутая lengthB.

Для создания системы шкива используется функция createPulleyJoint, вызов которой осуществляется следующим образом:

```
Scene.createPulleyJoint(...)
```

Кроме 6-ти стандартных параметров функция принимает:

— worldGroundAnchorA — мировая позиция по осям X и Y для первой якорной точки;

— worldGroundAnchorB — мировая позиция по осям X и Y для второй якорной точки;

— ratio — определяет расширение разницы коэффициента между lengthA и lengthB. По умолчанию равен 1;

— lengthA — начальное расстояние между sceneObjectA и worldGroundAnchorA. В качестве значения по умолчанию берется физическая дистанция (в игровых единицах) между sceneObjectA и worldGroundAnchorA;

— lengthB — начальное расстояние между sceneObjectB и worldGroundAnchorB. В качестве значения по умолчанию берется физическая дистанция (в игровых единицах) между sceneObjectB и worldGroundAnchorB;

— CollideConected — булево значение, если имеет положительное значение, это позволяет обеим якорным точкам сталкиваться друг с другом;

Соединение PulleyJoint не имеет дополнительных функций.

TargetJoint

«Целевое соединение» управляет только одним объектом, применяя к нему силы для удержания в позиции, заданной значением worldTarget. Для создания соединения данного типа используется функция createTargetJoint:

```
Scene.createTargetJoint(%sceneobjectA, worldTarget,
MaxForce, UseCenterofMass, frequency, Dampingratio,
CollideConnected);
```

Как видно по приведенному прототипу: функция имеет отличающийся набор параметров:

— sceneObjectA — объект, для которого строиться целевое соединение;

— worldTarget — определяется двумя координатами: X и Y, обозначает позицию, куда будет двигаться и которой будет держаться объект sceneObjectA;

— MaxForce — максимальная скорость для движения sceneObjectA к worldTaget;

— frequency — «пружинная» частота измеряется в Герцах (Hz):

— значение выше нуля позволяет соединению сокращаться и резко выпрямляться — эффект пружины;

— значение равное 0 означает жесткое взаимодействие;

— значение равное 1 означает эффект, когда оба объекта связаны эластичной лентой;

— DampingRatio — количественное затухание, примененное к мягкости пружины; значения распространяются от 0 до 1, где 0 означает без ослабления, 1 означает абсолютную жесткость;

— CollideConected — булево значение, если имеет положительное значение, это позволяет обеим якорным точкам сталкиваться друг с другом;

Дополнительные функции:

— Scene.setTargetJointTarget(jointID, worldTargetX/Y); — устанавливает целевую позицию, соответственно, по координатам X и Y;

— Scene.setTargetJointFrequency(jointID, frequency); — устанавливает частоту;

— Scene.setTargetJointDampingRatio(jointID, DampingRatio); — устанавливает затухание;

— Scene.getTargetJointTarget(jointID); — возвращает координаты целевой позиции;

— Scene.getTargetJointFrequency(jointID); — возвращает «пружинную» частоту, измеряемую в Герцах;

— Scene.getTargetJointDampingRatio(jointID); — возвращает фактор затухания для данного конкретного соединения, измеряется от 0 до 1;

PrismaticJoint

Призматическое соединение позволяет двум объектам перемещаться относительно друг друга и определенных осей координат. Лучше все-

го это заметно, когда один из объектов имеет статический тип, что полностью раскрывает возможности данного соединения.

Создание соединения осуществляется функцией createPrismaticJoint:

```
Scene.createPrismaticJoint(%sceneobjectA,
%sceneobjectB, localanchorA, localanchorB, worldAxis,
CollideConnected);
```

Первые 4 параметра уже много раз обсуждались в течение данной главы.

5-й параметр: worldAxis — описывает трансляционный свободный угол в мировых координатах;

6-й параметр: CollideConnected — неоднократно описывался, здесь он играет ту же роль.

Дополнительные функции:

— Scene.setPrismaticJointLimit(jointID, enableLimit, LowerTranslation, UpperTranslation);

— enableLimit — включает или выключает режим, при котором sceneObjectB, достигнув пределы lowerTranslation или upperTranslation, перестает движение:

— lowerTranslation — минимальная дистанция между sceneObjectA и sceneObjectB в мировых координатах;

— upperTranslation — наибольшая дистанция между sceneObjectA и sceneObjectB в мировых координатах;

— Scene.setPrismaticJointMotor(enableMotor, MotorSpeed, MotorMaxTorque); — активизирует мотор соединения, применяя значение MotorSpeed, как константную силу пока не будет достигнуто значение MotorMaxTorque;

— enableMotor — булево значение: включает или выключает мотор;

— MotorSpeed — сила, которая будет применена в моторе (измеряется в градусах в секунду);

— MotorMaxTorque — используется как верхний предел скорости мотора;

— Scene.getPrismaticJointLimit(jointID); — возвращает значения: enableLimit, lowerTranslation, upperTranslation, разделенные пробелами;

MotorJoint

Motor joint размещает оба объекта относительно LinearOffset и AngularOffset. Передвигая один из объектов с определенной силой, та-

кая же сила будет воздействовать на второй из них. Это позволяет создавать сложные механические взаимодействия.

Для создания Motor Joint используется функция createMotorJoint:

```
Scene.createMotorJoint(%sceneobjectA, %sceneobjectB,
LinearOffset, angularOffset, maxForce, maxTorque,
correctionFactor, CollideConnected);
```

Первые 2 параметра — ссылки на объекты, участвующие в соединении;

— LinearOffset — линейное смещение sceneObjectB в локальных координатах sceneObjectA;

— AngularOffset — угловое смещение между двумя объектами;

— MaximumForce — максимальная сила в Ньютонах. Когда движение одного из объектов превышает максимальную силу второго объекта, на него перестает влиять движение;

— MaximumTorque — максимальная вращательная скорость в Ньютон-метр. Когда движение одного из объектов превышает вращательную скорость второго объекта, на второй объект перестает влиять движение;

— correctionFactor — по-другому называется, как погрешность, определяет с какой неопределенностью происходят вычисления;

— CollideConected — булево значение, если имеет положительное значение, это позволяет обеим якорным точкам сталкиваться друг с другом;

Дополнительные функции в данной категории соединений позволяют получить и установить значения обсуждаемых выше свойств:

— Scene.setMotorJointLinearOffset(jointID, LinearOffset); — устанавливает линейное смещение sceneObjectB в локальных координатах sceneObjectA;

— Scene.setMotorJointAngularOffset(jointID, AngularOffset); — устанавливает угловое смещение между объектами;

— Scene.setMotorJointMaxForce(jointID, MaxForce); — устанавливает максимальную силу;

— Scene.setMotorJointMaxForce(jointID, MaxForce); — устанавливает максимальную вращательную силу;

— Scene.getMotorJointLinearOffset(jointID); — возвращает линейное смещение между объектами по X и Y;

— Scene.getMotorJointAngularOffset(jointID); — возвращает угловое смещение между объектами в градусах;

— Scene.getMotorJointMaxForce(jointID); — возвращает максимальную силу в Ньютонах;

— Scene.getMotorJointMaxTorque(jointID); — возвращает максимальную вращательную силу в Ньютон-метр.

RevoluteJoint

Соединение Revolute организует вращение обоих объектов вокруг общей якорной точки. Для лучшей наглядности поведения этого типа соединения рекомендуется сделать один из объектов статичным.

Для создания Revolute-соединения используется функция createRevoluteJoint:

```
Scene.createRevoluteJoint(%sceneobjectA,
%sceneobjectB, localanchorA, localanchorB,
CollideConnected);
```

Все параметры этой функции уже рассмотрены.

Дополнительные функции:

— Scene.setRevoluteJointLimit(jointID, enableLimit, lowerAngle, upperAngle); — включает лимит, когда угол наклона соединения достигает lowerAngle или upperAngle, движущийся объект прекращает движение;

— lowerAngle — минимальный угол наклона между sceneObjectA и sceneObjectB, выраженный в градусах;

— upperAngle — максимальный угол наклона между sceneObjectA и sceneObjectB, выраженный в градусах;

— Scene.setRevoluteJointMotor()(jointID, enableMotor, MotorSpeed, MotorMaxTorque); — активизирует мотор для соединения, применяя значение MotorSpeed в качестве константной скорости, при этом вращательная скорость мотора ограничена значение MotorMaxTorque;

— enableMotor — включает или отключает мотор;

— MotorSpeed — сила, применяемая к мотору, выражает в градусах в секунду;

— MotorMaxTorque — предел скорости мотора;

— Scene.getRevoluteJointLimit(jointID); — возвращает значения: enableLimit, lowerAngle, upperAngle, разделенные пробелами;

— Scene.getRevoluteJointAngle(jointID); — возвращает текущий угол соединения, измеряемый в градусах;

— Scene.getRevoluteJointSpeed(jointID); — возвращает текущее угловое ускорение;

Вывод

Некоторые физические механизмы движка мы не можем рассмотреть в рамках разрабатываемых нами игр, потому что эти возможности действительно очень широки и охватывают несколько игровых жанров! Для каждого свои возможности. Поэтому я решил разработать игру, представляющую собой физический эксперимент, чтобы показать и иметь возможность обсудить физические механизмы. Поехали! Torque 2D нам, как всегда, поможет!

3 Идея

Нашей целью будет создать симулятор вращающегося на веревке камня. Звучит скучновато. Но интерес здесь представляет физика. Построение реально выглядящей и действующей веревки представляет собой довольно сложную задачу. Torque 2D нам поможет! Кроме того, у нас должен быть способ раскручивания веревки. Мы не будем ограничиваться: раскручивание может быть осуществлено, как по часовой, так и против часовой стрелки. Нужно устройство позволяющее раскручивать веревку, на конце которой закреплен камень. Пусть будет своего рода диск, находящийся в основе механизма. В центре диска установлен держатель для веревки. А на конце веревки закреплен камень. Кроме того, для интереса повернем нашу конструкцию на 90 градусов по оси X, тем самым мы поместим ее в наклонном положении, в таком случае на камень будет действовать гравитация. Это добавит динамику в нашу демонстрацию.

Сделаем важное упрощение, от которого наша демонстрация будет выглядеть в разы эффектнее: заменим веревку цепью.

В итоге для реализации идеи нам нужны 5 визуальных объектов:

1) фон;
2) диск — раскручиваемая подставка;
3) крепеж — держатель;
4) веревка — цепь;
5) камень;

Нарисуем их в любом графическом редакторе — по выбору. Также вы можете взять необходимый для данной игры арт в материалах к книге. Далее перейдем к кодингу на Torque Script.

4 Разработка физического симулятора

На начальном этапе, чтобы не кодить с нуля, а создать заготовку для нашей демонстрации, воспользуемся менеджером проектов, описанном в прошлой главе. Создадим проект RopeStone, как показано на следующем рисунке — рис. 9.2.

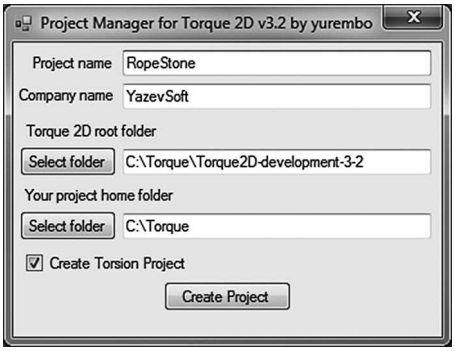

Рис. 9.2. Создание проекта RopeStone

Все настройки по умолчанию подходят нашему приложению и нас устраивают. Файл sprite.cs можно сразу удалить, мы создадим другой. В таком случае пока не будет добавлен новый файл, работоспособность приложения будет нарушена.

Фон

Запустите Torsion, открыв проект RopeStone.torsion. Внутри среды разработки переместитесь в папку \RopeStone\modules\RopeStone\scripts. Первым делом создадим фон — объект background. Для этого в папке scripts создайте новый скрипт, куда поместите следующую функцию:

```
function createBackground()
{
 %sprite = new Sprite();
 %sprite.setBodyType( static );
 %sprite.Position = "0 0";
 %sprite.Size = $screenWidth SPC $screenHeight;
 %sprite.SceneLayer = 31;
 %sprite.Image = "RopeStone:snow";
```

```
 myScene.add( %sprite );
}
```

Этой функции будет достаточно. Здесь создается спрайт, физический тип — статический (поскольку фон не должен взаимодействовать с другими объектами), размер: 100 × 75. слой — 31, изображение из ассета RopeStone:snow. По замыслу ничего больше для фона не надо. Taml-код для загрузки изображения снега выглядит следующим образом:

```
<ImageAsset
   AssetName="snow"
   ImageFile="snow.png" />
```

В нем тоже нет ничего сверхъестественного.

Диск

На фоне первым объектом нашей конструкции располагается диск для вращения. Он создается в функции createDisc:

```
function createDisc()
{
   %sprite = new Sprite(disk);
   %sprite.class = "Disc";
   %sprite.setBodyType( static );
   %sprite.Position = "0 0";
   %sprite.Size = "40 40";
   %sprite.SceneLayer = 31;
   %sprite.Image = "RopeStone:disc";
   %sprite.setUseInputEvents(true);
   myScene.add( %sprite );
}
```

Поскольку у этого объекта будут дополнительные, присущие только ему методы, создадим для него отдельный класс — «Disc»:

```
%sprite.class = "Disc";
```

При этом нам предстоит физически взаимодействовать с ним, то есть он должен реагировать на ввод пользователя (с помощью мыши или сенсорного экрана), в таком случае необходимо установить для него соответствующее свойство:

```
%sprite.setUseInputEvents(true);
```

Taml-код для загрузки изображения выглядит, как предыдущий:

```
<ImageAsset
   AssetName="disc"
   ImageFile="disc.png" />
```

Для реализации управления нам нужны 2 обработчика событий: нажатия и перетаскивания:

```
function Disc::onTouchDown(%this, %touchID,
%worldPosition)
{
   $lastX = %worldPosition.x;
   $lastY = %worldPosition.y;
}
function Disc::onTouchDragged(%this, %touchID,
%worldPosition)
{
   $X = %worldPosition.x;
   $Y = %worldPosition.y;
   %x = ($lastX - $X) * 5;
   %y = ($lastY - $Y) * 5;
   chain.setLinearVelocity(-%x, -%y);
}
```

Когда пользователь нажимает на диске курсором мыши или пальцем (на сенсорном экране) вызывается обработчик onTouchDown. Он принимает: указатель на текущий объект, в нашем случае диск только один, номер пальца, координаты места прикосновения — курсора или пальца: дуплет десятичных дробей, разделенных пробелом. Внутри обработчика мы сохраняем позицию касания/нажатия в двух глобальных переменных, соответственно, x в $lastX и y в $lastY.

Когда пользователь перемещает курсор с нажатой кнопкой или не отпуская палец в пределах диска, вызывается обработчик события onTouchDragged, он получает те же параметры, что предыдущий.

В теле метода мы сохраняем текущую позицию указателя в двух глобальных переменных; затем, выделив еще 2 переменные, сохраняем в них разность между старой позицией указателя и новой, умноженной на 5:

```
   %x = ($lastX - $X) * 5;
   %y = ($lastY - $Y) * 5;
```

Последним действием метода указываем полученные разности координат в качестве параметров для линейного ускорения по соответствующим координатным осям, предварительно инвертировав их:

```
chain.setLinearVelocity(-%x, -%y);
```

Объект chain — это начальный элемент цепи. Вскоре мы рассмотрим процесс его создания.

Крепеж

Следующим объектом нашей конструкции идет крепеж. Код его создания находится в файле pole.cs в функции createPole:

```
function createPole()
{
 %size = 9;
  %sprite = new Sprite(pole);
  %sprite.setBodyType( static );
  %sprite.Position = "0 0";
  %sprite.Size = %size SPC %size;
  %sprite.SceneLayer = 30;
  %sprite.Image = "RopeStone:pole";
  myScene.add( %sprite );
  return %sprite;
}
```

Никакую дополнительную функциональную нагрузку этот объект не несет, поэтому сделан статическим. Тем более в данном случае это важно, поскольку к крепежу крепится цепь. Не будь он статическим, физическое влияние цепи оказало бы на него не нужное воздействие.

Для загрузки изображения служит следующий элементарный код:

```
<ImageAsset
  AssetName="pole"
  ImageFile="pole.png" />
```

Камень

Визуальную роль камня играет изображение астероида из игры Asteroids. Для загрузки кадра изображения мы используем следующий код:

```
<ImageAsset
  AssetName="Asteroids"
  ImageFile="asteroids.png"
  CellCountX="2"
  CellCountY="2"
  CellWidth="128"
  CellHeight="128" />
```

Всего в изображении 4 кадра.

Чтобы камень имел возможность взаимодействовать с цепью, он должен иметь динамический тип. Для камня создадим отдельный скрипт stone.cs, он будет содержать только одну функцию — для создания камня — createStone:

```
function createStone()
{
   %sprite = new Sprite(stone);
   %sprite.setBodyType( dynamic );
   %sprite.Position = "0 -30";
   %sprite.Size = "12 12";
   %sprite.SceneLayer = 31;
   %sprite.Image = "RopeStone:asteroids";
   %sprite.Frame = 2;
   %sprite.setDefaultFriction( 0.3 );
   %sprite.setDefaultDensity( 30 );
   %sprite.setDefaultRestitution(1);
   %sprite.createCircleCollisionShape( 10 );
   myScene.add( %sprite );

   return %sprite;
}
```

В этом листинге стоит отметить: задание для создаваемого спрайта динамического типа, указание на загружаемый ассет, из которого происходит выбор 3-го кадра. Дополнительно указываются физические свойства: сила трения — устанавливается с помощью метода setDefaultFriction, плотность — устанавливается с помощью метода setDefaultDensity, восстановление состояние — устанавливается с помощью метода setDefaultRestitution. Эти свойства прямым образом влияют на движение и перемещение тел. С помощью метода createCircleCollisionShape создается оболочка, реагирующая на столкновение в форме круга. В данном случае, с радиусом 10 единиц.

Несмотря на то, что камень содержит достаточно сложную нагрузку — физическое взаимодействие, дополнительных методов нам писать не нужно, Torque 2D все заботы по обработке физики берет на себя.

Цепь

Перейдём к самому интересному, и, пожалуй, главному элементу нашей физической конструкции — цепи. Код для нее находится в файле chain.cs. Весь код заключен в одной функции — создании цепи:

```
$ChainLinks = 3;
```

```
function createChain(%posX, %posY)
{
   %linkWidth = 3;
   %linkHeight = %linkWidth * 3;
   %halfLinkHeight = %linkHeight * 0.5;
   %weightSize = 18;
   %weightHalfSize = %weightSize * 0.5;
   %pivotDistance = %linkHeight * $ChainLinks;
   %fixedObject = createPole();
   %lastLinkObj = %fixedObject;
   for ( %n = 1; %n <= $ChainLinks; %n++ )
   {
      if (%n == 1)
      %obj = new Sprite(chain);
      else
      %obj = new Sprite();
      %obj.setImage( "RopeStone:chain" );
      %obj.setPosition( %posX + (%n*%linkHeight), %posY
);
      %obj.setSize( %linkWidth, %linkHeight );
      %obj.setDefaultDensity( 30 );
      %obj.setDefaultFriction( 0.3 );
      %obj.createPolygonBoxCollisionShape( %linkWidth,
%linkHeight );
      %obj.SceneLayer = 31;
      myScene.add( %obj );
      if (%n == 1)
           myScene.createRevoluteJoint( %lastLinkObj,
%obj, 0, 0, 0, %halfLinkHeight, false );
              else
           myScene.createRevoluteJoint( %lastLinkObj,
%obj, 0, -%halfLinkHeight, 0, %halfLinkHeight, false
);

      %lastLinkObj = %obj;
   }
   %weight = createStone();
   myScene.createRevoluteJoint( %lastLinkObj, %weight,
0, -%halfLinkHeight, 0, %halfLinkHeight, false );
}
```

Цепь строится из отдельных частей. Их количество указано в глобальной переменной $ChainLinks, которая определена в начале файла. Чтобы цепь не была слишком длинная, трех ступеней, кажется, достаточно.

Функция принимает 2 параметра: %posX, %posY — координаты начала цепи по обеим осям.

В начале функции объявляются и определяются переменные, служащие для дальнейших расчетов:

— %linkWidth — ширина элемента цепи;

— %linkHeight — длина (высота) элемента цепи;

— %halfLinkHeight — половина длины;

— %weightSize — размер груза — в данном случае камня, висящего на конце цепи;

— %weightHalfSize — половина размера груза;

— %pivotDistance — общая длина цепи;

Следующим действием мы выполняем создание крепежа (диск создается в другом месте), вызывая функцию createPole и сохраняя возвращенную ею ссылку на крепеж в переменной %fixedObject:

```
%fixedObject = createPole();
```

Для дальнейших манипуляций скопируем ссылку в переменную %lastLinkObj.

Далее запускаем цикл for от 1 до количества элементов цепи (значение $ChainLinks) включительно. На каждой итерации цикла происходит создание новой части цепи и установка параметров для нее. Создание частей происходит не одинаково. Есть 2 нюанса. Первый нюанс заключается в том, что, если создается элемент под номером 1 (первый элемент), тогда ему присваивается имя — chain, посредством которого происходит управление цепью с помощью вращения диска (см. раздел создания диска (файл disc.cs)), в ином случае, когда номер выше единицы, тогда создается безымянный спрайт:

```
if (%n == 1)
%obj = new Sprite(chain);
   else
%obj = new Sprite();
```

Всем частям цепи присваивается одна и та же картинка: %obj. setImage(«RopeStone:chain»);, которая берется из ассета RopeStone:chain. Код для загрузки изображения выглядит следующим образом:

```
<ImageAsset
   AssetName="chain"
   ImageFile="chain.png" />
```

Позиция устанавливается так: для каждого элемента цепи координата по оси Y одинаковая — параметр %posY, а координата по оси X вычисляется: %posX + (произведение: номер элемента на его высоту):

```
%obj.setPosition( %posX + (%n*%linkHeight), %posY );
```

Размер каждого элемента одинаковый — это ширина элемента по оси X (значение переменной %linkWidth) и высота (длина) элемента по оси Y (значение переменной %linkHeight):

```
%obj.setSize( %linkWidth, %linkHeight );
```

Значения плотности и степени трения устанавливаются соответствующими методами для всех элементов цепи одинаково:

```
%obj.setDefaultDensity( 30 );
%obj.setDefaultFriction( 0.3 );
```

Кроме того, для каждой части создается коллизия в виде полигона, поскольку элемент цепи имеет прямоугольную форму, с размерами: %linkWidth — по ширине и %linkHeight — по высоте (длине):

```
%obj.createPolygonBoxCollisionShape(%linkWidth,
%linkHeight);
```

Затем помещаем каждую часть на 31-й слой. Как вы заметили: мы все созданные объекты поместили на 31-й (последний) слой. Как Torque 2D определит последовательность вывода объектов? Все очень просто: последовательность определяется порядком создания объектов и их заталкиванием в стек сцены.

Каждый элемент цепи с одного или обоих концов связан с соседним элементом. Как рассказывалось в начале главы: в Torque 2D реализованы 10 типов соединений, унаследованных от физического движка Box 2D. В случае цепи идеально подходит связь Revolute. Она позволяет связать 2 объекта типа SceneObject (или его потомков) так, чтобы они вращались вокруг общей якорной точки. Для создания связи Revolute используется метод createRevoluteJoint объекта класса Scene.

Второй нюанс заключается в том, что создание первого соединения отличается от создания остальных:

```
if (%n == 1)
 myScene.createRevoluteJoint( %lastLinkObj, %obj, 0,
0, 0, %halfLinkHeight, false );
```

```
else
myScene.createRevoluteJoint( %lastLinkObj, %obj, 0,
-%halfLinkHeight, 0, %halfLinkHeight, false );
```

Напомню, метод createRevoluteJoint принимает такие параметры:

1) 1-й объект, участвующий в связывании;

2) 2-й объект, участвующий в связывании;

3) координата по оси X в локальных координатах 1-го объекта места расположения 1-й якорной точки;

4) координата по оси Y в локальных координатах 1-го объекта места расположения 1-й якорной точки;

5) координата по оси X в локальных координатах 2-го объекта места расположения 2-й якорной точки;

6) координата по оси Y в локальных координатах 2-го объекта места расположения 2-й якорной точки;

7) значение булевого параметра определяет: будут ли якорные точки контактировать (сталкиваться) друг с другом;

Отличия заключаются в передаче разных параметров. Таким образом, отличия не очень существенные: в разных случаях указаны разные координаты якорной точки, потому что для первой итерации цикла (создание первой связи) начальным объектом, к которому присоединяется следующий элемент является крепеж, якорная точка которого находится по середине (в нулевых локальных координатах), однако для элементов цепи (которые создаются во всех последующих итерациях) якорная точка находится на конце.

В конце тела цикла мы присваиваем созданный походу текущей итерации графический элемент цепи — %obj объекту, который будет в следующей итерации ссылаться на прошлый созданный элемент:

```
%lastLinkObj = %obj;
```

Таким образом, происходит создание всех элементов цепи. После цикла создается камень, ссылка на который сохраняется в переменной:

```
%weight = createStone();
```

Затем мы создаем последнюю связь — оконечный элемент цепи с камнем:

```
myScene.createRevoluteJoint( %lastLinkObj, %weight, 0,
-%halfLinkHeight, 0, %halfLinkHeight, false );
```

В результате получилась корректно отвечающая законам физики цепь, конечные элементы которой прикреплены, соответственно, к крепежу и камню. Так как крепеж — статический объект, следовательно, он не двигается, то он становится опорной точкой; камень, в свою очередь, динамический объект, поэтому свободно следует за цепью, которая движется посредством вращения начального элемента вокруг собственной оси или крепежа.

Гравитация

В файле scene.cs в функции createScene надо включить гравитацию. Всемирное тяготение действует во всей игровой сцене, поэтому устанавливается для объекта сцены — myScene:

```
myScene.setGravity(0,-5);
```

Метод setGravity принимает 2 параметра: притяжение по оси X и притяжение по оси Y. Таким образом, отрицательное значение для оси Y устанавливает притяжение вниз — к нижней границе экрана и далее.

Остальной код рассматривать не будем, так как он подготовлен менеджером проектов.

5 Заключение

В результате проделанной работы у нас получилась вполне достойная физическая симуляция, работа которой происходит в соответствии с законами физики (рис. 9.3).

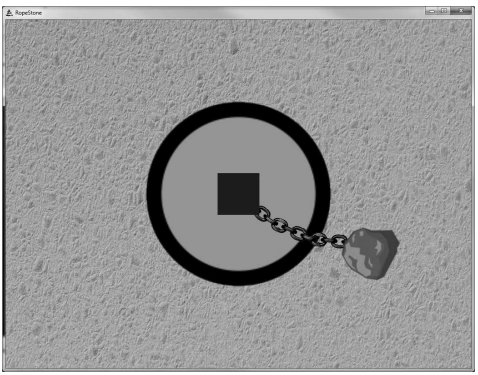

Рис. 9.3. Физическая симуляция

Пользователь может перемещать палец над диском, вращая его, тем самым цепь с камнем будут вращаться вслед за пальцем. На камень и цепь влияют все физические законы: трение, сила тяжести, ускорение, замедление, соударение при столкновении с другими объектами и прочее.

В этой главе мы не только узнали о физических соединениях, представленных в движке Torque 2D, но и испытали одно из них в своем демонстрационном приложении.

В следующей главе мы разработаем еще одну довольно-таки большую игру. Оставайтесь с нами!

Глава 10. Логическая игра Magic Mancala

Оглавление

1 Описание манкалы

Манкала — древнейшее семейство настольных игр для двух игроков. Историки и антропологи датируют его появление примерно тысячелетием до нашей эры. Манкала распространена по всему миру, особенно в Африке и Азии. При этом ее появления в разных частях света считаются независимыми. Поэтому у манкалы множество разных названий, правил и стилей.

Для игры в Манкалу используются доски с лунками, расположенные в несколько рядов, больше всего распространены доски с 2-мя и 4-мя рядами. Каждый ряд имеет одинаковое количество лунок. Число лунок в ряду в различных видах манкалы различно, оно колеблется от 5-ти до 10-ти. В лунки кладутся камешки. От вида манкалы зависит первоначальное количество камешков. Чем меньше лунок, тем игра происходит динамичнее. В некоторых играх имеются накопительные лунки больших размеров, называемые амбарами, казной или калахами. Они, как правило, располагаются справа от основного ряда игрока. Каждому игроку принадлежит ближний для него ряд лунок.

Игроки ходят по очереди. Ход игрока называется посевом. Во время хода из определенной лунки своего ряда изымаются все камешки и раскладываются во все последующие лунки по одной. Порядок лунок направлен против часовой стрелки. Если посев не завершился в ряду игрока, то есть пройдены все лунки, но остались неразложенные камешки, тогда их размещение продолжается в лунках в ряду оппонента, когда на доске 2 ряда, или в другом своем ряду, когда на доске 4 ряда лунок.

Цель игры — это захват камней противника, то есть надо набрать большее число камней, чем оппонент. При этом для удержания превосходства надо привести игру к такому состоянию, когда противник не может сделать ход. Кто это сделает, тот и победил. Правила захвата могут сильно отличаться от игры к игре. В некоторых играх, например, посев, заканчивающийся в лунке противника, приводит к захвату всех камней в этой лунке; или посев, заканчивающийся в пустой лунке, приводит к захвату всех камней в противоположной лунке (возможно, если выполнены дополнительные правила о числе камней в этой лунке). В других играх захватываются камни из лунок, в которых во время посева собирается определенное число камней. Захваченные камни либо изымаются из игры, либо размещаются в лунках игрока, который произвел захват.

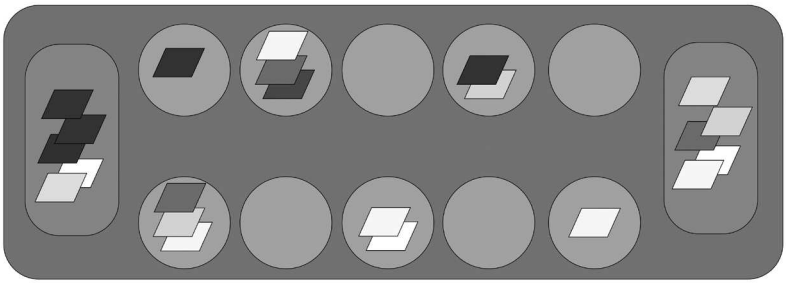

Рис. 10.1. Схематичное представление манкала

Как правило, в играх семейства манкала полностью отсутствует факт везения; таким образом, игру можно отнести к играм с полной информацией. Несмотря на то, что используются простые механизмы, сложность игры часто значительна. Как следствие, манкала представляет определённый интерес в различных разделах математики, таких как: теория игр, теория сложности.

Среди общих рекомендаций по стратегии игры следует отметить рекомендацию поддерживать собственную мобильность, то есть стараться, чтобы было по возможности большее число ходов, которые можно сделать без размещения камней в чужих лунках. Ходы, обеспечивающие собственную мобильность, часто более предпочтительны ходам, приводящим к захвату камней противника.

2 Дизайн документ для Magic Mancala

Как мы уже отмечали: существует множество видов манкалы. Давайте создадим новый тип этой игры — волшебная манкала или Magic Mancala. Прежде чем переходить непосредственно к разработке определимся, что мы хотим разработать, уточним детали — напишем дизайн документ.

Правила нашей игры будут, несомненно, основаны на оригинальной манкале. Доска в нашей игре будет иметь 2 ряда по 5 лунок в каждом. То есть по ряду для каждого игрока. Плюс к этому на двух противоположных концах доски будут находиться 2 калаха. Для игрока его калах будет находиться справа. В начале игры в каждую лунку помещаются по 4 камешка, в нашей игре их роль будут играть фишки — кристаллы. Для того

чтобы игроку было легче ориентироваться в содержимом лунок и калахов, рядом с каждым из них будет находиться счетчик, показывающий количество фишек, содержащихся в определенной лунке (или калахе).

Посев будет осуществляться автоматически: игрок выбирает (щелкнув курсором мышки или прикоснувшись пальцем) лунку, из нее забираются все фишки и последовательно раскладываются во все последующие лунки по одной фишке в каждую в направление против часовой стрелки. Если во время посева достигнут свой калах, в него так же кладется одна фишка. При этом если посев еще не закончен, то фишки раскладываются в лунки оппонента, последовательно по одной в каждую лунку. В то же время если в начале посева из исходной лунки было взято много фишек, то посев может не завершиться даже к концу ряда оппонента, в таком случае калах противника пропускается, и фишки продолжают раскладываться в лунки игрока, начавшего ход, так же по одной фишки в лунку последовательно. Таким образом, посев делает круговой оборот по всей доске.

Если последняя фишка из посева попадает в пустую лунку, тогда данная фишка, а так же все фишки из такой же лунки противоположного ряда переносятся в калах игрока, сделавшего ход.

Если последняя фишка из посева попадает в калах, тогда следующий ход делает тот же игрок, что и предыдущий, то есть ход не переходит оппоненту.

Как мы помним: цель игры захватить большее количество фишек, чем оппонент и привести игру к такому состоянию, когда невозможно сделать следующий ход. Но в нашей манкале можно выиграть, не положив в свой калах большее число фишек, если игрок накопит в своих лунках большее число фишек, тем самым оставив оппонента без возможности сделать ход, тогда игра завершается, и все фишки находящиеся в лунках игроков переносятся в соответствующие калахи, то есть фишки игрока — в его калах, фишки оппонента — в калах оппонента. После того как фишки окажутся в соответствующих калахах, считаются суммы всех фишек, находящиеся в калахах. И побеждает тот игрок, чья сумма фишек больше.

Все перечисленные выше правила в какой-то степени пересекаются с правилами настольных игр манкала разных видов. Главная особенность нашей манкалы — это использование магии. У игрока есть 4 вида магии: огонь, заморозка, молния и телепорт. Использование магии должно происходить перетаскиванием на любую лунку значка определенного заклинания, в результате чего должна проигрываться соответствующая

анимация и происходить определенное магией действие. Применение магии считается за 1 ход. После чего право сделать ход передается оппоненту, который так же может использовать магию.

Использование магии сопряжено с расходом манны — магической энергии. В начале игры пользователь получает 15 единиц манны. Счетчик манны должен выводиться в окне приложения. При каждом ходе, связанном с щелчком по лунке и перемещением фишек, игрок получает 1 единицу манны. Когда игрок использует магию огня, из общего количества манны вычитается 2 единицы, использование заморозки требует 3 единицы, молния — 4, а телепорт — 3. Если количества манны недостаточно, в таком случае нельзя выполнить заклинание.

При использовании на лунку магии огня, из нее удаляется 1 фишка; когда используется магия заморозки, делаются неактивными (замораживаются) 1 или 2 фишки, и в течение одного хода ими нельзя управлять, после чего они размораживаются — становятся активными; при использовании магии молнии, из лунки удаляются от 0 до 2-х фишек; а использование заклинания телепорта дает возможность переместить одну фишку из текущей лунки в любую другую.

Роль оппонента должен играть искусственны интеллект, другими словами, подпрограмма, составляющая часть игры. Он должен обладать теми же видением игрового поля и правами, что живой игрок. То есть ИИ не должен иметь дополнительных сведений или возможность жульничества в игре. В его праве то же самое, что у игрока.

После запуска приложения перед началом игрового процесса, пользователя должно встречать главное меню, откуда он может выбрать пункт для начала игры.

Во время игры у пользователя должна быть возможность приостановить игру, вызвав меню паузы, откуда он может либо продолжить, либо завершить игру.

Целевая платформа

Разработка будет вестись на платформе PC в операционной системе Windows. Целевой платформой являются планшеты и смартфоны с операционной системой Android. Так как для разработки мы будем использовать кроссплатформенный движок Torque 2D, игру можно будет реализовать на других поддерживаемых платформах, в том числе: на планшетах и смартфонах с операционной системой iOS.

Арт

Арт для манкалы очень специфичен. Он должен быть качественным и просто красивым — радовать глаз пользователя. Вы можете нарисовать свои картинки или воспользоваться теми, что прилагаются в материалах к книге.

Создание проекта: расположение папок и файлов

Создать заготовку для проекта можно с помощью менеджера проектов. Однако при разработке этого проекта я немного отклонился от стандартной заготовки, создаваемой менеджером проектов. Тем не менее менеджер проектов не создает панацею, он делает заготовку, чтобы ее можно было изменить.

Содержимое корневой папки проекта, созданного менеджером проектов, нас вполне устраивает. Нас интересует содержимое подкаталога modules. Здесь находятся 4 подпапки:

1. AppCore — стандартное содержимое — инициализационные и управляющие файлы;
2. MagicMancala — не содержит арта, только геймплейные скрипты с небольшой примесью управляющих;
3. Sandbox — папка, содержащая файлы, необходимые для запуска игры на Android-платформе (за подробностями обратитесь к главе 7);
4. ToyAssets — как раз здесь находятся все ассеты и арт для нашей игры: анимации — в подпапке animations, шрифты — в подпапке Fonts, изображения — в подпапке images;

3 Разработка логической игры MagicMancala

Создание окна вывода и сцены выглядит вполне стандартно и соответствует шаблонному коду из заготовки, поэтому рассматривать его не будем. Менеджер управления реагирует только на событие нажатия клавиши Escape, в результате чего происходит закрытие окна. При этом клавиша Escape есть только на компьютере, поэтому на мобильных платформах эта функциональность не нужна.

1 Начало выполнения игры — инициализация

Итак, дизайн документ написан, арт подготовлен, можно приступать к разработке, иными словами, кодированию нашей новой игры. Запустим

Torsion, в нем откроем проект MagicMancala.torsion, заботливо созданный менеджером проектов.

Согласно последовательности выполнения игры на движке Torque 2D, сразу после запуска исполняемого файла, управление передается в файл main.cs, находящийся в корневом каталоге игры. Это мы с вами уже проходили. Следующий код выполняется из файла \modules\MagicMancala\1\main.cs. Здесь находится конструктор и деструктор основного класса нашей игры — MagicMancala. В конструкторе подключаются все скриптовые файлы, инициализируется механизм случайных чисел, вызываются методы для создания окна сцены (createSceneWindow), самой сцены для размещения объектов (createScene), сцена подключается к окну (mySceneWindow.setScene(myScene)), создается объект управления (Control) и главное меню: MainMenu::create(); имеющее собственный класс. Конструктор класса MagicMancala выглядит следующим образом:

```
function MagicMancala::create( %this )
{
   exec("./scripts/init.cs");
   exec("./gui/guiProfiles.cs");
   exec("./scripts/scene.cs");
   exec("./scripts/control.cs");
   exec("./scripts/socket.cs");
   exec("./scripts/helpers/Time.cs");
   exec("./scripts/crystal.cs");
   exec("./scripts/gameGUI.cs");
   exec("./scripts/MagicIco.cs");
   exec("./scripts/AI.cs");
   exec("./scripts/background.cs");
   exec("./scripts/score.cs");
   exec("./scripts/StepOrderWin.cs");
   exec("./scripts/MagicEffect.cs");
   exec("./scripts/winWin.cs");
   exec("./scripts/loseWin.cs");
   exec("./scripts/pauseWin.cs");
   exec("./scripts/menuCommands.cs");
   exec("./scripts/MainMenu.cs");
   exec("./scripts/TextLabs.cs");

   %rand = getRealTime();
   setRandomSeed(%rand);
  createSceneWindow();
  createScene();
  mySceneWindow.setScene(myScene);
  MainMenu::create();
```

```
   new ScriptObject(Control);
   Control.InitControl();
}
```

В деструкторе класса MagicMancala выполняются только 2 действия: удаляется окно вывода сцены и разрушается элемент управления:

```
function MagicMancala::destroy( %this )
{
   destroySceneWindow();
   Control.delete();
}
```

Объект управления имеет класс Control, который описан в файле control.cs. В нем имеются только 2 метода: для создания и инициализации карты управления (moveMap) плюс запихивания в нее вызова одного обработчика:

```
function Control::InitControl()
{
if ( isObject( moveMap ) )
   moveMap.delete();
new ActionMap(moveMap);
moveMap.bindCmd(keyboard, "escape", "Control.
exitGame();", "");
moveMap.push();
}
```

Собственно, обработчик — exitGame вызывает только торковскую функцию quit, которая прекращает выполнение движка:

```
function Control::exitGame()
{
   quit();
}
```

Обратите внимание: как сказано в начале раздела, данная функциональность будет работать только на компьютере из-за присутствия только на клавиатуре клавиши Escape.

Создание сцены и окна вывода осуществляется стандартным образом в коде, подготовленном менеджером проектов, мы его уже рассматривали, поэтому не будем повторяться.

В подпапке \modules\MagicMancala\1\scripts\ создадим файл init.cs, который будет содержать все константы и глобальные переменные для игры:

```
//размеры области видимости
$cameraSizeWidth = 100;
$cameraSizeHeight = 75;
//скорость перемещения фишек
$moveSpeed = 30;
//расстояние между лунками
$distBetweenSockets = 14.15;
//номер сокета
$numSock = 0;
//максимальный номер сокета
$maxNumSock = 11;
//максимальное кол-во шагов: высчитывается в момент
//собирания цепочки
$maxStep = 0;
//семафор блокировки пользовательского хода:
//3 состояния:
//0 - никто не ходит
//1 - ходит 1-ый игрок
//2 - ходит 2-ой игрок
$semafor = 0;
//массив строк - картинок
$ToyAssetsPics = "ToyAssets:FireBall ToyAssets:Freeze
ToyAssets:Lightning ToyAssets:Teleport";
//массив цен на магию: их порядок соответствует
//порядку картинок --- для каждого уровня свой массив
$ToyAssetsPricesLevel1 = "2 3 4 3";
//в переменной сохраняется название текущего
//обрабатываемого ассета
$imageAsset = "";
//время, через которое выполнить следующее событие
$nextEventTime = 50;
//время целой анимации эффекта
$fullAnimTime = 1120;
//начальное значение манны
$beginMannaCount = 15;
//манна 1-го игрока
$fpMannaCount = $beginMannaCount;
//манна 2-го игрока
$spMannaCount = 5;
//режим выбора лунки (активируется при использовании
//магии "телепортации")
$chooseSocket = false;
//количество ходов до разморозки кристаллов
//для 1-го игрока
$unfreezeStepCount1 = 0;
//номер лунки, кристаллы в которой были заморожены
//для 1-го игрока
$freezeSocket1 = -1;//означает никакую
//для 2-го игрока
```

```
$freezeSocket2 = -1;//означает никакую
//открыто меню
$activeMenu = true;
//флаг конца игры:
//данный момент нужен, чтобы при конце игры,
//который произошел посредством переноса кристаллов из
//моей лунки и лунки оппонента
//не показывалось окно очереди хода
$endGame = false;
//флаг начала игры
//означает начало новой игры
$startGame = true;
//флаг, показывающий паузу
$pause = false;
//флаг, показывающий выполнение действия, имеет
//3 значения:
//0 - не выполняется,
//1 - выполняется,
//2 - выполнилось и надо перевести значение
$action = 0;
//во время исполнения магии нельзя ходить, однако при
//телепортации - можно
$execMagic = false;
//стоимость магии, переменная сделана глобальной в
//целях передачи значения между разными участками
//программы
$magicPrice = 0;
//итоги уровня:
//1-ая звезда: за один ход забрать у ИИ больше
//5 кристаллов
$star1_take5 = false;
//2-ая звезда: у игрока в 2 раза больше кристаллов,
//чем у ИИ
$star2_up2 = false;
//3-яя звезда: у игрока в 3 раза больше кристаллов,
//чем у ИИ
$star3_up3 = false;
//считатель ходов: в случае 5 непрерывных ходов игроку
//дается 5 звезд
$run_count = 0;
```

Сведений из комментариев пока достаточно для понимания предназначения переменных, в будущем при рассмотрении мест кода, где они используются, мы узнаем все детали в подробностях.

2 Фон

По традиции код для фона находится в файле background.cs. Создание фона происходит в функции createBackground:

```
function createBackground()
{
   %background = new Sprite();
   %background.Class = "background";
   %background.setBodyType( static );
   %background.Position = "0 0";
   %background.Size = $cameraSizeWidth SPC
$cameraSizeHeight;
   %background.SceneLayer = 31;
   %background.Image = "ToyAssets:background_green";
   %background.setUseInputEvents(true);
   myScene.add( %background );
}
```

Здесь все просто: создаем спрайт, сохраняем на него ссылку в переменной %background, организуем класс background, чтобы создать для него функции — обработчики событий, ставим объекту статический тип, позицию в начале координат (0 0), устанавливаем размер — на весь экран (окно), кладем на самый дальний слой — 31, загружаем для него изображение из ассета «ToyAssets:background_green», даем возможность получать сигналы ввода, и, наконец, запихиваем в стек сцены.

Ассет ToyAssets:background_green выглядит вполне стандартно — загружает текстуру:

```
<ImageAsset
   AssetName="background_green"
   ImageFile="background_green.png" />
```

Также в файле background.cs описаны обработчики событий:

- background::onTouchUp
- background::onTouchMoved
- background::onTouchDragged

Их работа связана с использованием заклинаний, ровно как функция checkToCreateEffect, поэтому оставим их рассмотрение до того момента, как познакомимся с магией.

В файле scene.cs кроме стандартного кода создания окна вывода и сцены создается доска для игры, состоящая из двух спрайтов: на нижнем слое размещается спрайт с отверстиями для вывода количества фи-

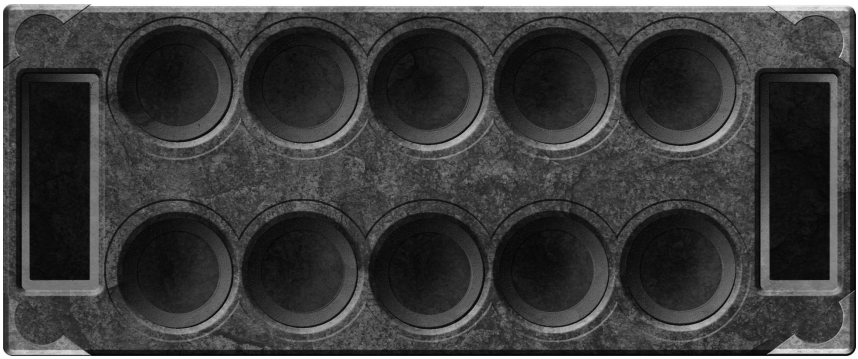

Рис. 10.2. Доска для игры в манкалу

шек в определенной лунке, а над ним размещается спрайт с текстурой доски, включающей лунки и калахи:

Функция создания доски — createBoard:

```
function createBoard()
{
   new SimSet(mySock);//общее вместилище, содержащее
// наборы для каждой лунки
   new SimSet(remCrys);//набор фишек, перемещаемых в
//калах в момент, когда  на последнем ходе в пустую
//лунку кладется кристалл, и с противоположной стороны
//из лунки забираются кристаллы оппонента
   new SimSet(moveCrys);//набор фишек, переносимых
//между лунками во время простого хода, используется
//для удаления фишек во время перезапуска игры из
//паузы
   new SimSet(FreezeCrystals2);//хранит замороженные
//кристаллы 2-го игрока
   new SimSet(FreezeCrystals1);//хранит замороженные
//кристаллы 1-го игрока

   %holes = new Sprite() {
     BodyType = static;
     Position = "0 0.16";
     Size = $cameraSizeWidth-9 SPC
$cameraSizeHeight/1.67;
     SceneLayer = 31;
     Image = "ToyAssets:black_holes";
  };
  myScene.add(%holes);
```

```
   %board = new Sprite();
   %board.setBodyType(static);
   %board.Position = "0 0.125";
   %board.Size = $cameraSizeWidth-4.3 SPC
$cameraSizeHeight/1.9;
   %board.SceneLayer = 31;
   %board.Image = "ToyAssets:board_stone";

   myScene.add(%board);
}
```

Как видно: в начале данной функции происходит создание глобальных объектов — наборов объектов — SimSet'ов: mySock, remCrys, moveCrys, FreezeCrystals2, FreezeCrystals1. Комментарии в коде подробно описывают их предназначение, поэтому не будем обсуждать их еще раз.

3 Кристаллы

Фон и доска — это базовые элементы цифровой манкалы, их роль по большей части декоративная. Центральное место в манкале играют фишки или кристаллы. Для нашей версии — волшебной манкалы подходит именно второе название — кристаллы. Потому как изображение этого элемента выглядит, как кристалл (рис. 10.3).

Рис. 10.3. Игровой кристалл

Вся игра сконцентрирована вокруг кристаллов: пользователь их выбирает, они переносятся с лунки в лунку, ведется их подсчет, от чего зависит победа в игре. В общей сложности у нас имеется 12 изображений для кристаллов разных цветов. В процессе игры от цветов ничего не зависит — все кристаллы равноценны. Разные текстуры нужны только для придания красоты игре. Есть еще один — 13-й кристалл, он представляет собой замороженную фишку. Таким образом, любой замороженный кристалл независимо от цвета принимает такой вид.

Поскольку кристаллы представляют самостоятельные объекты, создадим для них отдельный скриптовый класс, который поместим в обособленный файл. Создайте файл crystal.cs. Создание кристаллов осуществляется в функции createCrystals:

```
function createCrystals(%numSoc, %X, %Y, %side)
{
   new SimSet(sock @ %numSoc);
   mySock.add(sock @ %numSoc);
   for (%i = 0; %i < 4; %i++) {
  %cry = new Sprite(crys @ %i)
  {
       Size = "4 4";
       SceneLayer = 30;
       Image = "ToyAssets:" @ getCrystalAsset();
       class = "Crystal";
  };
  %cry.setSideName(%side);
  %cry.setBodyType(dynamic);
  %cry.setSockNum(%numSoc);
  %pos = getCrystalPos(%X, %Y, %numSoc);
  %xp = getWord(%pos, 0);
  %yp = getWord(%pos, 1);
  %cry.Position = %xp SPC %yp;
  (sock @ %numSoc).add(%cry);
  myScene.add(%cry);
   }
}
```

Функция вызывается в начале игры для каждой лунки по одному разу: она получает 4 параметра: %numSock — номер лунки (сокета) — счет с нуля; %X, %Y — координаты центра лунки по X и Y; %side — название стороны (right или left). Дело в том, что функция создает сразу 4 кристалла, то есть заполняет лунку. На правой стороне находятся лунки игрока, на левой стороне — лунки оппонента.

В начале тела функции создается контейнер определенной лунки для кристаллов:

```
new SimSet(sock @ %numSoc);
```

Он получает имя в зависимости от номера лунки — %numSock. Мы будем хранить в данном контейнере фишки, попавшие в определенную лунку. Затем мы помещаем этот именованный набор в глобальный контейнер для всех наборов mySock. После чего в цикле for, проходящем по числам от 0 до 4, создаются, естественно, 4 спрайта — кристалла. Для создания каждого спрайта мы применяем сокращенную форму записи без необходимости указывать объект перед свойством, это достигается путем открытия после вызова конструктора фигурной скобки. После этого можно писать свойства без имени объекта, так как по умолчанию

свойства относятся к создаваемому объекту. В завершении присвоения значений свойствам ставится закрывающая фигурная скобка с точкой с запятой:

```
%cry = new Sprite(crys @ %i)
{
       Size = "4 4";
       SceneLayer = 30;
       Image = "ToyAssets:" @ getCrystalAsset();
       class = "Crystal";
};
```

Объекты в конструкторе именуются, исходя из номера создаваемого кристалла — счетчика цикла. Размер кристалла: 4 × 4, 30-й экранный слой, как уже было сказано: для кристаллов отведен отдельный класс: Crystal. Имя изображения получается конкатенацией ToyAssets и значения возвращенного функцией getCrystalAsset. Эта функция специально предназначена для случайного выбора имени ассета кристалла:

```
function getCrystalAsset()
{
   %num = getRandom(1, 12);
   %str = "";
   if (strlen(%num) == 1)
  %str = "0" @ %num;
   else %str = %num;

   return %str;
}
```

Все ассеты кристаллов имеют имена от «01» до «12», иными словами ассет для 5-го кристалла имеет такое описание:

```
<ImageAsset
   AssetName="05"
   ImageFile="05.png" />
```

Остальные ассеты — похожи. Таким образом, функции getCrystalAsset надо сгенерировать случайное число от 1 до 12 (включительно): %num = getRandom(1, 12); Проверить, если длина сгенерированного числа 1 символ (числа меньшие 10), тогда добавить справа от цифры «0». Затем вернуть получившуюся или исходную строку.

Таким образом, конструктор класса Sprite получает имя необходимого для загрузки ассета.

После создания кристалла мы вызываем для него несколько методов: устанавливаем для него динамический тип, потому что ему придется перемещаться по доске. Методы setSideName и setSockNum являются моими модификациями, которые я внес в движок. Их мы рассмотрим несколько позже, когда научимся модифицировать исходный код движка Torque 2D. Тогда же мы узнаем, почему понадобилось вносить изменения в движок, а не ограничиться скриптовым кодом. Сейчас достаточно знать, что первая из них устанавливает для кристалла имя стороны, которой он принадлежит, а вторая — номер лунки.

В качестве параметров функция получила примерные координаты и название стороны, которой принадлежит лунка. Окончательную позицию для кристалла возвращает функция getCrystalPos, которая принимает координаты центра лунки и номер лунки:

```
function getCrystalPos(%X, %Y, %soc)
{
   %seed = 3.9;
   for (%i = 0; %i < 10; %i++) {//10 попыток найти
//не одинаковое место
  %sign = getSign();
  if (%sign)
       %xp = %X + getRandom(%seed);
  else %xp = %X - getRandom(%seed);
  if (%sign)
       %yp = %Y + getRandom(%seed);
  else %yp = %Y - getRandom(%seed);

  %xp = convertFloatToFloat(%xp);
  %yp = convertFloatToFloat(%yp);
  %is = false;//флаг отсутствия
  for (%j = 0; %j < mySock.getObject(%soc).getCount();
%j++) {
       %px = mySock.getObject(%soc).getObject(%j).
getPosition().X;
       %py = mySock.getObject(%soc).getObject(%j).
getPosition().Y;
       if (%xp == convertFloatToFloat(%px) && %yp ==
convertFloatToFloat(%py)) {
         %is = true;
         break;
       }//if
  }//for
  if (%is == false) break;
   }//for
   return %xp SPC %yp;
}
```

На вид функция выглядит довольно сложной, но, на самом деле, это не так. Почему нам нужна окончательная позиция кристалла? Иначе все кристаллы будут находиться один над другим, это будет выглядеть не естественно, так как в реальной жизни фишки бы не могли держаться столбиком и сразу бы рассыпались. Поэтому с помощью данной функции мы находим случайную позицию в рамках определенной лунки. В начале функции определяется предел разброса, другими слова, радиус — %seed; далее запускается цикл на 10 проходов, в каждый из которых находятся координаты по оси X и Y: сначала определяет знак (+/–) в функции getSign, она тоже выбирает по воле случае — случайно:

```
function getSign()
{
   if (getRandom() < 0.5)
  return false;
   else return true;
}
```

Стандартная торковская функция getRandom, вызванная без параметров, возвращает случайное число от 0 до 1. Когда число меньше 0.5 getSign возвращает false, иначе true.

В функции getCrystalPos, если getSign вернула положительное значение, тогда переменной %xp присваивается сумма координаты центра лунки по X и случайного значения от 0 до предела разброса — значения %seed. Если же getSign вернула отрицательное значение, переменной %xp присваивает разность вышеназванных операндов: %X и getRandom(%seed). Затем аналогичная последовательность операций выполняется для нахождения значения переменной %yp. После этого оба значения обрабатываются функцией convertFloatToFloat для преобразования числа с плавающей точкой неопределенной длины к виду: 2 знака после десятичной точки:

```
function convertFloatToFloat(%float)//преобразует
//длинное число с плавающей точкой в число с плавающей
// точкой определенной длины - 2
{//символа после десятичной точки
   %str1 = "";
   %str2 = "";
   %dot = false;
   %nstep = 0;
   for (%i = 0; %i < strlen(%float); %i++)
   {
  %ch = getSubStr(%float, %i, 1);
```

```
   if (!%dot) {
        if (%ch $= ".") {
           %dot = true;
           continue;
        }
        %str1 = %str1 @ %ch;
   } else {
        %str2 = %str2 @ %ch;
        %nstep++;
        if (%nstep == 2) break;
   }
    }
    if (%str2 $= "") %str2 = "0";
    %str = %str1 @ "." @ %str2;
    return %str;
}
```

Когда значения будут нормализованы происходит проверка: не заняты ли найденные координаты, чтобы не получилась стопка фишек. Для этого в цикле, проходящем по всем кристаллам, находящимся в данной лунке, их координаты сравниваются с найденными. В том случае, если находится совпадение, итерация повторяется. Если же совпадений не найдено, тогда цикл завершается и, найденные координаты возвращаются в функцию createCrystals. Но как мы отметили вначале: случайный поиск неповторяющихся координат выполняется только 10 раз. В случае, когда в лунке находится множество кристаллов, они могут перекрывать собой все пространство, и случайный поиск долго не сможет найти свободное место, поэтому мы ограничили цикл 10-ю итерациями, после чего прекращаем бессмысленный поиск, чтобы не тормозить игру, и кладем кристалл в найденные координаты, от этого большой стопки не получится.

После получения значения функция createCrystals присваивает его в качестве позиции созданному кристаллу. Предпоследним действием он добавляется в контейнер кристаллов лунки, а последним — в стек сцены для вывода на экран.

Удаление всех кристаллов вместе с содержащими их контейнерами происходит в функции deleteCrystals:

```
function deleteCrystals()//удаляет кристаллы вместе с
//логическими лунками
{
    while (mySock.getCount() > 0) {
   %cont = mySock.getObject(0);
   while (%cont.getCount() > 0) {
```

```
        %obj = %cont.getObject(0);
        %cont.remove(%obj);
        %obj.delete();
   }
   mySock.remove(%cont);
   %cont.delete();
    }

    if (isObject(FreezeCrystals1)) {
   while(FreezeCrystals1.getCount()) {
        %obj = FreezeCrystals1.getObject(0);
        FreezeCrystals1.remove(%obj);
        %obj.delete();
   }
    }
    if (isObject(FreezeCrystals2)) {
   while(FreezeCrystals2.getCount()) {
        %obj = FreezeCrystals2.getObject(0);
        FreezeCrystals2.remove(%obj);
        %obj.delete();
   }
    }

    while(moveCrys.getCount()) {
   %obj = moveCrys.getObject(0);
   moveCrys.remove(%obj);
   %obj.delete();
    }
    moveCrys.clear();
}
```

4 Сокеты

Сокеты или условные лунки представляют собой физические объекты, с которыми пользователь может взаимодействовать. Реальные лунки расположены на доске, и с ними невозможно взаимодействовать, так как они неотделимы от спрайта доски. Поэтому нам нужен объект, с которым можно взаимодействовать, чтобы он мог реагировать на нажатие указателем. Таким объектом может стать спрайт с прозрачной текстурой. Она находится в папке alphaRect, имеет название alphaRect-1.png. Между тем имя ассета socket1:

```
<ImageAsset
   AssetName="socket1"
   ImageFile="alphaRect-1.png" />
```

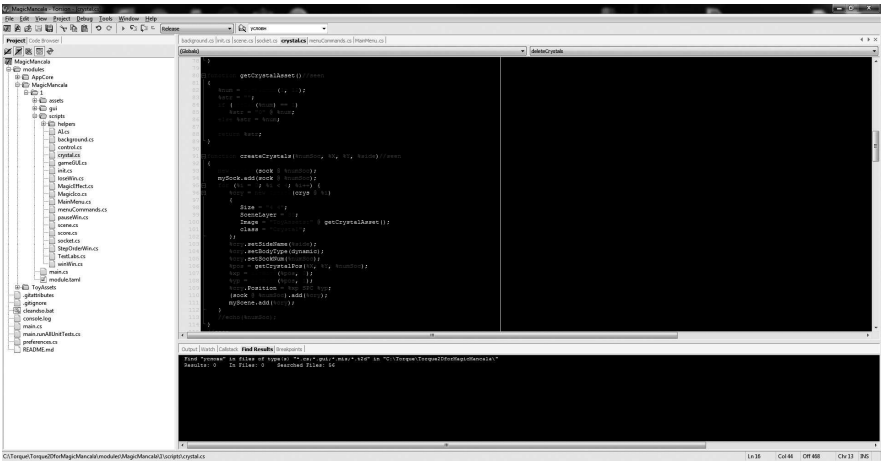

Рис. 10.4. Torsion IDE с открытым файлом crystal.cs

Таким образом, сокет представляет собой прямоугольный спрайт, находящийся над лункой и принимающий события от указателя.

Для сокетов мы создадим новый файл — socket.cs, в котором опишем класс Socket вместе с причастными к нему функциями.

Создание сокетов и некоторых других объектов происходит в функции createSockets. Это довольно большая функция, поэтому мы рассмотрим ее частями. Во-первых, эта функция не принимает никаких параметров. В ее начале осуществляется создание нижнего ряда лунок вместе с кристаллами:

```
function createSockets()
{
   %Y = -10;
   for (%i = 0; %i < 5; %i++) {
  %X = -28.25 + %i * $distBetweenSockets;
  %socket = new Sprite(myHole @ %i)
  {
       Position = %X SPC %Y;
       Size = "13.7 13.7";
       SceneLayer = 30;
       Image = "ToyAssets:socket1";
       class = "Socket";
  };
  %socket.setBodyType(static);
  %socket.setUseInputEvents(true);
```

```
   %socket.block = false;//разблокирован
   %socket.magic = "";//вначале к лунке не применена
//никакая магия;

   myScene.add(%socket);
   createCrystals(%i, %X, %Y, "right");
   TextLab::UpdateCounter(%i);
…
```

Сразу инициализируем переменную %Y, которая определяет расположение сокетов по оси Y. Потом запускается цикл на 5 итераций, где создаются лунки нижнего ряда. Сперва определяется значение переменной %X, которая хранит позицию для лунки по соответствующей координате: к смещению -28.25 прибавляется произведение счетчика цикла (или номеру сокета) и расстояния между лунками. Затем создается спрайт пользовательского класса Socket; тип статичный; разрешаем реагировать на ввод; отмечаем разблокирование состояние сокета и отсутствие выполнения магии. После чего добавляем сокет в стек сцены, чтобы иметь возможность взаимодействия с ним. Следующим действием происходит заполнение лунки кристаллами через вызов функции createCrystals, которую мы рассмотрели в предыдущем разделе. Функцией TextLab::UpdateCounter происходит обновление соответствующего данной лунке счетчика кристаллов. Мы рассмотрим это устройство чуть позднее.

Затем создается спрайт для подсветки:

```
%socket = new Sprite(teleLight @ %i)
{
   Position = %X SPC %Y;
   Size = "13.7 13.7";
   SceneLayer = 29;
   Image = "ToyAssets:socket1";
};
%socket.setBodyType(static);
myScene.add(%socket);
```

При создании ему назначается прозрачная текстура, однако во время активации телепортации на смену прозрачной текстуре будет назначена текстура в форме круга, но об этом позже. В качестве имени спрайт получает: teleLight + номер сокета. Размер и позиция такие же как у сокета, имеющего тот же номер. Далее спрайт получает статический тип и добавляется в стек сцены.

После создания условных лунок нижнего ряда, создается сокет для калаха справа:

```
//калах справа
 %socket = new Sprite(rightCallah)
 {
 Position = "41.5 0";
      Size = "9 25";
      SceneLayer = 30;
      Image = "ToyAssets:socket1";
      class = "Socket";
 };
 %socket.setBodyType(static);
 %socket.block = false;//разблокирован
 new SimSet(sock5);
 mySock.add(sock5);
 %socket.setUseInputEvents(true);
 myScene.add(%socket);
 TextLab::UpdateCounter(5);
```

Обратите внимание на различия позиции и размера, класс такой же — Socket. Еще одно отличие от сокета лунки, это то, что в данном случае контейнер кристаллов создается прямо здесь. Это связано с тем, что для калаха не вызывается функция createCrystals, которая создает контейнеры для лунок.

Затем %Y принимает новое значение: 10.5, и происходит создание лунок верхнего ряда. Все действия, рассмотренные выше, повторяются, однако теперь участвуют другие значения. Чтобы не дублировать большую часть кода, я не буду приводить в книге операции создания сокетов для лунок верхнего ряда, спрайта подсветки и левого калаха. Вы всегда можете посмотреть этот код в материалах к книге.

В конце функции createSockets глобальная переменная $semafor принимает значение 1. Это значит, что право хода передается первому игроку. В случае значения 2, ходит ИИ. Если же $semafor = 0, тогда никто не может совершить ход.

5 Обработка пользовательского ввода

Во время создания сокетов мы указали для них возможность получать команды, связанные с пользовательским вводом: %socket. setUseInputEvents(true);. Это означает, что при каком-либо вводе пользователя, будь то нажатие кнопки мыши, проведение указателем или пальцем, движок реагирует на это и передает управление в скриптовый

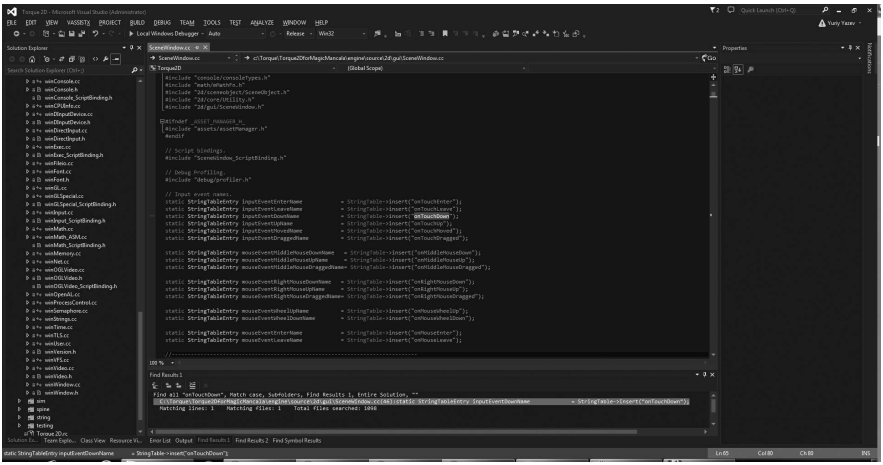

Рис. 10.5. Вызываемые из движка методы

код в соответствующую функцию — обработчик произошедшего события. Если такова не определена, то ничего не происходит, если же она описана, тогда выполняются определенные в ней действия.

Движок вызывает обработчики для объектов определенного класса. Класс определяется, как мы знаем во время создания объекта. Обработчики событий описываются, как методы этого класса. На какие события реагирует Torque 2D, и какие функции вызываются из скриптового кода, можно посмотреть в исходном коде движка (рис. 10.5).

Мы присвоили свойству Class спрайтов значение «Socket», тем самым мы определили новый класс, методы которого можем реализовать. В свою очередь, движок будет вызывать события класса предка в данном случае Sprite.

Нам надо обработать событие нажатия на спрайте — сокете. На скриншоте выше видно, что таким событием является onTouchDown. Оно ловит нажатие любым указателем, в том числе: курсором мыши или пальцем и, поэтому является универсальным. Событие принимает 3 параметра:

1) %this — указатель на объект, с которым произошло текущее событие нажатия;

2) %touchID — идентификатор пальца, которым осуществлено нажатие, другими словами, — номер;

3) %worldPosition — координаты точки в глобальной системе координат, где произошло нажатие;

Рассмотрим код данного обработчика события:

```
function Socket::onTouchDown(%this, %touchID,
%worldPosition)
{
   if ($activeMenu) return;

   if ($icoDown > 0) return;

   if (%this.block) return;

   if (remCrys.getCount() > 0)
  return;

   %num = getTrailingNumber(%this.getName());

   if (%num == -1)//калах
  return;

   if (%num > 4 && !$chooseSocket)
  return;

   if ($execMagic && !$chooseSocket)
  return;

  if ($chooseSocket) {
  %this.chooseSocket(%this);
  %this.lightSockets(%this, false);
  }
  else {
  if ($semafor == 1) {
  %this.Click(%this);
  }
  }
}
```

Однажды описанный обработчик вызывается при нажатии на каждом сокете.

В начале функции происходят несколько проверок: если открыто меню, тогда щелчка по лунке быть не может — покидаем обработчик; если глобальная переменная $icoDown больше 0, значит, выбрана какая-то магия, щелчок по лунке так же невозможен, так как мышь (или другой указатель находится в нажатом состоянии); если текущая лунка, на которой было совершено нажатие заблокирована, то выходим; и последнее,

от чего нам необходимо подстраховаться, чтобы в контейнере remCrys не было кристаллов, в обратном случае — выходим.

Если все проверки пройдены, следуем вниз по коду функции. Создавая сокеты, мы присваивали им уникальные имена: myHole/enHole + номер: от 0 до 5 (исключая последний) и от 6 до 11 (исключая последний). Следовательно, этот номер позволяет однозначно определить сокет. Поэтому нам надо его получить, это происходит в операторе:

```
%num = getTrailingNumber(%this.getName());
```

Функция getTrailingNumber возвращает оконечное число из названия, переданного ей в параметре, а в параметре мы передаем имя текущего сокета.

Стоп! В этом месте мы побежали вперед паровоза. Оказывается в движке Torque 2D отсутствует функция getTrailingNumber. Но почему же тогда Torsion выделяет ее, как ключевое слово? Она реализована в движке Torque 3D и там прекрасно работает, но почему-то в Torque 2D ее нет. Печально. Но не страшно! У нас есть исходный код движка, поэтому мы можем восстановить баланс силы, добавив в Torque 2D недостающую функцию.

6 Модификация движка Torque 2D

Одна из самых прекрасных особенностей движка Torque 2D — это наличие всего исходного кода на языке C++. И мы можем добавить любую недостающую функциональность.

Откройте в Visual Studio 2013 проект движка Torque 2D — файл sln: в моем случае: c:\Torque\Torque2D-dev-MM-book\engine\compilers\VisualStudio 2013\Torque 2D.sln, у вас начало пути может быть другим.

Как я говорил: нужная нам функция хорошо реализована в движке Torque 3D, который так же имеет открытый исходный код, поэтому мы можем позаимствовать реализацию функции оттуда.

Итак, в загруженном проекте Torque 2D в Visual Studio откройте файл \engine\source\string\stringBuffer.h. Прокрутите файл до конца объявления открытых (public) методов, перед разделом объявления закрытых методов — ключевым словом private поместите объявление функции GetTrailingNumber:

```
//функция извлечения концевого числа строки - аналог
//из Torque 3D
```

```
static StringBuffer GetTrailingNumber(const char* str,
S32& number);
```

Обратите внимание: мы объявили функцию статичной, из этого следует, что мы можем ее вызывать без необходимости создания объекта класса StringBuffer.

Далее откройте файл \engine\source\string\stringBuffer.cc. Прокрутите файл до конца, здесь мы поместим реализацию функции GetTrailingNumber:

```
//функция извлечения концевого числа строки - аналог
// из Torque 3D
StringBuffer StringBuffer::GetTrailingNumber(const
char* str, S32& number)
{
   // проверить строки
   if (!str || !str[0])
  return StringTable->EmptyString;
   // найти номер в конце строки
   StringBuffer base = str;
   const char* p = base.getPtr8() + base.length() - 1;
   // игнорировать пробелы на конце
   while ((p != base.getPtr8()) && dIsspace(*p))
  p--;
   // нужно хотя бы одно число
   if (!isdigit(*p))
  return base.getPtr8();
   // перебираем символы, пока не встретим не числовой
// символ
   while ((p != base.getPtr8()) && isdigit(*p))
  p--;
   // конвертация числа
   if ((*p == '-') || (*p == '_'))
  number = -dAtoi(p + 1);
   else
  number = ((p == base.getPtr8()) ? dAtoi(p) :
dAtoi(++p));
   // удаляем пробел между именем и номером
   while ((p > base.getPtr8()) && dIsspace(*(p-1)))
  p--;
    StringBuffer w = base.substring(0, p - base.
getPtr8()).getPtr8();
    return w;
}
```

В параметрах функция получает: указатель на строку и переменную-ссылку на число. Тип S32 равнозначен знаковому целому типу signed int.

Первым делом в теле функции проверяется переданная строка: если она пуста, тогда прекращаем выполнение функции и возвращаем пустую строку (объект класса StringBuffer), потому что делать больше нечего.

StringBuffer — это специальный класс движка Torque 2D, служащий для обертки строк и независимой работы с ними. Так как Torque 2D кроссплатформенный движок, он должен работать с разными форматами Юникода: 32, 16, 8. Класс StringBuffer предоставляет разные механизмы конвертации строк между различными форматами.

Затем создаем объект StringBuffer из переданной строки, чтобы потом создать указатель на последний символ переданной строки:

```
StringBuffer base = str;
const char* p = base.getPtr8() + base.length() - 1;
```

Смещаясь с последнего символа на предпоследний (и так далее) игнорируем пробелы, которые могут быть в конце строки:

```
while ((p != base.getPtr8()) && dIsspace(*p))
   p--;
```

Когда все пробелы будут пройдены, проверяем следующий (конечный) символ на принадлежность цифре, если это не цифра, можно выходить из функции, возвращая все слово:

```
if (!isdigit(*p))
   return base.getPtr8();
```

Если последний символ успешно прошел тест и относится к цифрам, тогда в цикле мы перебираем все символы, начиная с конца, которые принадлежат ряду цифр. Передвигая указатель на символ назад, мы тем самым расширяем строку (с конца), на которую указывает указатель (как бы масляно это не звучало):

```
while ((p != base.getPtr8()) && isdigit(*p))
   p--;
```

После этого, когда все цифры с конца будут выбраны, мы преобразуем получившуюся строку в число (в тип int) и присваиваем переменной number, переданной 2-м параметром:

```
number = ((p == base.getPtr8()) ? dAtoi(p) :
dAtoi(++p));
```

В следующем цикле мы удаляем пробелы, находящиеся между оконечным число и именем:

```
 while ((p > base.getPtr8()) && dIsspace(*(p-1)))
p--;
```

Когда дело будет сделано, на основе указателя мы формируем получившуюся строку — объект w класса StringBuffer:

```
StringBuffer w = base.substring(0, p - base.
getPtr8()).getPtr8();
```

То есть берем строку от начала и до символа, на который указывает указатель p. Оконечное число уже сохранено в переменной number, которая будет возвращена по ссылке. А строку w мы возвращаем с помощью оператора return.

Теперь функция GetTrailingNumber прекрасно работает в движке — в C++ коде. Но нам нужно работать с ней из скриптового кода на Torque Script. То есть нам надо сделать так, чтобы Torque Script увидел и смог вызывать эту функцию, иными словами, нам надо ее как-то экспортировать. Безусловно, в Torque 2D имеются такие механизмы.

Откройте для редактирования файл \engine\source\console\consoleFunctions.cc. Промотайте его вниз и добавьте 2 консольные функции:

```
//my_cod : функция извлечения концевого числа строки
ConsoleFunction(getTrailingNumber, S32, 2, 2, "")
{
    S32 suffix = -1;
    StringBuffer::GetTrailingNumber(argv[1], suffix);
    return suffix;
}
//my_cod : функция отбрасывает конечное число
ConsoleFunction(stripTrailingNumber, const char *, 2,
2, "")
{
    S32 suffix = -1;
    StringBuffer str = StringBuffer::GetTrailingNumber
(argv[1], suffix);
    const char* u = str.createCopy8();
    return u;
}
```

Консольные функции — это специальный тип функций движков семейства Torque, главная их особенность в том, что их можно вызывать из скриптового кода на Torque Script.

Итак, рассмотрим их подробнее. Первая функция ConsoleFunction(getTrailingNumber, S32, 2, 2, «») принимает 1 пользовательский параметр плюс параметр по умолчанию — указатель на вызывавший функцию объект. Пользовательским параметром является строка, из которой надо извлечь оконечное число. Во время выполнения данной функции вызывается написанная нами статическая функция StringBuffer::GetTrailingNumber. Ей передаются указатель на строку и ссылка на число, в которой возвращается искомый номер — оконечное число из строки. Уже это число посредством оператора return возвращается вызвавшему консольную функцию коду.

Вторая функция ConsoleFunction(stripTrailingNumber, const char *, 2, 2, «») принимает такие же параметры. Однако она действует по-другому. Она отбрасывает оконечное число и возвращает первую часть строки без номера. Для этого она сначала, так же, как предыдущая вызывает статическую функцию StringBuffer::GetTrailingNumber, сохраняя возвращенную строку в переменной str типа StringBuffer. Затем происходит преобразование этой строки в обычную 8-ми битную ascii-строку посредством метода createCopy8. Создается указатель на эту копию, и он возвращает вызвавшему скриптовому коду.

Еще нам надо подключить заголовочный файл для возможности работы с классом StringBuffer. Пролистайте файл кода вверх до раздела подключения заголовков и добавьте туда: #include «string/stringBuffer.h».

Для вступления изменений в силу необходимо перекомпилировать движок. Мы изменили только 3 файла, поэтому в полном перестроении движка нет необходимости, достаточно откомпилировать только изменившиеся файлы, а так же те, которые их используют. Для этого достаточно нажать F7, прежде выбрав нужный тип построения Debug или Release.

Теперь, когда нужная функциональность добавлена, движок пересобран, можно вернуться в скриптовый код для продолжения обзора геймплейного кода.

7 Скриптинг геймплея: лунки и сокеты

Мы остановились на том моменте, когда в переменную %num сохранили номер сокета, на котором совершено нажатие указателем. Следующим действием проверяем, чтобы номер не был равен -1, это может

быть в том случае, когда передаваемая функции getTrailingNumber строка не имеет оконечного номера. Когда нажатие происходит на калахах, возвращается -1, потому что в их названиях нет цифр: leftCallah, rightCallah. Далее если номер больше 4, то есть щелчок сделан ни на пользовательских лунках и одновременно глобальная переменная $chooseSocket сброшена (она установлена только, когда активен режим «телепортации»), происходит выход. Следующим действием проверяется включение переменной $execMagic (использование магии) и одновременное отключение выбора лунки, в таком случае происходит выход, так как не имеет право на развитие в таких обстоятельствах. Если все предыдущие проверки дали отрицательные результаты, благодаря чему выполнение функции не сброшено, и мы в этом месте, тогда при включенном режиме выбора лунки ($chooseSocket), происходит выбор текущей лунки — вызов функции chooseSocket данного сокета и отключение подсвечивания всех остальных лунок посредством выполнение функции lightSocket с параметром false. В случае, когда $chooseSocket неактивна, мы на всякий случай проверяем, чтобы $semafor был равен 1. Так как только при таком значение право хода у игрока. В таком случае вызывается метод Click текущего сокета.

В целом получился такой обработчик события нажатия на сокете, в большинстве своем состоящий из проверок.

Рассмотрим вызываемые из него функции. Когда используется магия телепортации, после того как соответствующее заклинание перенесено на лунку, из которой игрок хочет перенести кристалл в другую лунку, остальные лунки активизируются — подсвечиваются, становятся готовы принять кристалл ($chooseSocket = true). Игрок выбирает и щелкает на определенной лунке, нам как разработчикам совершенно все равно, какую лунку он выбрал, поскольку все лунки отвечают на нажатия одинаково. В таком случае вызывается метод chooseSocket. Он ничего не принимает, явно указанный параметр %this можно опустить, он передается по умолчанию.

Функция chooseSocket имеет такой вид:

```
function Socket::chooseSocket(%socket)
{
   if ($chooseSocket) {
   %socket.block = true;
   %num = getTrailingNumber(%socket.getName());
   %root_pos = 0;
   for (%i = 0; %i < teleportCrys.getCount(); %i++) {
        if (%root_pos == 0)
```

```
         %root_pos = teleportCrys.getObject(%i).
getPosition().X SPC teleportCrys.getObject(%i).
getPosition().Y;
       %pos = getCrystalPos(%socket.getPosition().X,
%socket.getPosition().Y, %num);
       teleportCrys.getObject(%i).Position = %pos;
   }
   $imageAsset = "ToyAssets:TeleportIco";
   MagicEffect::createMagicEffect(%socket, %socket.
getPosition(), true);
   %socket.schedule($fullAnimTime/2, "putCrystals",
%num);
   $chooseSocket = false;
    }
}
```

В ее начале еще раз на всякий случай проверяется включение режима $chooseSocket. Программы пишут параноики: они лучше дополнительный раз сделают проверку, чем что-то пропустят! Блокируем текущий сокет, из которого взяли кристалл, чтобы в него было нельзя вернуть этот кристалл. Получаем номер текущего сокета. Далее инициализируем переменную %root_pos нулем. После этого в цикле for пробегаем по содержимому контейнера (SimSet) teleportCrys, который создается и заполняется в момент использования заклинания телепортации. В теле цикла проверяем: равна ли переменная %root_pos нулю (состояние после инициализации), если да, помещаем в нее координаты по X и Y текущего объекта из контейнера teleportCrys, номером которого в данный момент является счетчик цикла. После этого в переменную %pos помещаем найденные с помощью функции getCrystalPos (рассмотрена выше) случайные и неповторяющиеся координаты для расположения кристалла в текущем (по которому щелкнули) сокете. Следующим действием размещаем выбранный из teleportCrys кристалл в найденных координатах. На этом итерация заканчивается и начинается следующая для другого элемента — кристалла из teleportCrys. При этом пока кристаллы невидимы.

Следующим действием помещаем в глобальную переменную $imageAsset имя ассета, который будет использоваться для создания эффекта. В следующей строке мы вызываем статическую функцию для создания эффекта:

```
MagicEffect::createMagicEffect(%socket, %socket.
getPosition(), true);
```

Подробнее мы рассмотрим магию и прилагаемые к ней эффекты поздней.

Далее происходит планирование выполнение метода putCrystals через $fullAnimTime / 2 миллисекунд для текущего сокета. Забегая вперед, этот метод кладет кристаллы в заданный номером сокет:

```
%socket.schedule($fullAnimTime/2, "putCrystals",
%num);
```

Последним действием функция chooseSocket сбрасывает глобальную переменную $chooseSocket.

Упомянутая выше функция putCrystals вызывается только в одном месте игры — в рассмотренном выше. Она выполняет последний этап телепортации кристаллов из одного сокета в другой, другими словами, кладет в лунку кристаллы, делая их видимыми, вместе с тем, очищая и уничтожая контейнер для телепортации кристаллов teleportCrys, рассмотрим код подробнее:

```
function Socket::putCrystals(%this, %num)//положить
// кристаллы в лунку
{
   while (teleportCrys.getCount()) {
  %obj = teleportCrys.getObject(0);
  %obj.Visible = true;
  mySock.getObject(%num).add(%obj);
  teleportCrys.remove(teleportCrys.getObject(0));
  TextLab::UpdateCounter(%num);
   }
   teleportCrys.delete();
   if (!$endGame)//только если игра не закончена!!!
  StepOrderWin::showStepOrderWin();
}
```

В цикле while перебираем все фишки, находящиеся в teleportCrys. Каждую из них делаем видимой, добавляем в контейнер, определенный значением переменной %num лунки, удаляем рассматриваемую фишку из контейнера teleportCrys, обновляем количество находящихся в сокете фишек, выводя это значение на табло. После конца цикла, когда все содержимое набора будет обработано, teleportCrys удаляется. Затем, если игра не закончена, выводится окно, сообщающее право хода.

Кроме chooseSocket из Socket::onTouchDown следом вызывается lightSocket. Эта функция подсвечивает или снимает подсветку. Ранее над каждой лункой мы создали спрайт teleLight, в обычном состоянии в

Рис. 10.6. Текстура подсветки

него загружена полностью прозрачная текстура, но, когда надо посветить лунку, то загружается текстура с кругом на прозрачном фоне (рис. 10.6).

```
function Socket::lightSockets(%this, %socket,
%switchOn)//подсветить лунки
{
   %num = getTrailingNumber(%socket.getName());
   for (%i = 0; %i < 11; %i++) {
  if (%i == 5) continue;
  if (%switchOn) {
       if (%i == %num) continue;
       (teleLight @ %i).Image = "ToyAssets:Light";
  } else (teleLight @ %i).Image = "ToyAssets:socket1";
   }
}
```

Функция принимает указатель на сокет, на котором совершено событие нажатия и булеву переменную %switchOn, значение которой указывает: включить подсветку или нет. Внутри функции первым делом получаем номер переданного сокета, затем перебирает номера от 0 до 11: если %switchOn включена, присваиваем спрайту teleLight с текущим номером счетчика ассет с именем ToyAssets:Light (подсветка), если же %switchOn выключена, тогда — текстуру ToyAssets:socket1 (полностью прозрачная). При этом если счетчик равен 5 (номер правого калаха) или номеру текущего сокета, пропускаем итерацию.

Когда игра находится в стандартном режиме, то есть выключен режим телепортации, в этом случае из обработчика события Socket::onTouchDown вызывается метод Socket::Click с единственным параметром — ссылкой на текущий сокет:

```
function Socket::Click(%socket)
{
   %name = %socket.getName();
   if (getCallah(%name) !$= "") return;
```

```
    %i = getTrailingNumber(%name);
    $numSock = %i;
    %side = getSideHoleName(%name);
    %max = mySock.getObject(%i).getCount();
    %semafor = $semafor;
    if (%max > 0 && $semafor < 2) {
   $semafor = 0;
   Score::scorePlus();
   $action = 1;
   $run_count++;
    }
    for (%j = 0; %j < %max; %j++) {
   %set = mySock.getObject(%i);
   %obj = %set.getObject(%j);
   if (%obj.image !$= "ToyAssets:ice") {
        %obj.setSockNum(%i);
        %obj.setStepNum(%i);
        %obj.setStepCount(%i+%j+1);
        %obj.setSideName(%side);
        if (%j == %max-1)
           $maxStep = %i+%j+1;
        moveCrys.add(%obj);
        %obj.stageOne();
   }
    }
    mySock.getObject($numSock).clear();
    TextLab::UpdateCounter($numSock);
}
```

Эта функция подготавливает механизмы игры для начала хода, инициализирует переменные, обновляет надписи, начинает первый этап хода. В целом ход состоит из трех этапов. Мы рассмотрим их поздней.

В начале функции Click после того, как получено имя сокета, мы убеждаемся, что это не калах. Для этого передаем имя сокета функции getCallah, если она возвращает не пустой результат, то объект — калах, значит, прекращаем выполнение функции. В ином случае извлекаем из имени оконечное число, сохраняя его в переменной %i. Это же значение сохраняем в глобальной переменной $numSock, в будущем пригодиться. В переменную %side сохраним название стороны, к которой принадлежит данный сокет. Это выполняется в функции getSideHoleName, в параметре принимающей имя сокета. Код ее выглядит весьма просто:

```
function getSideHoleName(%name)//по имени лунки
//возвращает соответствующую сторону
{
```

```
   %word = stripTrailingNumber(%name);
//правая и левая стороны - это, соответственно,
//нижняя - наша сторона и (верхняя) сторона противника
   if (%word $= "myHole")
   %side = "right"; else
   if (%word $= "enHole")
   %side = "left";

   return %side;
}
```

От имени сокета отбрасываем оконечный номер и получаем стандартного вида имя, сравнивая которое, однозначно определяем сторону.

В переменной %max сохраняем количество кристаллов текущей ячейки:

```
%max = mySock.getObject(%i).getCount();
```

Как мы помним: mySock хранит все контейнеры, они находятся в упорядоченном порядке, поэтому номер сокета соответствует номеру контейнера. getCount() возвращает количество объектов, хранящихся в указанном контейнере.

Следует проверка: чтобы в текущем сокете находился хотя бы один кристалл и $semafor был меньше 2, то есть ход делает игрок. При стечении этих обстоятельств, $semafor обнуляется (для невозможности сделать ход), происходит увеличение очков: вызов метода Score::scorePlus(). Глобальная переменная $action устанавливается в 1. Забегая вперед, отмечу, что она нужна для определения состояния выполнения действия. В следующей глобальной переменной — $run_count накапливается число непрерывно сделанных ходов, что влияет на итоговые результаты.

Далее запускается цикл от 0 до максимального количества кристаллов в лунке; в переменной %set сохраняется ссылка на текущий контейнер, а из него выбирается кристалл под номером счетчика и сохраняется в переменной %obj. Затем происходит проверка, чтобы свойство Image выбранного кристалла не было равно ToyAssets:ice, ведь в таком случае кристалл является замороженным, и с ним нельзя взаимодействовать.

Если проверка прошла успешно, и данный кристалл не заморожен, тогда устанавливаем свойства кристалла:

```
%obj.setSockNum(%i);
```

```
%obj.setStepNum(%i);
%obj.setStepCount(%i+%j+1);
%obj.setSideName(%side);
```

Первый метод устанавливает для кристалла номер родительской лунки. Второй метод устанавливает номер шага. Третий — устанавливает количество шагов, которое, в данном случае равно сумме %i (номер лунки), %j (номер кристалла), 1. Четвертый — устанавливает название стороны, к которой принадлежит лунка, где находится текущий кристалл.

Эти свойства и методы их установки реализованы в движке. Вполне закономерен вопрос: почему нельзя реализовать эти свойства в скриптовом коде? Мы ответим на него в следующем разделе данной главы.

А пока продолжим рассмотрение функции Click: после установки значений свойств кристалла, проверяем: ни последний ли кристалл лунки мы перебираем, если так оно и есть, тогда глобальной переменной $maxStep присваиваем максимальное количество шагов — сумму значений %i, %j и 1. После этого добавляем обрабатываемый кристалл в контейнер moveCrys, где хранятся фишки, переносимые между лунками. Следующим действием запускаем для определенного кристалла 1-й этап перемещения.

После того как цикл заканчивается, мы очищаем контейнер фишек текущего сокета и обновляем его счетчик кристаллов.

8 Различия между скриптовым кодом и кодом движка на C++

Действительно новые свойства добавляются к классам на языке Torque Script очень легко: достаточно в любом месте посредством точечной нотации добавить новое свойство и оно будет закреплено за этим объектом. Но, чтобы этими свойствами обладали все объекты определенного класса, надо добавить новое свойство или в описание класса, или в конструктор.

В Torque Script нет модификаторов доступности членов класса — все свойства и методы открытые (public).

Тем не менее в данном случае нам понадобилось сделать свойства членами класса на языке C++. Какие преимущества это нам дает?

Основное преимущество заключается в том, что при создании мультиплеерной игры, имея данные в движке (C++ коде), мы можем легко

передавать данные на сервер и всем клиентам. В Torque Script отсутствуют такие гибкие возможности.

Однако MagicMancala не является онлайновой игрой. Но, зная как сделать поддержку новых свойств и методов в классах движка, читатель сможет добавить мультиплеерный режим самостоятельно для возможности игры двух игроков друг против друга.

Итак, займемся добавлением новых свойств. В Visual Studio откройте файл \engine\source\2d\sceneobject\Sprite.h. Добавим в класс Sprite новые поля: в раздел private добавьте объявление следующих переменных:

```
// my_cod
    F32 mSockNum;//номер сокета
    F32 mStepCount;//количество шагов
    F32 mStepNum;//номер шага
    StringBuffer mSideName;//название стороны,
//к которой принадлежит данный спрайт
```

Сокеты создаются от класса Sprite, поэтому они будут обладать всеми свойства и методами, которыми обладает последний.

Между тем нам надо проинициализировать эти переменные. Откройте файл \engine\source\2d\sceneobject\Sprite.cc, перейдите в конструктор класса Sprite и добавьте инициализацию добавленных переменных:

```
Sprite::Sprite() :
   mFlipX(false),
   mFlipY(false),
     //my_cod
     mSockNum(0),
     mSideName("")
{
}
```

Плюс к этому мы хотим (на будущее), чтобы значения этих переменных передавались по сети. Для этого надо, используя оператор addField, добавить эти переменные. В Torque 2D у каждого класса есть специальный метод для осуществления этого действия:

```
void Sprite::initPersistFields()
{
   // Call parent.
   Parent::initPersistFields();
   /// Render flipping.
   addField("FlipX", TypeBool, Offset(mFlipX, Sprite),
```

```
&writeFlipX, "");
   addField("FlipY", TypeBool, Offset(mFlipY, Sprite),
&writeFlipY, "");
    //my_cod
    addField("SockNum", TypeS32, Offset(mSockNum,
Sprite));
    addField("SideName", TypeString, Offset(mSideName,
Sprite));
}
```

Метод addField принимает следующие параметры: имя поля, тип поля, смещение поля относительно начала класса, вычисляется посредством макроса Offset.

Это переменные, значения которых устанавливаются в приведенном в конце прошлого раздела коде с помощью методов, следовательно, нам надо добавить эти методы. Файл Sprite.h прокрутите вниз до раздела открытых членов класса (public) и добавьте туда следующие методы:

```
//my_cod
inline F32 getSockNum(void) const { return mSockNum; }
void setSockNum(const F32 num) { mSockNum = num; }
void incSockNum() { mSockNum++; }
void decSockNum() { mSockNum--; }
inline F32 getStepCount() const { return mStepCount; }
void setStepCount(const F32 stepCount)
{ mStepCount = stepCount; }
inline F32 getStepNum() const { return mStepNum; }
void incStepNum() { mStepNum++; }
void decStepNum() { mStepNum--; }
void setStepNum(const F32 step) { mStepNum = step; }
inline StringBuffer getSideName(void) const
{ return mSideName; }
void setSideName(const StringBuffer buf) { mSideName =
buf; }
```

Каждый из этих методов по продолжительности занимает одну строчку, поэтому мы поместили реализацию каждого из них прямо в заголовочный файл вместе с прототипом.

Метод getSockNum возвращает значение переменной mSockNum. Метод setSockNum устанавливает значение переменной mSockNum.

Метод incSockNum увеличивает значение переменной mSockNum на 1. Метод decSockNum уменьшает значение переменной mSockNum на 1.

Метод getStepCount возвращает значение переменной mStepCount. Метод setStepCount устанавливает значение переменной mStepCount на значение, переданное в параметре.

Метод getStepNum возвращает значение переменной mStepNum. Метод incStepNum увеличивает значение переменной mStepNum на 1. Метод decStepNum уменьшает значение переменной mStepNum на 1. Метод setStepNum устанавливает значение переменной mStepNum равное значению, переданному в параметре.

Метод getSideName возвращает имя стороны — значение переменной mSideName. Метод setSideName устанавливает переменную mSideName в значение, переданное в параметре.

Однако все эти методы можно вызвать только в пределах движка — из C++ кода. Поэтому нам надо добавить обертки для вызова этих методов из языка Torque Script.

Откройте файл \engine\source\2d\sceneobject\Sprite_ScriptBinding.h, переместитесь в его низ и добавьте туда следующие консольные функции:

```
//my_cod
ConsoleMethod(Sprite, getSockNum, F32, 2, 2, "()")
{
    return object->getSockNum();
}
ConsoleMethod(Sprite, setSockNum, void, 3, 3, "")
{
    object->setSockNum(dAtof(argv[2]));
}
ConsoleMethod(Sprite, getSideName, const char*, 2, 2,
"")
{
    StringBuffer buf = object->getSideName();
    const char* c = buf.createCopy8();
    return c;
}
ConsoleMethod(Sprite, setSideName, void, 3, 3, "")
{
    object->setSideName(argv[2]);
}
ConsoleMethod(Sprite, getStepCount, F32, 2, 2, "")
{
    return object->getStepCount();
}
ConsoleMethod(Sprite, setStepCount, void, 3, 3, "")
{
    object->setStepCount(dAtof(argv[2]));
}
ConsoleMethod(Sprite, incSockNum, void, 2, 2, "")
{
```

```
    object->incSockNum();
}
ConsoleMethod(Sprite, decSockNum, void, 2, 2, "")
{
    object->decSockNum();
}
ConsoleMethod(Sprite, getStepNum, F32, 2, 2, "")
{
    return object->getStepNum();
}
ConsoleMethod(Sprite, setStepNum, void, 3, 3, "")
{
    object->setStepNum(dAtof(argv[2]));
}
ConsoleMethod(Sprite, incStepNum, void, 2, 2, "")
{
    object->incStepNum();
}
ConsoleMethod(Sprite, decStepNum, void, 2, 2, "")
{
    object->decStepNum();
}
```

Тем самым мы просто добавили возможность вызова описанных ранее в классе Sprite методов из скриптового кода на языке Torque Script.

9 Остальные обработчики событий сокетов

В скриптовом классе Socket есть еще 2 обработчика событий. Первый из них — Socket::onRightMouseDown вызывается в момент, когда пользователь щелкает по сокету правой клавишей мыши. Сразу понятно: обработчик не кроссплатформенный. Он введен чисто для отладочных целей. Код, присутствующий в нем, выводит в консоль (закладка Output в Torsion) номер лунки, количество содержащихся в ней фишек, а так же имя и позицию каждой. Для экономии пространства я не буду приводить листинг обработчика в книге. Но вы всегда можете посмотреть его в материалах к книге.

Второй обработчик — Socket::onTouchUp вызывается при отпускании нажатого указателя (курсора мыши, пальца и прочее), вызывается из движка и принимает такие же параметр, как функция, вызываемая при нажатии — onTouchDown: указатель на объект, вызвавший обработчик, принявший нажатие, идентификатор указателя, двумерная мировая позиция указателя.

Код обработчика Socket::onTouchUp выглядит так:

```
function Socket::onTouchUp(%this, %touchID,
%worldPosition)
{
   //3-им
   %name = %this.getName();
   %num = getTrailingNumber(%name);
   if ($icoDown == 2 && $canCreateMagicEffect) {
     $icoDown = 3;
     if (%num > -1)
        $socket = %this;//сохраняем активный сокет
     else $socket = -1;
   }
}
```

Комментарий: «3-им» говорит о том, что сокет получает это событие 3-им в порядке использовании магии. Подробнее при рассмотрении магии. До него его получают: сначала заклинание, затем фон. Внутри обработчика получаем имя сокета и его номер. Его цель — сохранить ссылку на активный сокет в глобальной переменной $socket для дальнейшего использования.

10 Перенос кристаллов

Последовательность переноса кристаллов состоит из 3-х этапов — шагов. На 1-м шаге кристалл поднимается из лунки (по оси Y), на 2-м шаге кристалл перелетает к другой лунке по оси X), на 3-м шаге кристалл опускается в целевую лунку (по оси Y). В MagicMancala этот трехступенчатый процесс организован тремя функциями — методами скриптового класса Crystal.

Первый метод — stageOne класса Crystal, как мы видели выше: запускается из метода Click класса Socket. Таким образом, каждый кристалл выполняет полет обособленно от остальных. Но, так как для всех кристаллов лунки функция вызывается одновременно, полет выглядит синхронно:

```
$distDown = 10;//дистанция для подъема/спуска
function Crystal::stageOne(%this)
{
   %dist = $distDown;
   echo("stageOne = " @ %this.getStepCount());
   %pos = %this.getPosition();
   if (%this.getSockNum() < 5)
     %pos = setWord(%pos, 1, getWord(%pos, 1) + %dist);
   else
   if (%this.getSockNum() > 5)
```

```
      %pos = setWord(%pos, 1, getWord(%pos, 1) - %dist);
   %this.moveTo(%pos, $moveSpeed);
   %time = calcTime(%dist);
   %this.schedule(%time, stageTwo);
}
```

Дистанция для подъема и/или спуска определена в размере 10-ти метров (как вы помните, в Torque 2D измерения проводятся в метрах). В переменную %pos сохраняем позицию текущего кристалла. Далее с помощью метода getSockNum, возвращающего номер лунки, проверяем: в каком ряду находится кристалл: верхнем или нижнем, то есть лунки с номерами до 5 составляют нижний ряд, больше 5 — верхний. Если текущая лунка в нижнем ряду, тогда изменяем позицию — переменную %pos, увеличивая координату Y на %dist:

```
%pos = setWord(%pos, 1, getWord(%pos, 1) + %dist);
```

Если лунка в верхнем ряду, тогда изменяем переменную %pos путем уменьшения координаты Y на 10 метров (значение %dist):

```
%pos = setWord(%pos, 1, getWord(%pos, 1) - %dist);
```

Переменная %pos состоит из двух элементов — чисел; функция getWord возвращает элемент из строки, указанной в 1-м параметре, под номером, указанном во 2-м параметре (счет от нуля). Функция setWord устанавливает в строке, указанной в 1-м параметре, для элемента под номером, указанном во 2-м параметре, значение, указанное в 3-м параметре.

После этого для текущего кристалла мы вызываем метод moveTo, ему передаются 2 параметра: позиция (X Y), куда надо переместить текущий объект и скорость перемещения. Позиция хранится в переменной %pos, а скорость указана в глобальной переменной $moveSpeed:

```
%this.moveTo(%pos, $moveSpeed);
```

После конца перемещения по оси Y нам надо перейти на 2-й этап передвижения кристалла — перемещение по оси X. Другими словами, вызвать для кристалла метод stageTwo. Вопрос в том: как мы можем узнать, когда закончится 1-й этап — закончится перемещение по оси Y? В движке Torque 2D, когда заканчивается перемещения, начатое с помощью функции moveTo, происходит событие, которое можно обработать. Однако в таком случае получается, что наш код растечется по разным

функциям и его будет сложно поддерживать. Поэтому есть вариант лучше. Зная скорость перемещения кристалла и расстояние, мы можем вычислить время, через которое перемещение завершится, и можно будет запланировать вызов метода stageTwo. Вычислением времени занимается функция calcTime, вот как выглядит ее вызов:

```
%time = calcTime(%dist);
```

Функция calcTime является вспомогательной функцией, поэтому файл, в котором она описана — Time.cs находится в папке helpers. Ее тело занимает одну строчку:

```
function calcTime(%dist)
{
   return %dist / $moveSpeed * 1000.0;
}
```

Возвращаемый результат — время равно: расстояние (переданное в параметре) деленное на скорость ($moveSpeed) и умноженное на 1000, чтобы привести время к секундам.

Возвращенный функцией результат мы сохраняем в переменной %time, а затем планируем через этот промежуток времени вызов метода stageTwo для текущего кристалла:

```
%this.schedule(%time, stageTwo);
```

Этап №2 в процессе переноса кристаллов осуществляется в функции stageTwo:

```
function Crystal::stageTwo(%this)
{
   %dist = $distBetweenSockets;
   %i = %this.getSockNum();
   %j = %this.getStepCount();
   echo(%this.getStepNum() @ " = " @ %this.
getStepCount());
   if (%this.getStepNum() < %this.getStepCount()) {
  %this.incStepNum();
  echo(%this.getStepNum());
   if (%i == 11) %i = 0; else
  %i = %i + 1;
   }
   %this.setSockNum(%i);
   %pos = %this.getPosition();
   if (%i == 0) {
```

```
   %pos = leftCallah.getPosition();
   %pos = setWord(%pos, 0, getWord(%pos, 0) + %dist/2);
    } else if (%i < 5) {
   %pos = setWord(%pos, 0, getWord(%pos, 0) + %dist);
   %time = calcTime(%dist);
    } else if (%i == 5) {
   %pos = setWord(%pos, 0, getWord(%pos, 0) + %dist/2);
   %pos = setWord(%pos, 1, getWord(%pos, 1) + %dist/2);
   %time = calcTime(%dist/2);
    } else if (%i == 6) {
   %pos = rightCallah.getPosition();
   %pos = setWord(%pos, 0, getWord(%pos, 0) - %dist/2);
   //%time = calcTime(%dist/2);
   %time = calcTime(0);
    } else if (%i < 11) {
   %pos = setWord(%pos, 0, getWord(%pos, 0) - %dist);
   %time = calcTime(%dist);
    } else if (%i == 11) {
   %pos = setWord(%pos, 0, getWord(%pos, 0) - %dist/2);
   %pos = setWord(%pos, 1, getWord(%pos, 1) - %dist/2);
   %time = calcTime(%dist/2);
    }
    %this.moveTo(%pos, $moveSpeed);

    %this.schedule(%time, stageThree);
}
```

Расстояние между лунками: 14.15. В начале функции в переменных %i и %j мы сохраняем, соответственно, номер родительской лунки кристалла и его количество шагов, вызывая методы getSockNum и getStepCount.

Игра понимает: какую фишку класть в ближайшую лунку, когда, сверив количество сделанных шагов (перелетов от лунки к лунке) и максимальное количество шагов, обнаруживает, что значения равны.

Если же количество сделанных шагов меньше максимального количества, тогда увеличиваем количество шагов с помощью метода incStepNum. Если предыдущее количество шагов, сохраненное в переменной %i, равно 11 (количество лунок — счет с нуля), то сбрасываем его на 0, иначе увеличиваем на 1, и, наконец, изменяем номер родительской лунки с помощью метода setSockNum, передавая в параметре переменную %i. Вот такой круговорот кристаллов в манкале!

В переменной %pos сохраняем позицию обрабатываемого кристалла. В данный момент он находится в максимальной/минимальной позиции по оси Y. Так как изначально в лунке кристаллы лежат в уникаль-

ных (неповторяющихся позициях), то поднявшись над лункой на одинаковое расстояние по оси Y, они продолжают занимать уникальные позиции.

Далее выполняется длинный условный оператор, состоящий из 6-ти ветвей. Первая проверка успешна, если %i == 0, в таком случае, значит, что следующей родительской лункой является левый калах. Замечу: родительской лункой является не обязательно та лунка, в которую должен попасть кристалл, а та, к которой он должен подлететь, в таком случае получается, что фишки облетают игровое поле по кругу, перемещаясь от лунки к лунке, но, поскольку вычисления следующего шага происходит очень быстро, то задержка отсутствует. В переменной %pos сохраняем позицию левого калаха. Затем модифицируем ее значение: так как в порядке облета лунок (против часовой стрелки) левый калах идет после самой левой лунки верхнего ряда, в целевой позиции кристалла расположение по Y нас устраивает, однако по X надо прибавить частное %dist / 2. Такая целевая позиция будет означать, что кристалл не залетит на калах, а остановится на указанном расстоянии от него:

```
%pos = leftCallah.getPosition();
%pos = setWord(%pos, 0, getWord(%pos, 0) + %dist/2);
```

Иначе если значение %i меньше 5, но больше 0, из чего следует: выбрана лунка нижнего ряда, и нам надо изменить позицию кристалла по оси X, тем самым передвинуть его вправо:

```
%pos = setWord(%pos, 0, getWord(%pos, 0) + %dist);
```

Вдобавок надо вычислить время, через которое запустить метод stageThree для выполнения третьего этапа полета:

```
%time = calcTime(%dist);
```

Следующая ветка активизируется, когда родительская лунка кристалла имеет номер 5, в этом случае кристалл смещается по X и Y плюс происходит вычисление времени:

```
%pos = setWord(%pos, 0, getWord(%pos, 0) + %dist/2);
%pos = setWord(%pos, 1, getWord(%pos, 1) + %dist/2);
%time = calcTime(%dist/2);
```

Если целевая лунка под номером 6, тогда цель — правый калах, и нам надо изменить только координату по X, получив позицию правого калаха:

```
pos = rightCallah.getPosition();
%pos = setWord(%pos, 0, getWord(%pos, 0) - %dist/2);
%time = calcTime(0);
```

Время равно 0.

Случай, когда номер лунки меньше 11, но больше 6, означает лунку верхнего ряда, следовательно, движение фишек производится справа налево:

```
%pos = setWord(%pos, 0, getWord(%pos, 0) - %dist);
%time = calcTime(%dist);
```

Последний — замыкающий вариант происходит тогда, когда %i == 11, следствием чего происходит смещение фишки по обеим осям:

```
%pos = setWord(%pos, 0, getWord(%pos, 0) - %dist/2);
%pos = setWord(%pos, 1, getWord(%pos, 1) - %dist/2);
%time = calcTime(%dist/2);
```

После условного оператора вызывается метод moveTo для текущего кристалла, этот метод отправит данный кристалл в указанную позицию — %pos со скоростью $moveSpeed.

Последним оператором функции мы планируем выполнение метода stageThree через вычисленное ранее значение %time миллисекунд.

Метод stageThree завершает последовательность переноса кристаллов и так же вызывается для каждого кристалла. Этот метод является еще массивнее, а, следовательно, интереснее, чем предыдущие 2:

```
function Crystal::stageThree(%this)
{
   %gameOver = false;
   %dist = $distDown;
   %num = %this.getSockNum();
   %pos = getNextSocket(%num, %this);
   %del = false;//флаг того, что кристалл был удален
//или нет
   echo(%this.getStepNum() @ " = " @ %this.
getStepCount());
   if (%pos != 0 && %this.getStepNum() >= %this.
getStepCount()) {//последний кристалл - ставим в лунку
  %time = calcTime(%dist);
  %this.moveTo(%pos, $moveSpeed);
  %b = false;
  %nextStepByCurrPlayer = false;//следующий ход
//делает/не делает этот же игрок
```

```
   if ((%num == 5 || %num == 11) && $maxStep == %this.
getStepNum()) {
        %nextStepByCurrPlayer = true; //последний
//кристалл положен в калах, ходит этот же игрок
        Score::scorePlus();
   } else
   if (%num == 5 || %num == 11) {
        mySock.getObject($numSock).remove(%this);
        Score::scorePlus();
   } else
   if (%num != 5 && %num != 11 && $maxStep == %this.
getStepNum() && mySock.getObject(%num).getCount() == 0
&& checkRootSide(%num)) {
     %b = removeCrystals(%num, %this);//переместить
//свой последний в раздаче кристалл и кристаллы
//оппонента из противоположной лунки в свой калах
   }
   if (!%b)
        mySock.getObject(%num).add(%this);
   //проверяем магию лунки
   %del = checkMagicSocket(%num, %this);
   //обновляем счетчик кристаллов
   TextLab::UpdateCounter(%num);
    } else if (%pos == 0) {
             %this.decStepNum();
          }
    if (%this.getStepNum() < %this.getStepCount() &&
$maxStep > %this.getStepNum()) {
   %this.stageTwo();
    } else
    if (%this.getStepNum() >= %this.getStepCount() &&
$maxStep == %this.getStepNum()) {
   %gameOver = winGame(getInverseSide(checkToLo
se()));//во //как!!!! + проверка на победу/проигрыш
   if (!%b && !%gameOver && !%nextStepByCurrPlayer) {
        schedule(%time*2, 0, "StepOrderWin::showStepOrd
erWin");
   }
   if (%nextStepByCurrPlayer && !%gameOver) {
        InverseSemafor();
        if (!$endGame)//только если игра не закончена!!!
          StepOrderWin::showStepOrderWin();
   }
    }
}
```

Вначале инициализируем переменные: берем номер целевого сокета, получаем в нем случайную позицию для размещения кристалла. По-

следнее действие выполняется с помощью функции getNextSocket. Она получает 2 параметра: номер лунки и указатель на кристалл:

```
function getNextSocket(%num, %cry)//возвращает
//рандомную позицию в следующем сокете
{
   %hole = getSocket(%num, %cry);
   if (%hole $= "0")
  return 0;
   %pos = %hole.getPosition();
   %xp = getWord(%pos, 0);
   %yp = getWord(%pos, 1);

   if (%num != 5 && %num != 11) {
  %pos = getCrystalPos(%xp, %yp, %num);
   } else
   if (%num == 5 || %num == 11) {
  %pos = getCrystalPos2(%xp, %yp);
   }

   return %pos;
}
```

В начале своего выполнения функция getNextSocket для нахождения позиции в лунке должна сначала найти лунку, для этого она вызывает функцию getSocket, которая принимает те же параметры (номер лунки и указатель на кристалл):

```
function getSocket(%num, %cry)//находит имя сокета по
//его номеру и расположенному на нем кристаллу
{
   if (%cry !$= "")
  %side = %cry.getSideName();
   if (%num < 5)
  %hole = (myHole @ %num);
   else
   if (%num == 5) {
  if (%side $= "left")
       return 0;
  %hole = "rightCallah";
   } else
   if (%num < 11)
  %hole = (enHole @ %num);
   else
   if (%num == 11) {
  if (%side $= "right")
       return 0;
```

```
   %hole = "leftCallah";
    }

    return %hole;
}
```

Если переданный в функцию кристалл — не пустое значение, по хранящемуся внутри него значению можно узнать сторону, на которой находится этот кристалл. Это осуществляется с помощью метода getSideName кристалла, эту функцию мы добавили в движок собственноручно. Ее результат сохраняется в переменной %side. После этого происходит проверка: если переданный в функцию номер сокета меньше 5, значит, это сокет игрока, который имеет типичное имя myHole вместе с номером, в итоге мы конкатенируем эти 2 значения и возвращаем результат. В другом случае, если номер равен 5 (указывает на правый калах), а переменная %side равна %left, значит, произошел какой-то сбой, и мы возвращаем 0. Иначе переменной %hole мы присваиваем значение «rightCallah», значение которой возвращаем после выполнения функции. В другом случае, если номер сокета меньше 11 (случай, когда номер меньше 5 произойти уже не может), значит, сокет принадлежит стороне оппонента, условные лунки которой имеют названия: enHole + номер. В следующей проверке смотрится: если номер равен 11, при этом %side равен left, тогда произошел сбой, и мы должны вернуть 0, однако если %side не равен left (по определению, тогда он равен right), возвращаем leftCallah.

Функция getSocket рассмотрена, после нее выполнение возвращается в функцию getNextSocket (см. выше), которая, получив имя сокета, генерирует в нем новую уникальную позицию. Тем не менее, если getSocket возвращает 0 (мы увидели, в каких случаях это имеет место), тогда getNextSocket тоже возвращает 0. Однако если мы получили имя сокета, то мы берем его позицию, делим ее на компоненты:

```
   %xp = getWord(%pos, 0);
   %yp = getWord(%pos, 1);
```

Далее если номер сокета не равен 5 и не равен 11, тогда для получения рандомной позиции внутри лунки мы вызываем функцию getCrystalPos. Эту функцию мы рассматривали ранее, поэтому не будем повторяться. С другой стороны, если номер, напротив, равен 5 или 11, тогда — это калах, и для нахождения в нем случайной позиции мы вызываем функцию getCrystalPos2:

```
function getCrystalPos2(%x, %y)
{
   %seed = 2;
   %sign = getSign();
   if (%sign)
  %xp = %x + getRandom(%seed);
   else %xp = %x - getRandom(%seed);

   %seed = 7;
   %sign = getSign();
   if (%sign)
  %yp = %y + getRandom(%seed);
   else %yp = %y - getRandom(%seed);

   return %xp SPC %yp;
}
```

Она работает подобно getCrystalPos, только ищет позицию в калахе. Отличается она от последней тем, что не делает нескольких попыток для нахождения уникальной позиции. Это упрощение сделано по той причине, что в калахах будет находиться гораздо больше фишек, чем в лунках, и, поскольку в таком случае будет сложнее найти свободную — уникальную позицию, то не стоит и тратить время на не нужные итерации. К тому же, попав в калах, фишка уже не покинет его. Поэтому возвращаем позицию в вызвавшую функцию.

Теперь выполнение вместе с координатами позиции вернулись в функцию stageThree класса Crystal. На этом месте мы отмечаем флаг (булеву переменную) %del, как имеющую ложное значения, чтобы тем самым показать, что фишка еще не была удалена.

Затем попадаем в условный оператор. В нем происходит проверка: если %pos не равна 0 и количество сделанных шагов (перелетов от лунки к лунке) кристалла больше или равно максимально возможному количество шагов кристалла, тогда переходим к выполнению тела условного оператора. К слову, больше шагов, чем максимальное их количество кристалл сделать по определению не может. Внутри тела мы вычисляем время с помощью знакомой нам функции calcTime, передав ей дистанцию (10 метров). Далее опускаем кристалл вниз в лунку:

```
%this.moveTo(%pos, $moveSpeed);
```

Затем обнуляем булевы переменные %b и %nextStepByCurrPlayer, последняя служит индикатором: делает ли следующий ход текущий игрок. Осуществляем новую проверку: если номер текущего сокета, куда был

опущен кристалл, равен 5 или 11, из чего следует, что это калах, а так же, если глобальная переменная %maxStep, значение которой равняется текущему номеру шага, то мы делаем положительной переменную %nextStepByCurrPlayer и начисляем очки с помощью функции scorePlus класса Score. Сделав значение переменной %nextStepByCurrPlayer положительным, мы тем самым отметили что следующий ход делает тот же игрок. В другом случае, если номер так же равен 5 или 11, но %maxStep не равен номеру шага (что означает, что сделан ход не последним кристаллом), тогда мы удаляем кристалл из контейнера, откуда он был взят:

```
mySock.getObject($numSock).remove(%this);
```

Плюс начисляем очки:

```
Score::scorePlus();
```

Следующий вариант развития событий происходит тогда, когда номер сокета не равен 5 или 11, одновременно с этим выполняется условие на последний номер шага, рассмотренное выше ($maxStep == %this.getStepNum()), а так же, когда лунка, куда кладется этот последний в посеве кристалл пустая (mySock.getObject(%num).getCount() == 0), и, когда текущая лунка принадлежит той стороне, с которой начался ход.

Последнее условие проверяется в функции checkRootSide. Ее код выглядит так:

```
function checkRootSide(%num)
{
   if ($numSock >= 0 && $numSock <= 4
   && %num >= 0 && %num <= 4)
   return true;
   else
   if ($numSock >= 6 && $numSock <= 10
   && %num >= 6 && %num <= 10)
   return true;
   else
   return false;
}
```

В начале хода в глобальную переменную $numSock записывается номер лунки, с которой этот ход начинается, параметр %num равен номеру текущей лунки, куда кладется кристалл. Поэтому, если номер начальной и текущей лунок одновременно равен или больше 0, но равен или меньше 4 (нижний ряд), или номер начальной и текущей лунок одно-

временно равен или больше 6, но равен или меньше 10 (верхний ряд), значит, стартовая лунка и конечная лунка принадлежат одному ряду, возвращается true, в любом другом случае — возвращается false.

Если все вышеперечисленные условия совпадают, тогда выполняется функция removeCrystals, результат которой присваивается объявленной ранее переменной %b. Функция removeCrystals перемещает последний — единственный кристалл из своей лунки, а так же все кристаллы из противоположной лунки оппонента в свой калах. Представим эту функцию:

```
/*если последняя фишка кладется в пустую лунку, то
переместить имеющиеся в противоположной лунке фишки в
соответствующий калах*/
function removeCrystals(%num, %cry)//параметры: номер
//сокета, перемещаемый кристалл
{
   switch (%num) {
   case 0 : %sock = 10;
   case 1 : %sock = 9;
   case 2 : %sock = 8;
   case 3 : %sock = 7;
   case 4 : %sock = 6;
   //---
   case 10 : %sock = 0;
   case 9 : %sock = 1;
   case 8 : %sock = 2;
   case 7 : %sock = 3;
   case 6 : %sock = 4;
    }
    %res = false;//означает, что переданный в качестве
//параметра кристалл переместился или нет
    %top = mySock.getObject(%sock).getCount();
    if (%top > 0) {
   if (%top > 5) $star1_take5 = true;
   remCrys.clear();
   remCrys.add(%cry);
   %hole = getCallahName(%cry.getSideName());
   %pos_hole = %hole.getPosition();
   %pos_crys = getCrystalPos2(getWord(%pos_hole, 0),
getWord(%pos_hole, 1));
   %cry.moveTo(%pos_crys, $moveSpeed);
   while (mySock.getObject(%sock).getCount()) {
        %obj = mySock.getObject(%sock).getObject(0);
        remCrys.add(%obj);
        %pos_crys = getCrystalPos2(getWord(%pos_hole, 0),
getWord(%pos_hole, 1));
        %obj.moveTo(%pos_crys, $moveSpeed);
```

```
        mySock.getObject(%sock).remove(%obj);
        Score::scorePlus();
   }
   %socket = getSocket(%sock, "");
   %hyp = getHypotenuse(%socket.getPosition(), %pos_
hole);
   %time = calcTime(%hyp);
   schedule(%time, 0, endRemove, %hole);
   %res = true;
    }
    return %res;
}
```

В качестве двух параметров функция получает: %num — номер сокета, в который помещен последний кристалл, %cry — ссылка на последний кристалл. Тело функции начинается с оператора ветвления %switch, в котором происходит сравнение и подстановка соответствующего номера сокета, тем самым мы узнаем номер противоположной лунки. Объявляем и обнуляем переменную %res, которая станет результатом функции. Получаем и сохраняем в переменной %top количество кристаллов, находящихся в этой противоположной лунке. Следующий код выполняется только, если в лунке находится хотя бы один кристалл. Если в лунке лежит больше пяти кристаллов, то мы присваиваем глобальной переменной $star1_take5 значение true. Эта переменная сыграет свою роль в финале — в результирующем подсчете очков. Далее очищаем контейнер перемещаемых фишек и добавляем в него активный — последний кристалл (на который указывает %cry):

```
remCrys.clear();
remCrys.add(%cry);
```

Затем, вызвав функцию getCallahName, ее результат записываем в переменную %hole. Им является имя калаха соответствующей стороны: «rightCallah» или «leftCallah». Функция получает название стороны: «right» или «left»:

```
function getCallahName(%side)//возвращает имя калаха,
//соответствующее стороне
{
   switch$(%side) {
   case "right" : return "rightCallah";
   case "left" : return "leftCallah";
    }
}
```

Приведенный выше код не нуждается в комментариях.

Вернувшись в функцию removeCrystals, мы узнаем координаты выбранного калаха: %pos_hole = %hole.getPosition(); После чего выбираем случайную позицию в калахе: %pos_crys = getCrystalPos2(getWord(%pos_hole, 0), getWord(%pos_hole, 1)); Получив координаты для размещения кристалла в калахе, перемещаем в них фишку: %cry.moveTo(%pos_crys, $moveSpeed); Таким образом, мы разобрались с единственным кристаллом, в одиночку достигшим пустой лунки.

Дальше дело за содержимым противоположного сокета. Для этого в цикле while мы перебираем все имеющиеся там фишки, последовательно выбирая каждую и добавляя ее в контейнер для перемещения — remCrys. После этого с помощью функции getCrystalPos2 находим для каждой фишки позицию в калахе и помещаем ее туда — %obj.moveTo. Затем удаляем фишку из контейнера сокета, откуда она была перенесена: mySock.getObject(%sock).remove(%obj);, наконец, начисляем очки: Score::scorePlus();.

После завершения цикла нам надо вычислить время, через которое фишки достигнут калаха, и мы сможем продолжить игру, передав ход другому игроку. Сначала мы находим имя сокета, в котором располагаются кристаллы оппонента: %socket = getSocket(%sock, «»); Затем находим расстояние между позицией данного сокета и позицией активного калаха: %hyp = getHypotenuse(%socket.getPosition(), %pos_hole); Как видно: это осуществляется с помощью функции getHypotenuse, она получает: координаты двух позиций, принимает их за вершины прямоугольного треугольника, из чего узнает 2 катета, откуда вычисляет гипотенузу, поскольку фишки, перемещаясь из лунки в калах, движутся по диагонали — по гипотенузе прямоугольного треугольника:

```
function getHypotenuse(%pos1, %pos2)//находит
//гипотенузу для вычисления расстояния движения по
//диагонали
{
   %x1 = getWord(%pos1, 0);
   %y1 = getWord(%pos1, 1);
   %x2 = getWord(%pos2, 0);
   %y2 = getWord(%pos2, 1);
   %x3 = %x2;
   %y3 = %y1;
   %leg1 = mAbs(%x3 - %x1);
   %leg2 = mAbs(%y3 - %y2);
   %leg1 = %leg1 * %leg1;
   %leg2 = %leg2 * %leg2;
```

```
   %hyp = %leg1 + %leg2;
   %hyp = mSqrt(%hyp);

   return %hyp;
}
```

Здесь происходят типичные математические вычисления, а в конце функции, применив теорему Пифагора, мы получаем гипотенузу. Отмечу: mAbs — функция для нахождения абсолютного (положительного) числа, то есть, если число отрицательное, то минус отбрасывается; mSqrt — функция для нахождения квадратного корня из параметра.

После нахождения гипотенузы, с помощью уже знакомой нам функции calcTime мы можем вычислить время, необходимое для прохождения отрезка равного длине гипотенузы. Теперь мы можем запланировать выполнение завершающей перемещение фишек и ход в целом функции endRemove. Это действие выполняется с помощью функции schedule:

```
schedule(%time, 0, endRemove, %hole);
```

Если процесс выполнения кода добрался до этого места, значит, мы можем установить результирующую переменную %res в значение true. После этого закрывается единственная ветвь условного оператора (когда была проверка, чтобы в лунке был хотя бы 1 кристалл), и возвращается результат.

Как было упомянуто выше: процесс перемещения фишек завершает функция endRemove, она очищает контейнер переноса, перекладывает фишки в контейнер калаха, обновляет счетчик кристаллов, передает право хода:

```
function endRemove(%hole)//конец перемещения фишек
{
   %num = getCallahSock(%hole);
   %top = remCrys.getCount();
   while (%top) {
  %obj = remCrys.getObject(0);
  mySock.getObject(%num).add(%obj);
  remCrys.remove(%obj);
  %top--;
   }
   remCrys.clear();
   TextLab::UpdateCounter(%num);
   if (!$endGame)//только если игра не закончена!!!
  StepOrderWin::showStepOrderWin();
```

```
}
```

Она принимает имя калаха, преобразуя его в равнозначный индекс с помощью вызываемой функции getCallahSock:

```
function getCallahSock(%hole)//по имени калаха
//возвращает его индекс
{
   switch$(%hole) {
   case "rightCallah" : return 5;
   case "leftCallah" : return 11;
    }
}
```

Уверен, приведенный выше код не нуждается в дополнительных комментариях. Далее в функции endRemove берется количество фишек, содержащихся в контейнере remCrys, где находятся перемещаемые в калах фишки. Затем из данного контейнера один за другим кристаллы переносятся в контейнер калаха, удаляясь из remCrys:

```
%obj = remCrys.getObject(0);
mySock.getObject(%num).add(%obj);
remCrys.remove(%obj);
```

У калаха обновляется текстовая метка, отображающая количество содержащихся в нем фишек, в случае если игра еще не завершена, право хода передается оппоненту.

После выполнения функции removeCrystals управление возвращается в метод stageThree скриптового класса Crystal, где дальнейшее выполнение зависит от переменной %b. Как мы помним: эта переменная введена и проинициализирована нулем еще в начале метода (в первой же ветке условного оператора), поэтому даже если removeCrystals не выполнилась, переменная имеет значение. В случае если %b == false, а это, собственно, означает самый обычный ход, тогда в текущий сокет добавляется активный кристалл: mySock.getObject(%num).add(%this);.

Заклинания могут воздействовать не только на кристаллы, но и на лунки. Поэтому следующим действием мы проверяем магию лунки. Это действие выполняется в функции checkMagicSocket. Она получает: номер активного сокета и указатель на обрабатываемый кристалл:

```
//проверяем магию лунки
function checkMagicSocket(%socketnum, %crystal)
{
```

```
   %del = false;
   %hole = getSocket(%socketnum, %crystal);
   if (%hole.magic $= "fire1") {
  mySock.getObject(%socketnum).remove(%crystal);
  %del = true;
   }
   return %del;
}
```

В текущий момент развития игры MagicMancala у лунок поддерживается только магия огня. Выполнение этой функции заключается в проверке активизации магии огня на определенном сокете, имя которого находится по его индексу с помощью функции getSocket. Если магия огня активизирована на сокете, то положенный в лунку кристалл (ссылка на который передана в параметре %crystal) уничтожается, и возвращается положительное значение, если магии нет, возвращается отрицательное значение.

Вновь управление возвращается в stageThree, где, затем обновляется счетчик лунки. Напомню: мы тем самым завершили ветку условного оператора, где %b == false. В другом случае если %pos равняется строке «0», а это может произойти только в том случае, когда getNextSocket возвращает 0 по причине сбоя. Поэтому мы только уменьшаем количество шагов: %this.decStepNum();

Следующее условие проходит в том случае, когда количество сделанных шагов, возвращаемых методом getStepNum меньше, чем максимальное количество шагов, заметьте, это является анти условием к предыдущему. В таком случае вызывается метод текущего кристалла stageTwo, который переносит фишку к следующей лунке.

В ином случае, когда число сделанных шагов равняется их максимуму, происходит проверка на предварительные выигрыш и/или проигрыш. Почему предварительные? Потому что эта проверка вычисляет момент, когда игра завершилась и/или продолжение бессмысленно. Окончательное определение победителя будет произведено позднее. Проверка на предварительный выигрыш/проигрыш осуществляет функция checkToLose, не имеющая параметров:

```
function checkToLose()
{
   %lose = true;
   for (%i = 0; %i < 5; %i++) {
  if (mySock.getObject(%i).getCount() > 0) {
       %lose = false;
```

```
        break;
   }
    }
    if (%lose == true)
   return "right";

    %lose = true;
    for (%i = 6; %i < 11; %i++) {
   if (mySock.getObject(%i).getCount() > 0) {
        %lose = false;
        break;
   }
    }
    if (%lose == true)
   return "left";

    return "0";
}
```

Она возвращает название выигравшей стороны (left, right). Однако если игра не завершена, следовательно, никто не выиграл/проиграл, то возвращается 0. Определение выигравшей стороны происходит путем проверки наличия в сокетах одной из сторон фишек, то есть, если все лунки определенной стороны пусты, значит, эта сторона выиграла, и функция возвращает ее название. Данный трюк осуществляется через перебор лунок, принадлежащих одной из сторон и, в случае наличия в какой-либо лунке кристалла, меняется значение флага на false. Если же он равен true после завершения цикла, тогда возвращается название стороны, прерывая выполнение функции. В ином случае проверке подвергается вторая сторона.

Обратите внимание: функция возвращает название выигравшей стороны, однако нам надо название проигравшей стороны. Инвертировать сторону мы можем с помощью очень простой функции getInverseSide:

```
function getInverseSide(%side)//возвращает
// инвертированную сторону
{
   switch$(%side) {
   case "right" : %side = "left";
   case "left" : %side = "right";
    }
    return %side;
}
```

Она получает название стороны, а возвращает ее противоположность.

Последняя функция этой функциональной «матрешки» — winGame. На входе она получает выход функции getInverseSide (название стороны), а выдает: true — в случае завершения игры и false — в случае продолжения игры. Точнее, она возвращает false, когда от getInverseSide получает 0, которая возвращает данное значение, когда получает его от checkToLose.

Рассмотрим код winGame подробнее:

```
function winGame(%side)//переносит фишки победителя в
//его калах, подсчитывает очки, завершает игру
{
   if (%side $= "0") return false;

   %hole = getHoleNameBySide(getInverseSide(%side));
//Look out!!! getInverse
   %ind = getIndexBySide(%side);
   %callah = getCallahName(/*getInverseSide*/
(%side));//Look //Out!!! исправление правил

   %time = sendCrystalsToCallah(%hole, %ind, %callah);

   $endGame = true;//игра закончена

   schedule(%time, 0, "calcCollectCrystals");
//подсчитать количество набранных кристаллов, затем
//завершить игру

   echo("win");

   return true;
}
```

Как было сказано выше: вначале происходит проверка параметра, если он равен 0, то функция возвращает false. Дальше в переменной %hole мы сохраняем имя ячейки заданной стороны — инвертированного параметра, который передается функции getHoleNameBySide:

```
%hole = getHoleNameBySide(getInverseSide(%side));
function getHoleNameBySide(%side)//возвращает имя
//ячейки, соответствующее стороне
{
   switch$(%side) {
  case "left" : %hole = "myHole";
  case "right" : %hole = "enHole";
```

```
    }
    return %hole;
}
```

Затем мы узнаем индекс 1-й лунки в ряду посредством вызова функции getIndexBySide и передачи ей названия стороны:

```
%ind = getIndexBySide(%side);
function getIndexBySide(%side)
{
    switch$(%side) {
   case "left" : %ind = 6;
   case "right" : %ind = 0;
    }
    return %ind;
}
```

После этого нам еще нужно имя калаха, его мы получаем функцией getCallahName, которой передаем название стороны, а сохраняем в переменной %callah:

```
%callah = getCallahName((%side));
```

Функция getCallahName была уже нами рассмотрена.

Когда у нас есть все полученные выше значения, мы можем перенести все фишки из всех лунок в калах победителя. Это выполняется функцией sendCrystalsToCallah:

```
%time = sendCrystalsToCallah(%hole, %ind, %callah);
```

Рассмотрим ее подробнее:

```
function sendCrystalsToCallah(%hole, %ind, %callah)
{
    %count = 0;
    %pos = %callah.getPosition();
    %x = getWord(%pos, 0);
    %y = getWord(%pos, 1);
    %sock = getCallahSock(%callah);
    %root_pos = (%hole @ %ind).getPosition();
    %callah_pos = %pos;
    for (%i = %ind; %i < %ind + 5; %i++) {
   %max = mySock.getObject(%i).getCount();
   if (%max == 0) continue;
   %count += %max;
   while (mySock.getObject(%i).getCount() > 0) {
```

```
        %pos = getCrystalPos2(%x, %y);
        mySock.getObject(%i).getObject(0).moveTo(%pos,
$moveSpeed);
        mySock.getObject(%sock).add
(mySock.getObject(%i).getObject(0));
        mySock.getObject(%i).remove
(mySock.getObject(%i).getObject(0));
   }
    }
    %hyp = getHypotenuse(%root_pos, %callah_pos);
    %time = calcTime(%hyp);
    TextLab::UpdateCounter(%sock);

    return %time;
}
```

Напомню, в качестве параметров функция получает: имя сокета, индекс 1-го сокета в ряду, имя калаха выбранной стороны. В начале функции объявляется и обнуляется переменная %count, в %pos помещается позиция калаха, а в %x и в %y — ее компоненты. В %sock помещается индекс калаха, найденный с помощью getCallahSock. Затем конкатенируем значения переменных %hole и %ind, чтобы узнать имя 1-й лунки в ряду, после чего на основе полученного имени объекта узнаем его позицию (сохраняем ее в переменную %root_pos). В цикле for, начиная от %ind и заканчивая %ind + 5, перебираются сокеты, в каждой итерации цикла из сокета берется количество содержащихся в нем фишек, которые затем в цикле меняют свои принадлежность (между контейнерами лунки и калаха) и расположение (с помощью функции moveTo):

```
%pos = getCrystalPos2(%x, %y);
mySock.getObject(%i).getObject(0).moveTo(%pos,
$moveSpeed);
mySock.getObject(%sock).add
(mySock.getObject(%i).getObject(0));
mySock.getObject(%i).remove
(mySock.getObject(%i).getObject(0));
```

Между позицией лунки и позицией калаха вычисляется гипотенуза, по этому расстоянию мы вычисляем время, за которое фишки доберутся до места назначения, обновляем счетчик фишек и возвращаем время в качестве выходного параметра функции.

Получив время, выполнение функции winGame продолжается: мы устанавливаем глобальную переменную $endGame в значение true, тем самым показывая, что игра завершена. Затем планируем выполнение

функции calcCollectCrystals через полученное количество миллисекунд, хранящееся в переменной %time. После этого завершаем функцию, возвращая true.

Функция calcCollectCrystals подсчитывает набранные сторонами кристаллы, тем самым выявляя конечного победителя, и завершает игру, выводя поздравительное при выигрыше, либо омрачающее в случае проигрыша, сообщение (с помощью соответствующей функции) на экран для игрока. Функция выглядит следующим образом:

```
function calcCollectCrystals()
{
   %right = mySock.getObject(5).getCount();//правый
//калах
   %left = mySock.getObject(11).getCount();//левый
//калах
   if (%right > %left) {
      %prod = %right / %left;
      %prod = mFloor(%prod);
      if (%prod >= 3) $star3_up3 = true; else
      if (%prod == 2) $star2_up2 = true;
      Window::showWinWin();
   } else Window::showLoseWin();
}
```

Она ничего не получает и не возвращает. Она берет содержимое обоих калахов и сравнивает количественные показатели содержащихся в них фишек между собой. В случае если правый (пользовательский) калах содержит больше фишек, чем левый, тогда, кроме вывода окна поздравления игрока осуществляются вычисления: количество фишек правого калаха делится на количество левого, частное округляется к меньшему и помещается в переменную %prod. Если это значение больше или равно 3, тогда мы отмечает глобальную переменную $star3_up3, иначе, если %prod равно 2, тогда отмечаем глобальную переменную $star2_up2. Они играют свою роль во время выведения окна поздравления.

Если же в левом калахе больше фишек, чем в правом, значит, выиграл оппонент, и для игрока надо вывести окно с сообщением о проигрыше:

```
Window::showLoseWin();
```

Возвращенное функцией winGame значение помещается в локальную переменную %gameOver метода stageThree. Далее на ложное значение проверяются 3 переменные: %b, %gameOver,

%nextStepByCurrPlayer, это может быть в том случае, когда не было переноса кристаллов (в функции removeCrystals), не завершилась игра и следующий ход выполняет другой игрок, что соответствует планомерному игровому процессу. Если это имеет место, то мы планируем выполнение функции передачи права хода и показа окна: schedule(%time*2, 0, «Step OrderWin::showStepOrderWin»);

В последнем условии проверяется значения %nextStepByCurrPlayer и %gameOver: первая — положительная, вторая отрицательная, тогда вызывается функция InverseSemafor, которая меняет значение семафора на противоположное. Далее если игра все еще не завершена (переменная $endGame имеет нулевое значение), выводится окно, сообщающее о праве хода. Как мы видели ранее: значение семафора ($semafor) определяет очередность хода. Функция InverseSemafor имеет следующий вид:

```
function InverseSemafor()//инвертирует значение
//семафора
{
   switch($semafor) {
  case 2 : $semafor = 1;
  case 0 or 1 : $semafor = 2;
   }
}
```

Она находится в файле pauseWin.cs, и единственное, что делает — это меняет значение семафора на противоположное.

На этом обзор метода stageThree класса Crystal завершен, а вместе с ним закончен процесс переноса фишек. На первый взгляд это довольно-таки простое действие, но на самом деле оно требует внимания, усидчивости, удержания в уме большого числа условий и недюжинного труда программиста для своей реализации. Но мы с этим справились. Механизм переноса фишек является центральным местом во всей манкале, поэтому его реализации уделено столько внимания, остальные вещи будут менее сложными. Вперед!

11 Окно очереди хода

В прошлых разделах мы рассмотрели центральные части игры: сокеты, кристаллы и перемещение последних. Это не только самые важные части игры, но и самые большие. Разобравшись с кристаллами и сокетами, не составит большого труда разобраться в остальных классах MagicMancala.

На очереди небольшой класс StepOrderWin, который отвечает за окно, выводимое после каждого хода. В нем отображается информация о том, какой из игроков ходит следующим. На данный момент MagicMancala поддерживает двух игроков: 1 Player — человек, играющий в MagicMancala на своем устройстве и CPU — искусственный интеллект — программа, которая пытается обыграть человека. Как управляет игрой человек, мы уже рассмотрели в разделах, посвященных обработке ввода, как управляет игрой ИИ, мы разберем в следующем разделе. Однако сейчас увидим как работает окно для вывода порядка хода.

Вначале при запуске игры, создании различных меню и окон, в том числе создается окно порядка хода вызовом соответствующего статического метода: StepOrderWin::createStepOrderWin();. Сам метод выглядит так:

```
function StepOrderWin::createStepOrderWin()
{
   if (!SOWin.Active) {
   %win = new Sprite(SOWin)
   {
        Position = "0 0";
        Size = "40 20";
        SceneLayer = 29;
        Image = "ToyAssets:player1step";
        Visible = true;
   };
   myScene.add(%win);
    } else {
   SOWin.Image = "ToyAssets:player1step";
   SOWin.Visible = true;
    }
    schedule(1000, 0, "StepOrderWin::HideStepOrderWin",
"1");
}
```

Первое условие проверяет: если SOWin (что является создаваемым окном) не активно, что так же является индикатором того, что оно не создано, тогда создается спрайт SOWin, собственно, само окно. Сразу в конструкторе задаем значения свойств: позиция — в центре экрана, размер 40 × 20, экранный слой — 29, изображение берется из ассета «ToyAssets:player1step». Сразу показываем спрайт: Visible = true. За рамками конструктора добавляем созданный спрайт в стек экрана. Если же в первом условие выясняется, что спрайт SOWin активен, другими словами, создан, тогда загружаем текстуру и делаем его видимым.

В конце функции через секунду планируем выполнение метода HideStepOrderWin, который скроет окно порядка хода:

```
schedule(1000, 0, "StepOrderWin::HideStepOrderWin",
"1");
```

Метод принимает 1 параметр — %notRun.

```
function StepOrderWin::HideStepOrderWin(%notRun)
{
   SOWin.Visible = false;
   if (!$pause) {
  $activeMenu = false;//меню становится неактивно
  $startGame = false;//уже не старт
   }
  if (%notRun $= "") CheckSemafor();
  moveCrys.clear();
  $execMagic = false;//Look out!!!
}
```

Первым делом метод делает спрайт — окно SOWin не видимым. Также в этом методе обнуляются глобальные переменные: $activeMenu и $startGame при условии, что $pause отключена (равна false). Затем если параметр пуст, вызывается метод CheckSemafor. Независимо от значения параметра очищается контейнер для перемещения фишек moveCrys, сбрасывается булева переменная $execMagic (ее роль мы узнаем в будущем).

Последний метод, содержащийся в классе StepOrderWin — showStepOrderWin уже много раз встречался нам при рассмотрении предыдущих листингов. Его основная задача заключается в отображении окна порядка хода на экране с корректной картинкой, говорящей о том, кто ходит следующим.

```
function StepOrderWin::showStepOrderWin(%pauseExit)
{
   if (!$pause) {
  if ($freezeSocket1 > -1 || $freezeSocket2 > -1)
       MagicEffect::unFreezeCrystals();

  if ($run_count >= 5)//5 или более ходов
       $star3_up3 = true;

  switch($semafor) {
         case 0 or 1:
           if (%pauseExit !$= "")
```

```
            %img = "ToyAssets:player1step";
         else
            %img = "ToyAssets:cpustep";
      case 2 :
         if (%pauseExit !$= "")
            %img = "ToyAssets:cpustep";
         else {
            %img = "ToyAssets:player1step";
            for (%i = 0; %i < 11; %i++) {
                 if (%i == 5) continue;
                 %hole = getSocket(%i, "");
                 if (%hole.magic $= "fire1")
                      %hole.magic = "";
            }
         }
   }
   SOWin.Image = %img;
   SOWin.Visible = true;
   schedule(1000, 0, "StepOrderWin::HideStepOrderWin",
%pauseExit);
   $action = 0;
    }
    else
   $action = 2;
}
```

Как видно по коду, метод принимает 1 параметр — %pauseExit. Он используется только в передаче методу HideStepOrderWin и имеет не пустое состояние, когда выход произошел из режима паузы, во всех других случаях %pauseExit имеет пустое значение.

В начале тела метода проверяется значение глобальной переменной $pause: если она равна false, что происходит, например, в момент выхода из режима паузы, то следующая проверка выясняет значения глобальных переменных $freezeSocket1 и $freezeSocket2. В этих переменных хранятся номера лунок, соответственно, для 1-го и 2-го игрока, кристаллы в которых были заморожены. Если одна из этих переменных больше -1, что означает в соответствующей лунке имеются замороженные кристаллы, тогда вызывается метод MagicEffect::unFreezeCrystals();. Под его действием: замороженные кристаллы становятся размороженными. Он относится к магии, мы подробно его рассмотрим в соответствующем разделе.

Далее проверяется глобальная переменная $run_count, в которую записывается количество непрерывно сделанных шагов, то есть, когда право сделать ход не переходит к оппоненту. Если в ней находится зна-

чение больше 4, активируется глобальная переменная $star3_up3. Ее состояние учитывается в результативном окне. Следующим выполняется оператор ветвления, отталкивающийся от значения глобальной переменной $semafor. Это центральная часть рассматриваемой функции. Рассмотрим его условия и ветви. Если семафор равен 0 или 1, при этом значение параметра %pauseExit не пусто, тогда в переменной %img сохраняется ссылка на изображение из ассета «ToyAssets:player1step», с другой стороны, когда параметр пуст, сохраняем ссылку на изображение из ассета «ToyAssets:cpustep». Код ассетов выглядит тривиально:

```
<ImageAsset
   AssetName="player1step"
   ImageFile="player1step.png" />
```

Если семафор равен 2, вместе с тем значение параметра не пусто, тогда в переменную %img кладем ассет «ToyAssets:cpustep», иначе ассет — «ToyAssets:player1step», то есть при разных значениях семафора изображения противоположные. Вместе с этой ветвью оператора в последнем условии перебираются все лунки, в каждой их которых деактивируется магия огня.

После оператора ветвления в спрайт окна очереди хода загружается выбранная текстура: SOWin.Image = %img;, и этот спрайт делается видимым. Через 1000 миллисекунд планируется на выполнение метод HideStepOrderWin, который скроет спрайт окна: schedule(1000, 0, «StepOrderWin::HideStepOrderWin», %pauseExit);. Глобальная переменная $action сбрасывается в 0.

Если самое первое условие не выполняется: когда активна переменная $pause, $action принимает значение 2. Напомню: $action имеет 3 значения: 0 — действие не выполняется, 1 — выполняется, 2 — выполнилось и надо перевести значение.

Рис. 10.7. Ходит игрок

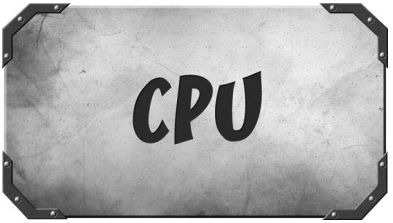

Рис. 10.8. Ходит искусственный интеллект

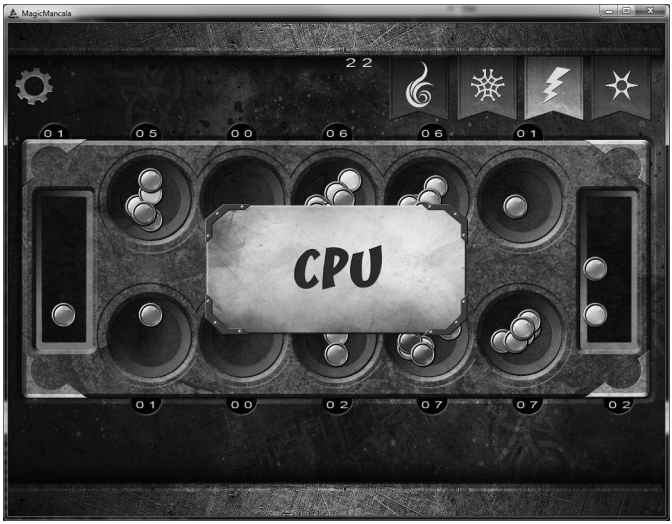

Рис. 10.9 Окно порядка хода

На этом рассмотрение класса StepOrderWin закончено. Это было не сложно! Двигаемся дальше!

12 Искусственный интеллект

На самом деле искусственный интеллект в MagicMancala очень прост. Его код содержится в файле AI.cs и занимает всего 52 строчки. Мы изначально не собираемся создавать непобедимую программу, включающую в себя механизм обучения, нейронные сети и другие достижения в области машинного разума. Это потребовало бы неоправданно много сил и времени. А часто бывает так, что в разработке игры надо добиться не лучшего результата с затратами больших ресурсов, а среднего и лишь похожего на исключительный, но с минимальными затратами. Так дела обстоят с системой ИИ в MagicMancala. Нам вполне достаточно, чтобы компьютерный игрок делал ходы и использовал магию.

Как уже было сказано выше: центральную роль в определении последовательности хода играет семафор. В момент закрытия окна порядка хода — SOWin вызывается функция CheckSemafor. Она не принимает параметров и выглядит следующим образом (место ее расположения — файл crystal.cs):

```
function CheckSemafor()
{
   switch($semafor) {
        case 0 or 1:
          $semafor = 2;
          if (!$pause)
            AI::Click();
        case 2 :
          $semafor = 1;
   }
   unblockAllSockets();
}
```

Первым делом в функции выполняется оператор ветвления, он проверяет значение глобальной переменной $semafor: если его значение 0 или 1, тогда новое значение 2, при этом, если игра не находится в режиме паузы (переменная $pause равняется нулю), тогда совершается ход ИИ, то есть вызывается метод AI::Click(). Вторая ветка оператора ветвления вызывается тогда, когда $semafor равен 2, она только делает значение этой переменной равным 1. После оператора ветвления вызывается unblockAllSockets. Эта функция перебирает все сокеты и снимает с каждого из них блокировку, при этом, пропуская калах (функция находится в файле socket.cs):

```
function unblockAllSockets()
{
   for (%i = 0; %i < 11; %i++) {
   if (%i == 5) continue;
   if (%i < 5)
        (myHole @ %i).block = false;
   else if (%i > 5)
        (enHole @ %i).block = false;
   }
}
```

Подготовительные действия перед выполнением хода ИИ рассмотрены, теперь перейдем непосредственно к рассмотрению последнего, он происходит в методе Click класса AI:

```
function AI::Click()
{
   if (AI::fullSockets()) {
   if ($semafor == 2) {
        $run_count = 0;
        $action = 1;
        %count = 0;
```

```
        %hole = 0;
        while (%count == 0) {
           %hole = getRandom(6, 10);
           %count = mySock.getObject(%hole).getCount();
        }
        %socket = "enHole" @ %hole;
           if ((%score = AI::useMagic()) < 0)
              %socket.Click(%socket);
   }
    } else
   %gameOver = winGame(getInverseSide(checkToLose()));
}
```

Следуя своему названию, этот метод представляет щелчок, генерируемый системой ИИ. Метод ничего не получает и не возвращает, начинается с условия, в котором с помощью метода AI::fullSockets проверяется наличие в одной из лунок на стороне ИИ кристалла:

```
function AI::fullSockets()//возвращает true, если хотя
//бы одна из лунок содержит кристаллы
{
   for (%i = 6; %i < 11; %i++)
   if (mySock.getObject(%i).getCount() > 0)
        return true;
   return false;
}
```

Уверен, вышеприведенный код не требует дополнительных комментариев. Если этот метод возвратит true, следующее условие проверяет (на всякий случай: дополнительная проверка никогда не бывает лишней) значение семафора, которое по определению в случае хода ИИ должно быть ровно 2. Дальше присваиваем переменным значения и попадаем в цикл while. В условии проверяется значение переменной %count, если оно равно 0, тогда выполняется тело цикла, где случайным образом на стороне ИИ выбирается лунка (от 6 до 10), номер сохраняется в переменной %hole. Затем из выбранной лунки выбираются кристаллы, их количество сохраняется в переменной %count, а последняя проверяется в условии цикла. Когда цикл пройден, в переменную %socket формируется имя выбранной лунки: имя вражеской лунки — «enHole» конкатенируется с номером — %hole. После этого вызывается метод useMagic класса AI. Числовой результат метода присваивается переменной %score. В методе useMagic выбирается заклинание и сокет, на который использовать выбранное заклинание:

```
function AI::useMagic()
{
   //определяем сокет
   %num = AI::selectRandomSocket();
   $socket = getSocket(%num, "");
   //определяем магию
   %num = getRandom(0, 2);
   $magicPrice = getWord($ToyAssetsPricesLevel1,
%num);
   $imageAsset = getWord($ToyAssetsPics, %num);
   $imageAsset = $imageAsset @ "Ico";

   return checkToCreateEffect();
}
```

Для выбора сокета используется метод AI::selectRandomSocket():

```
function AI::selectRandomSocket()
{
   %num = getRandom(0, 10);
   while (%num == 5)//правый калах
  %num = getRandom(0, 10);

   return %num;
}
```

В нем берется случайное число от 0 до 10, и пока это число равняется 5, попытки взять случайной число продолжаются. Как только подходящее число найдено, оно возвращается в метод useMagic. В нем с помощью рассмотренной ранее функции getSocket мы получаем имя сокета с индексом, возвращенным методом selectRandomSocket, затем сохраняем это имя в глобальной переменной $socket для дальнейшего использования. Определение заклинания начинает с взятия случайного числа от 0 до 2 (ИИ не может использовать магию телепортации). Далее в глобальной переменной $magicPrice мы сохраняем стоимость выбранного заклинания:

```
$magicPrice = getWord($ToyAssetsPricesLevel1, %num);
```

В глобальной переменной $ToyAssetsPricesLevel1 находятся цены на магию, разделенные пробелами:

```
$ToyAssetsPricesLevel1 = "2 3 4 3";
```

Их порядок соответствует условному порядку заклинаний. Для выбора определенной цены — элемента строки, используется функция getWord, в первом параметре указывается строка, а во втором — номер выбираемого элемента.

В переменной $imageAsset сохраняем анимационный ассет, который будет воспроизведен при активации данного заклинания. Все ассеты изображений для заклинаний, разделенные пробелами, хранятся в списке $ToyAssetsPics, откуда выбираются посредством функции getWord:

```
$imageAsset = getWord($ToyAssetsPics, %num);
```

Следующим действием к значению переменной $imageAsset конкатенируется последовательность символов «Ico»:

```
$imageAsset = $imageAsset @ "Ico";
```

Это нужно, потому что функция создания магии ждет параметр именно в таком формате, почему — разберем в ближайших разделах.

В заключение метода useMagic вызывается функция checkToCreateEffect, результат которой возвращается в метод AI::Click. Саму функцию checkToCreateEffect мы рассмотрим позже.

Результат AI::useMagic сохраняется в переменной %score, и, если он меньше 0, тогда вызывается метод Click класса Socket:

```
%socket.Click(%socket);
```

Действия, выполняемые в этом методе, рассмотрены нами ранее.

С другой стороны, если выполненная нами в начале проверка (AI::fullSockets) окажется неудачной, то есть на стороне оппонента нет заполненных лунок, тогда надо проверить игровую доску на конец игры:

```
%gameOver = winGame(getInverseSide(checkToLose()));
```

Данную конструкцию мы уже рассматривали, не будем повторяться.

На этом обзор класса AI завершен. Мы увидели, как всего в несколько десятков строчек кода можно реализовать искусственный интеллект для игры, однако для этого, ранее реализуя функции для осуществления хода, мы сделали их максимально гибкими для управления игрой.

13 gameGUI.cs

В файле gameGUI.cs находится набор функций, служащих для создания элементов пользовательского интерфейса внутри игры. Рассмотрим их по порядку — сверху вниз. После того как уровень будет загружен, вызывается функция createGameGUI. Она ничего не получает и не возвращает:

```
function createGameGUI()
{
   createOptionsBut();
   createMagicSlots();
   createUpStub();
   createDownStub();

   for (%i = 0; %i < getWordCount($ToyAssetsPics);
%i++) {
  MagicIco::createMagicIco(%i, getWord($ToyAssetsPics,
%i));
   }

   StepOrderWin::createStepOrderWin();
   Score::createMannaScore();
}
```

Как видно из приведенного выше листинга: из нее вызываются другие функции, а так же методы для создания объектов классов.

С помощью функции createOptionsBut создается кнопка для открытия меню паузы:

```
function createOptionsBut()
{
   %but = new Sprite(butOptions)
   {
  Position = "-46 28";
  Size = "6 6";
  SceneLayer = 31;
  Image = "ToyAssets:gear";
  UseInputEvents = true;
  class = "butGamePause";
   };
   myScene.add(%but);
   %but.rotate(100);
}
```

Эта кнопка представляет собой вращающуюся в разные стороны шестеренку:

Рис. 10.10. Вращающаяся шестеренка — кнопка вызова паузы

Ассет для загрузки изображения достаточно прост:

```
<ImageAsset
   AssetName="gear"
   ImageFile="gear.png" />
```

В качестве базового объекта выбран удобный класс Sprite, с его конструктором и параметрами мы уже много раз встречались. Однако замечу, что для шестеренки задается имя — butOptions, и объявляется новый класс — butGamePause. В конце функции мы запускаем метод rotate класса butGamePause:

```
function butGamePause::rotate(%this, %angle)
{
   %speed = 50;
   butOptions.RotateTo(%angle, %speed);
   %time = mAbs(%angle) / %speed * 1000;
   if (%this.Visible)
  schedule(%time, 0, "butOptions::rotate", %this,
-%angle);
}
```

Он принимает 1 параметр (кроме указателя): количество градусов, на которое нужно повернуть шестеренку. Скорость поворота устанавливается прямо внутри метода. Для осуществления вращения используется унаследованный от класса Sprite метод rotateTo, в качестве параметров принимающий угол поворота и скорость поворота:

```
butOptions.RotateTo(%angle, %speed);
```

Затем вычисляем время, требуемое для поворота шестеренки на указанный угол:

```
%time = mAbs(%angle) / %speed * 1000;
```

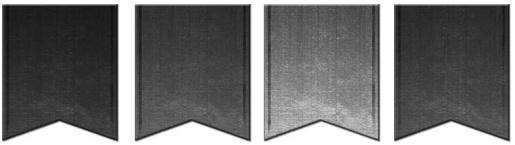

Рис. 10.11. Слоты для заклинаний

В заключение планируем через вычисленное время запуск этой же функции с передачей указанного числа градусов с обратным знаком, чтобы вращение запустилось в другую сторону:

```
schedule(%time, 0, "butOptions::rotate", %this,
-%angle);
```

Кроме того, шестеренка реагирует на нажатие, в результате чего отображается меню паузы:

```
function butGamePause::onTouchDown(%this, %touchID,
%worldPosition)
{
   Window::showPauseWin();
}
```

Это меню будет рассмотрено в соответствующем разделе.

Следующая функция — createMagicSlots, вызываемая из createGameGUI, создает слоты для магических заклинаний, вместе они представляют панельку:

Ассет для загрузки изображения так же прост:

```
<ImageAsset
   AssetName="slots"
   ImageFile="slots.png" />
```

Сама функция createMagicSlots весьма типична:

```
function createMagicSlots()
{
   %slot = new Sprite(slots)
   {
   Position = "28 27.5";
   Size = "40 10";
   SceneLayer = 31;
   Image = "ToyAssets:slots";
```

```
    };
    myScene.add(%slot);
}
```

Внутри функций createUpStub и createDownStub создаются полоски вверху и внизу игрового поля, соответственно, чисто визуальные компоненты:

```
function createUpStub()
{
    %stub = new Sprite(upStup)
    {
   Position = "0 33";
   Size = "100 9.5";
   SceneLayer = 31;
   Image = "ToyAssets:stub";
    };
    myScene.add(%stub);
}
function createDownStub()
{
    %stub = new Sprite(downStub)
    {
   Position = "0 -33";
   Size = "100 9.5";
   SceneLayer = 31;
   Image = "ToyAssets:stub";
    };
    downStub.setFlip(false, true);
    myScene.add(%stub);
}
```

Следующим действием выполняется цикл, в котором создаются иконки для магических заклинаний:

```
for (%i = 0; %i < getWordCount($ToyAssetsPics); %i++)
{
  MagicIco::createMagicIco(%i, getWord($ToyAssetsPics,
%i));
}
```

Рис. 10.12. Полоска

Магические заклинания будут подробно рассмотрены в следующем разделе. Отмечу лишь, что цикл выполняется от 0 до значения возвращенного функцией getWordCount, которая извлекает количество элементов в строке, указанной в параметре, разделенных пробелами имен картинок.

Под конец функции createGameGUI вызываются 2 конструктора двух классов: StepOrderWin::createStepOrderWin(); — окно очереди хода мы уже рассматривали, а Score::createMannaScore(); рассмотрим в разделе 15.

14 Магические заклинания

Магические заклинания являются главным дополнением MagicMancala, отличающие ее от других версий манкалы. Они позволяют разнообразить игру. С их помощью можно существенно продвинуться по игровому процессу вперед противника, для этого надо взвешенно применять заклинания, планируя ходы. Тем не менее использование заклинаний вносит элемент случайности. Заклинания нельзя применить к калахам. Зато их можно использовать как на своих, так и на вражеских лунках.

В MagicMancala существуют 4 вида заклинаний:

1. Магия Огня — уничтожает в лунке один кристалл.

2. Магия Заморозки — за один ход замораживает от 1 до 2 кристаллов в лунке, при этом в течение этого хода их нельзя переместить, выбрать или совершить другие операции.

3. Магия Молнии — уничтожает в определенной лунке от 0 до 2-х кристаллов.

4. Магия Телепортации — телепортирует из указанной лунки 1 кристалл в любую другую лунку.

Для использования магии нужна манна. В начале игры игрок получает 15 единиц манны, в то же время ИИ — только 5. Во время использования магии тратится манна. Как уже было сказано выше: каждое заклинание имеет свою цену. Счетчик манны отображается в верхней части

Рис. 10.13. Панель заклинаний — 4 вида магических заклинаний

игрового поля посередине окна. Каждый сделанный ход (щелчок по сокету) приносит игроку одну единицу манны. Также одну единицу манны приносит попадание кристалла в свой калах. Счетчик манны представлен отдельным классом, который мы рассмотрим в ближайшем разделе.

Использование заклинания начинается с выбора типа заклинания на панели в правом верхнем углу игрового поля (см. на изображении выше). Удерживая указатель, иконка с магией переносится на лунку, на которой мы хотим использовать заклинание. При отпускании указателя, волшебство произойдет над выбранной лункой. При этом над ней будет воспроизведена анимация. Для каждого заклинания заготовлена своя анимационная последовательность:

1. При использовании магии Огня над лункой произойдет взрыв в виде огненного шара, в результате чего из лунки исчезнет 1 кристалл.

2. Когда пользователь воспользуется заморозкой, над лункой произойдет раскол льда, от чего в лунке будут заморожены фишки.

3. При использовании Молнии внутри лунки пробежит электрический разряд, в результате чего из нее будут удалены фишки.

4. Когда используется заклинание Телепортации, из выбранной лунки исчезает один кристалл, остальные лунки при этом подсвечиваются голубым светом, это говорит о том, что игрок может выбрать любую из подсвеченных лунок. Совершая нажатие на любой из них, пропавший кристалл оказывается внутри нее. Оба события: исчезновение и появление кристалла сопровождаются анимационными эффектами. В первом случае происходит эффект «электронных волн от внешних границ внутрь лунки», во втором случае — «электронные волны изнутри к внешним границам лунки».

Когда количества манны не хватает для исполнения определенного заклинания, его иконка на панели становится «погасшей» — неактивной. Как только игрок набирает достаточное количество манны, иконка заклинания вновь становится активной.

Для реализации магии служат 2 класса: MagicIco и MagicEffect. Начнем по порядку с класса MagicIco. Объекты данного класса унаследова-

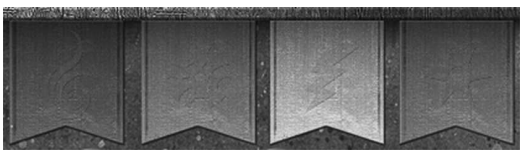

Рис. 10.14. Неактивные иконки заклинаний

ны от торковского класса Sprite, от чего имеют всю функциональность последнего. Создание объекта класса MagicIco осуществляется в методе createMagicIco, который, как обсуждалось в прошлом разделе, вызывается из функции createGameGUI:

```
MagicIco::createMagicIco(%i, getWord($ToyAssetsPics,
%i));
```

Метод получает порядковый номер и строку — ссылку на ассет, загружающий изображение. Разделенные пробелом строки хранятся в переменной $ToyAssetsPics, нужная из которых выбирается функцией getWord:

```
$ToyAssetsPics = "ToyAssets:FireBall ToyAssets:Freeze
ToyAssets:Lightning ToyAssets:Teleport";
```

Приведем код метода createMagicIco:

```
function MagicIco::createMagicIco(%num, %image)
{
   %x = 12.8;
   %x = %x + 10.2 * %num;
   %y = 28;

   %ico = new Sprite(MagicIco @ %num)
   {
  Position = %x SPC %y;
  Size = "8 8";
  SceneLayer = 30;
  class = "MagicIco";
  image = %image @ "Ico";
   };
   %ico.setUseInputEvents(true);
   %ico.type = %num;//числовой тип, соответствует
//номеру ассета из массива
   %ico.Price = getWord($ToyAssetsPricesLevel1, %num);

   myScene.add(%ico);
}
```

В начале метода вычисляются координаты для размещения на панели заклинаний в правом верхнем углу игрового поля в зависимости от номера заклинания (множитель %num). В следующем далее конструкторе создается спрайт для иконки магии, устанавливаются значения свойств: размер, положение, слой сцены, класс (в данном случае MagicIco), в ка-

честве изображения, к имеющемуся названию ассета добавляется «Ico». После конструктора устанавливаем возможность получения событий ввода плюс добавляем два пользовательских поля: type, Price. В первое записывается порядковый номер, который однозначно определяет тип заклинания. В поле Price записывается цена на определенный вид магии. Все цены через пробел записаны в списке $ToyAssetsPricesLevel1 в порядке соответствующем порядку заклинаний списка $ToyAssetsPics, другими словами, порядку расположения слотов магии на панели заклинаний. Цены извлекаются так же с помощью функции getWord.

После того как иконки созданы, они просто отображаются на экране в своих начальных позициях, пока игрок не нажмет указателем на одной из них. Так как объекты класса MagicIco реагируют на события ввода, они обрабатывают нажатие в обработчике события onTouchDown:

```
function MagicIco::onTouchDown(%this, %touchID,
%worldPosition)
{
   if ($semafor == 1 && !$objMD) {
   %image = %this.Image;
   %image = getSubStr(%image, strlen(%image)-4,
strlen(%image));
   if (%image $= "Dark")
        return;
   $magicPrice = %this.price;
        canvas.hideCursor();
        %ico = new Sprite()
        {
          class = %this.class;
        };
        %ico.Position = %this.Position;
        %ico.Size = %this.Size;
        %ico.SceneLayer = %this.SceneLayer - 1;
        %ico.image = %this.image;
        %ico.setUseInputEvents(true);
        %ico.type = %this.type;
        myScene.add(%ico);
        $objMD = %ico;
        $icoDown = 1;//был осуществлен щелчок на иконке
//магии
   }
}
```

Мы уже обсуждали параметры, получаемые обработчиком события, поэтому не будем повторяться на этом месте, сразу заглянем внутрь функции. В ее начале проверяется, чтобы $semafor был равен 1 (что

естественно, так как ходит игрок), и переменная $objMD была равна 0. Последняя хранит ссылку на перемещаемый объект. Если проверка пройдена, мы попадаем внутрь блока кода, где первым делом проверяется имя загруженного в спрайт (текущий объект) изображения: из имени берутся 4 последних символа, это делается в функции getSubStr, она получает: строку для парсинга, номер символа с которого начать их отбор и номер символа, на котором надо закончить выборку. Далее проверяется: чему равны эти 4 символа, если они равны последовательности «Dark», тогда мы покидаем функцию, так как в таком случае иконка заклинания является неактивной — у игрока не хватает манны для использования данной магии. Если вышеприведенное условие пройдено, тогда в переменную $magicPrice помещается цена на текущее заклинание. Происходит скрытие курсора. Затем создается новый объект класса MagicIco. Значения всех свойств для него копируются из того объекта, который подвергся нажатию указателем. Однако свойство SceneLayer устанавливается в значение на 1 меньше. Это делается для того, чтобы новый объект был на слой выше старого. В заключение метода ссылку на новый объект присваиваем глобальной переменной $objMD, при этом значение другой глобальной переменной $icoDown устанавливаем в 1. Это делается, чтобы показать, что произошло нажатие указателем на объекте.

Когда указатель удерживается на объекте и перемещается, управление из движка передается в обработчик onTouchDragged:

```
function MagicIco::onTouchDragged(%this, %touchID,
%worldPosition)
{
   if ($objMD && $icoDown < 2) {
       $objMD.setPosition(%worldPosition);
   }
}
```

Параметры те же самые, что в предыдущей функции. Внутри обработчика проверяется: $objMD не должна быть равна 0 и значение $icoDown должно быть меньше 2 (как мы помним: после нажатия на иконку, этой переменной присваивается 1). В случае совпадения условий с реальным положением дел, объект, ссылка на который хранится в $objMD, перемещается в координаты указателя. А так как этот обработчик вызывается непрерывно, объект $objMD всегда следует за указателем: пальцем или курсором мыши.

В момент, когда пользователь отпускает указатель, генерируется событие onTouchUp:

```
function MagicIco::onTouchUp(%this, %touchID,
%worldPosition)
{
   //при отпускание магии это событие происходит 1-ым
   if ($icoDown == 1) {
     $icoDown = 2;//был отпущен
     $imageAsset = %this.image;
     schedule($nextEventTime, 0, "deleteSprite");
   }
}
```

Его обработчик получает те же параметры, что предыдущие 2. Обратите внимание на первый комментарий в теле обработчика: из него следует, что во время отпускания указателя именно объект класса MagicIco первым получает реакцию на данное событие, это очень важный момент. Когда $icoDown равен 1, то эта переменная устанавливается в значение 2 (что означает состояние: указатель отпущен), глобальной переменной $imageAsset присваивается ссылка на текущее изображение объекта MagicIco, а через $nextEventTime миллисекунд планируется выполнение функции deleteSprite. Эта функция удаляет объект, на который указывает переменная $objMD:

```
function deleteSprite()
{
   if ($objMD) {
  $objMD.delete();
  $objMD = 0;
   }
}
```

Но почему мы вызываем функцию deleteSprite через планировщик — через некоторое время, а не напрямую? Дело в том, что мы не можем уничтожить объект из его собственного метода. Очевидно, что после завершения выполнения метода, объект должен существовать, а, если мы удалим его во время исполнения метода, тогда память, выделенная для объекта, будет разрушена, что, естественно, вызовет сбой в работе движка! Будьте осторожны при удалении объектов! Хотя Torque Script кажется безобидным скриптовым языком, неправильные конструкции на нем могут повредить всю игру!

В момент отпускания указателя происходит удаление перемещаемого значка заклинания, на чем заканчивается работа класса MagicIco. Сразу после него в работу вступает класс MagicEffect. Но откуда вызывается метод для создания объекта данного класса? Для разъяснения этого, нам надо вернуться в класс background. Выше я говорил, что реакцию на событие onTouchUp первым получают объекты класса MagicIco. В двумерном мире, построенном и живущем по правилам движка Torque 2D, объекты лежат друг на друге в разных слоях или на одном слое, как в стеке. Поэтому событие нажатия/отпускания указателя получают все объекты, находящиеся в одинаковых координатах. Следовательно, каждый объект получает оповещение о событии onTouchUp. Таким образом, фон (объект класса background) получает извещение вторым — после иконки заклинания (объект класса MagicIco), о чем говорит комментарий:

```
function background::onTouchUp(%this, %touchID,
%worldPosition)
{
   //2-ым
   if ($socket > -1)
  schedule($nextEventTime, 0, "checkToCreateEffect");
   canvas.showCursor();
}
```

В начале тела обработчика происходит проверка, чтобы значение переменной $socket было больше -1. На самом деле значение -1 — это особый случай, даже не инициализированная (пустая) переменная в Torque Script имеет значение "", что считается больше, чем -1, ибо это 0. В случае успешного прохождения проверки, на выполнение планируется функция checkToCreateEffect: schedule($nextEventTime, 0, "checkToCreateEffect");. И в любом случае курсор делается видимым (только на PC и Mac версиях): canvas.showCursor();.

Следующим по порядку объектом, получающим реакцию на событие onTouchUp, является сокет — объект класса Socket (хотя этот обработчик уже был рассмотрен в рамках класса Socket, не лишним будет взглянуть на него еще раз):

```
function Socket::onTouchUp(%this, %touchID,
%worldPosition)
{
   //3-им
   %name = %this.getName();
```

```
   %num = getTrailingNumber(%name);
   if ($icoDown == 2 && $canCreateMagicEffect) {
     $icoDown = 3;
     if (%num > -1)
        $socket = %this;//сохраняем активный сокет
     else $socket = -1;
   }
}
```

Здесь нам интересна строка: $socket = %this;//сохраняем активный сокет

В ней мы присваиваем указатель на активный сокет переменной $socket.

После выполнения данного — 3-го обработчика onTouchUp наступает время выполнения запланированной функции checkToCreateEffect:

```
function checkToCreateEffect()
{
   if ($semafor == 0 || $semafor == 1)
  %val = $fpMannaCount - $magicPrice; else
   if ($semafor == 2) {
  %val = $spMannaCount - $magicPrice;
  if (%val >= 0) {
       Score::editScore(%val);
       MagicEffect::createMagicEffect($socket,
$socket.getPosition(), false);
  }
   }

   if ($icoDown == 3 && $socket > 0 && %val >= 0) {
     Score::editScore(%val);
     MagicEffect::createMagicEffect($socket, $socket.
getPosition(), false);
     $socket.block = true;
     $socket = 0;
   }
   $icoDown = 0;

   return %val;//для AI
}
```

В зависимости от значения семафора — очереди хода вычисляется разность манны игрока или манны ИИ, где вычитаемым служит ранее установленный (в методах MagicIco::onTouchDown или AI::useMagic(), в зависимости от игрока или ИИ) $magicPrice. Если $semafor равен 2 (ход ИИ) и разность (%val) больше или равна 0, значит, количества манны

хватило на использование выбранного заклинания, и мы присваиваем счетчику манны это значение:

```
Score::editScore(%val);
```

Дальше создается магический эффект:

```
MagicEffect::createMagicEffect($socket, $socket.
getPosition(), false);
```

Это мы подробно рассмотрим немного позднее.

В другом случае, когда $semafor не равен 2, значит, ходит игрок, кроме значения разности происходят проверки: $icoDown должен быть равен 3 (состояние после нажатия) устанавливается в обработчике onTouchUp класса Socket, а $socket больше 0 — устанавливается там же. Если все условия сошлись, присваиваем разность — %val счетчику манны, создаем эффект магического заклинания, блокируем сокет: $socket.block = true; и обнуляем переменную $socket. После завершения условия обнуляем переменную $icoDown. Возвращаем разность. Последнее действие нужно только для ИИ, поскольку, ориентируясь на полученный результат, ИИ видит: удалось ли использовать выбранное заклинание и, следовательно, принимает решение о необходимости сделать ход — передвинуть фишки.

Вот мы и добрались до самого интересного — до главенствующей темы данного раздела — создания магического эффекта (метод расположен в файле MagicEffect.cs):

```
$canCreateMagicEffect = true;
function MagicEffect::createMagicEffect(%socket, %pos,
%delOrCre)
{
   if (!$canCreateMagicEffect) return;
   $canCreateMagicEffect = false;

   $execMagic = true;//исполнение магии

   %Round = "15 15";

   if ($imageAsset $= "ToyAssets:FireBallIco"
  || $imageAsset $= "ToyAssets:FreezeIco") {
  %w = getWord(%Round, 0);
  %w = %w / 100 * 30;
  %Round.x += %w;
  %Round.y += %w;
```

```
   }

   if ($imageAsset $= "ToyAssets:TeleportIco" &&
!%delOrCre) {
  %effect = "ToyAssets:TeleportReverse";
   } else
   %effect = getSubStr($imageAsset, 0,
strlen($imageAsset)-3);

   %anim = %effect @ "Anim";
   %fire = new Sprite(MagicEffect)
   {
  Position = %pos;
  Size = %Round;
  SceneLayer = 29;
  Animation = %anim;
   };
   myScene.add(%fire);
   $imageAsset = "";

   if (!%delOrCre) {
  %teleport = (%effect $= getWord($ToyAssetsPics, 3)
|| %effect $= "ToyAssets:TeleportReverse");//teleport
  if (!%teleport) %time = $fullAnimTime; else %time =
$fullAnimTime / 2;
  schedule(%time, 0, "MagicEffect::deleteMagicEffect",
%socket, %effect);
  if (!%teleport)
       if (!$endGame)//только если игра не
//закончена!!!
          schedule($fullAnimTime, 0, "StepOrderWin::
showStepOrderWin");
   } else
  schedule($fullAnimTime/2, 0, "MagicEffect::
immediatelyDelete", %socket);
}
```

Во-первых, разберемся, почему для эффектов мы используем только один объект (статические методы в Torque Script позволяют выполняться только одному объекту)? Можно подумать, что понадобиться много эффектов одновременно. Однако в MagicMancala геймплей протекает таким образом, что в каждый промежуток времени может существовать только один магический эффект. На это влияет: и строгая очередь хода, и последовательное выполнение ходов.

Метод createMagicEffect получает 3 параметра: сокет, в котором надо создать эффект, координаты этого сокета, булеву переменную, опреде-

ляющую способ удаления эффекта, от значения этого параметра зависит вызываемая функция, осуществляющая удаление. Ниже мы подробно разберем различия.

Первым же делом внутри метода проверяется глобальная переменная $canCreateMagicEffect, если она имеет нулевое значение, тогда выполнение метода прекращается. После проверки мы устанавливаем значение этой переменной в false, тем самым предотвращая попытки создания дополнительных эффектов, чего быть не должно.

Далее задаем значения переменным:

— $execMagic = true; — служит меткой, говорящей о том, что исполняется магия;

— %Round = «15 15»; — условный размер круга;

В условие проверяем значение переменной $imageAsset. В ней хранится имя ассета изображения для иконки магического заклинания, если оно равно «ToyAssets:FireBallIco» или «ToyAssets:FreezeIco», тогда из размера берем 1-е значение, делим его на 100, умножаем на 30 и прибавляем его к обоим компонентам размера.

Если переменная $imageAsset равна «ToyAssets:TeleportIco» вместе с тем, параметр %delOrCre равен 0, переменной %effect присваиваем имя ассета: «ToyAssets:TeleportReverse», в другом случае %effect получает название ассета без последних 3-х символов — с конца убирается «Ico». Далее в переменную %anim мы помещаем конкатенацию значения переменной %effect и последовательности символов «Anim» в результате чего получается имя ассета анимации.

К примеру, последовательность изображений для эффекта телепортации с ассетом для его загрузки и разделения на кадры плюс 2 анимационных ассета: прямого и обратного проигрывания выглядят следующим образом:

```
<ImageAsset
   AssetName="Teleport"
   ImageFile="Teleport.png"
   CellCountX="3"
   CellCountY="3"
   CellWidth="300"
   CellHeight="300" />
<AnimationAsset
   AssetName="TeleportAnim"
   Image="@asset=ToyAssets:Teleport"
   AnimationFrames="0 1 2 3 4 5 6"
   AnimationTime="0.56"
    AnimationCycle="false" />
```

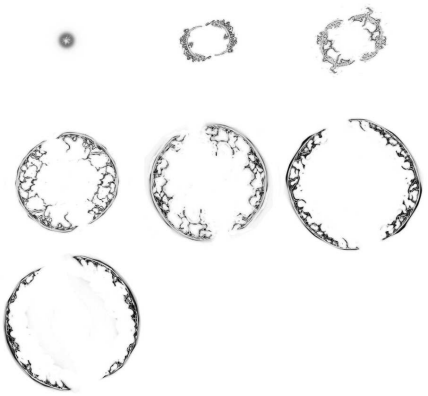

Рис. 10.15. Последовательность телепортации

```
<AnimationAsset
   AssetName="TeleportReverseAnim"
   Image="@asset=ToyAssets:Teleport"
   AnimationFrames="6 5 4 3 2 1 0"
   AnimationTime="0.56"
 AnimationCycle="false" />
```

Следующим действием мы создаем спрайт с именем MagicEffect, присваивая ему позицию (полученную в параметре), размер (вычисленный ранее), слой и анимацию — Animation, имя ассета, которое получено из имени изображения иконки заклинания. После этого добавляем спрайт в стек сцены и очищаем переменную $imageAsset.

Вновь обращаем внимание на параметр %delOrCre, если он нулевой, выполняем проверку, в которой проверяем значение переменной %effect: если она не равна имени одного из ассетов телепортации (ToyAssets:Teleport или ToyAssets:TeleportReverse), тогда в переменную %time помещается значение $fullAnimTime (время полной анимации эффекта равно 1120 миллисекунд), в обратном случае, когда выбрана анимация телепортации, для ее продолжительности отводится только половина данного количества миллисекунд, так как она вдвое короче. После вычисления времени мы планируем выполнение метода для удаления эффекта через полученное количество миллисекунд:

```
schedule(%time, 0, "MagicEffect::deleteMagicEffect",
%socket, %effect);
```

Вдобавок если эффект — не относится к телепортации и, если игра до сих пор не завершена, мы планируем после проигрывания анимации (через $fullAnimTime миллисекунд) отображение окна очередности хода (выполнение метода StepOrderWin::showStepOrderWin).

Если же %delOrCre равна true, тогда через $fullAnimTime/2 миллисекунд планируем непосредственное удаление магического эффекта:

```
schedule($fullAnimTime/2, 0, "MagicEffect::immediately
Delete");
```

На этом метод createMagicEffect завершен. Но у нас остались 2 метода для удаления эффекта: deleteMagicEffect и immediatelyDelete. Их различие заключается в том, что 1-й прежде, чем удалить эффект проверяет его название и вызывает соответствующий метод для выполнения эффекта, тогда, как 2-ой сразу удаляет объект эффекта. Рассмотрим их вплотную.

Метод deleteMagicEffect получает указатели на сокет и сам эффект:

```
function MagicEffect::deleteMagicEffect(%socket,
%effect)
{
   if (MagicEffect !$= "") {
   switch$(%effect) {
        case getWord($ToyAssetsPics, 0) : MagicEffect::f
ireEffect(%socket);
        case getWord($ToyAssetsPics, 1) : MagicEffect::
freezeEffect(%socket);
        case getWord($ToyAssetsPics, 2) : MagicEffect::
lightningEffect(%socket);
        case getWord($ToyAssetsPics, 3) : MagicEffect::
teleportEffect(%socket);
        case "ToyAssets:TeleportReverse": MagicEffect::
teleportEffect(%socket);
   }
   MagicEffect::immediatelyDelete();
   %gameOver = winGame(getInverseSide(checkToLose()));
    }
}
```

Внутри тела метода происходит проверка: если объект MagicEffect инициализирован, тогда в операторе ветвления switch$, который в Torque Script может сверять строки, сравнивается значение параметра %effect со строками из последовательности строк $ToyAssetsPics. При совпадении вызывается метод исполнения соответствующего эффекта.

После оператора ветвления происходит непосредственное удаление эффекта с помощью метода immediatelyDelete. Затем знакомым для нас образом выполняется проверка на завершение игры:

```
%gameOver = winGame(getInverseSide(checkToLose()));
```

Второй метод — immediatelyDelete не получает параметров:

```
function MagicEffect::immediatelyDelete()
{
   MagicEffect.delete();
   $canCreateMagicEffect = true;
}
```

В его задачи входят только 2 действия: удаление магического эффекта и присвоения глобальной переменной $canCreateMagicEffect значения true, тем самым, разрешая игре, создать очередной эффект, поскольку предыдущий удален.

Метод MagicEffect::deleteMagicEffect, как сказано выше: планируется на выполнение через время, необходимое для завершения проигрывания анимационного эффекта, после чего в лунке должен произойти результат использования заклинания. Следовательно, данный метод в зависимости от используемой магии вызывает соответствующий метод.

Если было использовано заклинание «FireBall», вызывается метод завершения эффекта MagicEffect::fireEffect. Все методы завершения эффектов принимают единственный параметр — указатель на сокет. Под указателем имеется в виду индекс объекта, позволяющий однозначно идентифицировать игровой объект.

Итак, fireEffect имеет следующий вид:

```
function MagicEffect::fireEffect(%socket)
{
   %num = getTrailingNumber(%socket.getName());
   %count = mySock.getObject(%num).getCount();
   if (%count > 0) {
  %obj = mySock.getObject(%num).getObject(0);
  mySock.getObject(%num).remove(%obj);
  %obj.delete();
  TextLab::UpdateCounter(%num);
   } else
   if ($semafor == 0 || $semafor == 1)
  %socket.magic = "fire1";
}
```

В локальной переменной %num запоминается оконечное число из имени сокета, которое, выступая индексом контейнеров, позволяет обратиться к одному из последних и получить количество, содержащихся в нем фишек, что происходит в следующей строчке. Если это количество больше 0, выполняется блок кода, в котором из контейнера берется самая первая фишка, сначала она удаляется из контейнера, затем происходит ее уничтожение. Потом происходит пересчет фишек в сокете и вывод в счетчике рядом с соответствующей лункой корректного числа. Если же лунка пуста, и ход осуществлен игроком (семафор равен 0 или 1), тогда лунка приобретает магию Огня. Уничтожение одной фишки есть результат магии Огня.

Когда используется заклинание заморозки — Freeze, вызывается метод freezeEffect:

```
function MagicEffect::freezeEffect(%socket)
{
   %freeze = getRandom(1, 2);
   %num = getTrailingNumber(%socket.getName());
   %count = mySock.getObject(%num).getCount();
   if (%count > 0) {
  %i = 0;
  while (%i < %freeze) {
       %obj = mySock.getObject(%num).getObject(%i);
       %obj.Image = "ToyAssets:ice";
       if ($semafor == 2)
          FreezeCrystals2.add(%obj); else
       FreezeCrystals1.add(%obj);
       mySock.getObject(%num).remove(%obj);
       if ($semafor == 0 || $semafor == 1)
          $freezeSocket1 = %num; else
       if ($semafor == 2)
          $freezeSocket2 = %num;
       %i++;
  }
  TextLab::UpdateCounter(%num);
   } else %socket.magic = "freeze";
}
```

В переменную %freeze помещается случайно выбранное число: 1 или 2, оно означает количество замороженных фишек. Следующим действием получаем индекс контейнера и количество содержащихся в нем фишек. Если это количество больше 0, тогда выполняем блок кода, в котором, выбрав из контейнера одну фишку, загружаем для нее текстуру из ассета ToyAssets:ice; далее по значению семафора выбираем в какой

контейнер поместить эту фишку: FreezeCrystals1 или FreezeCrystals2; после чего удаляем ее из начального контейнера; в зависимости от значения семафора в переменной $freezeSocket(1 или 2) сохраняем индекс контейнера, в котором была заморожена фишка; выполняем эту последовательность действий, пока количество замороженных фишек не будет равно значению переменной %freeze. Наконец, обновляем счетчик фишек соответствующей лунки. В том случае, когда заклинание использовано на пустом сокете (где %count равен 0), для этого сокета устанавливаем магию заморозки: %socket.magic = «freeze»;

Обратная процедура заморозки — разморозка кристаллов — MagicEffect::unFreezeCrystals() вызывается в момент отображения окна порядка хода — StepOrderWin, в том случае, если одна из переменных: $freezeSocket1 или $freezeSocket2 имеет значение больше -1. Функция MagicEffect::unFreezeCrystals выглядит так:

```
function MagicEffect::unFreezeCrystals()
{
   if ($semafor == 0 || $semafor == 1) {
     %cont = FreezeCrystals2;
  while(%cont.getCount()) {
       %obj = %cont.getObject(0);
       %obj.Image = "ToyAssets:" @ getCrystalAsset();
       mySock.getObject($freezeSocket2).add(%obj);
       %cont.remove(%obj);
  }
  TextLab::UpdateCounter($freezeSocket2);
  $freezeSocket2 = -1;
   } else
   if ($semafor == 2) {
       %cont = FreezeCrystals1;
  while(%cont.getCount()) {
       %obj = %cont.getObject(0);
       %obj.Image = "ToyAssets:" @ getCrystalAsset();
       mySock.getObject($freezeSocket1).add(%obj);
       %cont.remove(%obj);
  }
  TextLab::UpdateCounter($freezeSocket1);
  $freezeSocket1 = -1;
   }
}
```

Если $semafor равен 0 или 1, тогда мы берем контейнер FreezeCrystals2, поскольку, очевидно, что предыдущий ход заморозки сделан оппонентом. Перебираем в этом контейнере все фишки, загружаем для них слу-

чайные текстуры (выбранные с помощью функции getCrystalAsset), добавляем их в контейнер с индексом $freezeSocket2, а из FreezeCrystals2 — удаляем. Под конец обновляем счетчик фишек в лунке. С другой стороны, если $semafor равен 2, выполняем похожую последовательность операций, заменив контейнер на FreezeCrystals1 и индексную переменную на $freezeSocket1.

В случае использования магии молнии, из deleteMagicEffect вызывается функция lightningEffect:

```
function MagicEffect::lightningEffect(%socket)
{
   %num = getTrailingNumber(%socket.getName());
   %count = mySock.getObject(%num).getCount();
   %del = getRandom(0, 2);
   if (%count > 0) {
  %i = 0;
  while (mySock.getObject(%num).getCount() && %i <
%del) {
       %obj = mySock.getObject(%num).getObject(0);
       mySock.getObject(%num).remove(%obj);
       %obj.delete();
       %i++;
  }
  TextLab::UpdateCounter(%num);
   } else %socket.magic = "lightning";
}
```

Как в предыдущих методах, здесь мы получаем индекс текущего контейнера, из которого — количество фишек. В переменную %del кладем случайно выбранное между нулем и двойкой число удаляемых фишек. Продолжая ход выполнения, в случае, если число фишек больше 0, мы удаляем %del фишек из контейнера. Потом обновляем счетчик кристаллов для соответствия действительности. Если контейнер лунки изначально пуст, устанавливаем для сокета магию молнии.

Когда свое выполнение заканчивает 1-й этап заклинания телепортации, это происходит в момент, когда игрок перетаскивает на лунку значок заклинания, и над ней проигрывается волновая анимация, вызывается функция teleportEffect:

```
function MagicEffect::teleportEffect(%socket)
{
   $chooseSocket = true;
   new SimSet(teleportCrys);
   %socket.lightSockets(%socket, true);
```

```
    %num = getTrailingNumber(%socket.getName());
    %count = mySock.getObject(%num).getCount();
    if (%count > 0) {
   %obj = mySock.getObject(%num).getObject(0);
   %obj.Visible = false;
   teleportCrys.add(%obj);
   mySock.getObject(%num).remove(%obj);
   TextLab::UpdateCounter(%num);
    } else %socket.magic = "teleport";
}
```

Глобальная переменная $chooseSocket сразу устанавливается в true, чтобы сделать доступным режим выбора. Затем создается набор объектов (SimSet) teleportCrys, он предназначен для хранения фишек, ждущих своей телепортации — перемещения. Затем мы вызываем метод lightSockets класса Socket, этот метод подсвечивает другие лунки. Он был рассмотрен во время обзора класса Socket. Следующими действиями мы получаем индекс текущего сокета и количество кристаллов из сопряженного с ним контейнера. Если это количество превышает 0, тогда мы берем самый 1-й кристалл, делаем его невидимым, добавляем в созданный набор teleportCrys, удаляем из контейнера сокета, обновляем счетчик. Если же сокет пуст, присваиваем ему магию телепортации.

На этом экскурс в класс MagicEffect завершен. Мы увидели создание магических эффектов и последствия их исполнения.

15 Счетчик манны

Счетчик манны представлен отдельным классом Score. В игре он отображается в центре в верхней части игрового поля.

Код класса очень небольшой. Создание текстового счетчика происходит в методе createMannaScore класса Score.

В главе 6 «Разработка аркадной игры Asteroids» для реализации текстовых надписей мы применяли объекты класса ImageFont. Это был единственный способ для создания текстовых надписей в движке Torque 2D до стабильной версии 3.2 включительно. В настоящее время из версии движка 3.3 удален класс ImageFont. На его место встал класс TextSprite — оптимизированный и предлагающий более широкие возможности способ организации текстовых надписей. Так как мы с вами договорились ис-

Рис. 10.16. Счетчик манны

пользовать самую новую версию движка Torque 2D для экспериментирования и создания наших игр, при разработке MagicMancala мы будем использовать этот класс; я уже перенес нашу игру на последнюю версию движка.

Метод createMannaScore вызывается из функции createGameGUI и выглядит следующим образом:

```
function Score::createMannaScore()
{
   %object = new TextSprite(mannaScore);
   %object.SceneLayer = 30;
   %object.Font = "ToyAssets:ArialFont";
   %object.Position = "0 31";
   %object.FontSize = "3 3";
   %object.Size = "4 3";
      %object.FontPadding = 0;
   %object.TextAlignment = "Right";
   %object.BlendColor = "1 1 1 1";
   %object.Text = Score::fullScore($fpMannaCount);
   %object.setBodyType(static);
   myScene.add(%object);
}
```

Внутри метода создается объект класса TextSprite с именем mannaScore. Как мы выше отметили, TextSprite представляет надпись — способ для вывода текста. Как и ImageFont, TextSprite в качестве основы для вывода текста использует растровую карту символов, другими словами, битовое изображение, включающее символы. Но по сравнению с ImageFont у нового класса есть отличительные особенности.

Для загрузки карты символов TextSprite использует объект движка FontAsset, так же внесенный в последнюю версию движка — 3.3. FontAsset представляет собой класс, унаследованный от AssetBase, как и все ассеты, он позволяет загружать и создавать ресурсы во время выполнения игры. FontAsset дополнительно к унаследованным имеет только одно поле — FontFile. В качестве значения для этого свойства указывается имя файла шрифта с расширением fnt. Описание ассета на TAML имеет следующий вид:

```
<FontAsset
   AssetName="ArialFont"
   FontFile="Arial.fnt" />
```

Для создания растрового шрифта, в частности, файла с расширением fnt, используется свободная программа Angle Code's Bitmap Font Generator (http://www.angelcode.com/products/bmfont/). Она уже неоднократно упоминалась в книге, например, с помощью нее создана карта символов в главе 6. Однако если раньше использовалась только сгенерированное программой изображение выбранного шрифта, то сейчас, так же используется подготавливаемое этой программой описание созданной карты символов. Таким образом, ассет вначале загружает только описание, в котором содержатся ссылки на изображения, включающие карты символов определенного шрифта, параметры шрифта, координаты и размеры символов в карте и другое.

TextSprite включает много свойств: как унаследованных от SceneObject, так и принципиально новых, относящихся к шрифтам. Из общих свойств для вновь созданного объекта устанавливаются: позиция, экранный слой, тип тела для взаимодействия (в данном случае: static); из специальных: Font — ссылка на ассет (FontAsset) — описание карты символов, FontSize (указывается по двум осям) — размер шрифта, Size — размеры вмещающего надпись прямоугольника (так, например, если он будет меньше надписи, то она будет усечена), FontPadding (заполнение) — расстояние между символами, TextAlignment — выравнивание текста, BlendColor — цвет текста (в данном случае с помощью «1 1 1 1» задается белый непрозрачный цвет) и, собственно, свойство Text принимает значение для вывода, в данном случае ему присваивается результат функции fullScore. Таким образом, с помощью этих свойств

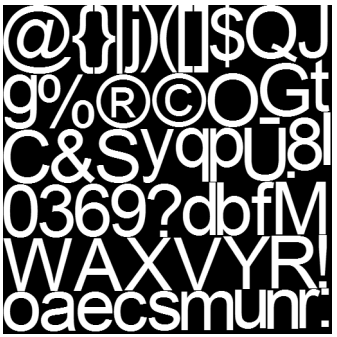

Рис. 10.17. Битовые карты символов, используемые в нашей игре

Рис. 10.18. Битовые карты символов, используемые в нашей игре

можно переопределять значения параметров шрифта, заданные в fnt файле.

Функция fullScore принимает параметр — число, если оно состоит из одной цифры, то дополняется слева 0, если переданное число двузначное, тогда — возвращается без изменения:

```
function Score::fullScore(%str)
{
   if (strlen(%str) == 1)
   return "0" @ %str;
   else
   return %str;
}
```

Метод scorePlus не принимает и не возвращает параметров, проводя все операции внутри:

```
function Score::scorePlus()
{
   if ($semafor == 0 || $semafor == 1) {
   $fpMannaCount++;
   mannaScore.setText(Score::fullScore($fpMannaCount));
   } else
   if ($semafor == 2) {
   $spMannaCount++;
   }
   MagicIco::chooseDarkSide();
}
```

В начале своего выполнения он проверяет $semafor: если ход совершил игрок (данный метод всегда вызывается в результате хода), тогда на 1 увеличивается манна игрока — значение переменной $fpMannaCount, а для надписи mannaScore с помощью метода setText задается новая надпись, результат функции fullScore:

```
mannaScore.setText(Score::fullScore($fpMannaCount));
```

Если ход сделал ИИ, тогда увеличивается его манна: $spMannaCount++;

В конце функции вызывается метод chooseDarkSide класса MagicIco. Он ничего не принимает и не возвращает:

```
function MagicIco::chooseDarkSide()
{
   for (%i = 0; %i < 4; %i++) {
   %obj = (MagicIco @ %i);
```

```
   %image = %obj.image;
   %dark = getSubStr(%image, strlen(%image)-4,
strlen(%image));
   %price = %obj.price;
   if ($fpMannaCount >= %price) {
        if (%dark $= "Dark")
          %obj.image = getSubStr(%image, 0,
strlen(%image)-4);
   }//if
   else {
        if (%dark !$= "Dark")
          %obj.image = %image @ "Dark";
   }//else
    }
}
```

В цикле for он перебирает все 4 иконки заклинания, сохраняя для каждой имя изображения, затем с помощью функции getSubStr из имени происходит выборка последних 4-х символов. Если манна игрока равна или превышает цену на магию, а последовательность символов равна «Dark», то для спрайта загружается изображение без последних 4-х символов. В случае, когда у игрока меньше манны, чем цена на магию, на спрайт иконки загружается изображение со словом Dark. Среди арта MagicMancala есть изображения для иконок заклинания вида «FireBall» и «FireBallDark». Первые отображаются, когда у игрока достаточно манны для использования определенного заклинания, вторые — затемненные, когда у пользователя не хватает манны. С учетом того, что этот метод вызывается после каждого хода игрока, иконки всегда находятся в актуальном виде.

Последний метод в классе Score — editScore принимает один параметр — новое значение количества манны, таким образом, в зависимости от семафора, в методе обновляется манна игрока или ИИ, кроме того, в 1-м случае обновляется счетчик манны. Плюс в случае хода игрока в конце метода вызывается функция chooseDarkSide.

Класс Score весьма небольшой, но очень полезный и информативный для игрока.

16 Текстовые надписи

Класс TextLab предназначен для создания текстовых меток. Он похож на предыдущий класс Score, являясь упрощенным вариантом последнего, однако работает по другой логике. Он позволяет создавать текстовые счетчики для количества фишек в лунках и калахах. Этот

класс состоит всего из 2-х методов. 1-й из них предназначен для создания счетчика:

```
function TextLab::createCounter(%num)
{
   if (%num < 5) {
   %y = -20.5;
   %x = -28.8 + %num * 14.5;
   } else
   if (%num == 5) {
   %y = -20.5;
   %x = -28.8 + %num * 14.4;
   } else
   if (%num < 11) {
   %y = 20.7;
   %x = 29.2 - (%num - 6) * 14.5;
   } else
   if (%num == 11) {
   %y = 20.7;
   %x = 29.2 - (%num - 6) * 14.4;
   }

   %counter = new TextSprite(textLab @ %num);
      %counter.SceneLayer = 30;
      %counter.Font = "ToyAssets:ArialFont";
      %counter.Position = %x SPC %y;
      %counter.FontSize = "2 2";
      %counter.Size = "2 2";
      %counter.BlendColor = "1 1 1 1";
      %counter.FontPadding = 0;
      %counter.TextAlignment = "Center";
      %counter.setBodyType(static);
   myScene.add(%counter);
}
```

Он вызывается из функции StartGame класса MainMenu следующим образом:

```
for (%i = 0; %i < 12; %i++)
   TextLab::createCounter(%i);
```

То есть из цикла по всем лункам, включая калахи.

Вернемся однако к методу createCounter. В качестве параметра он получает номер, который конкатенируется к последовательности символов textLab, из чего в итоге получается уникальное имя. Кроме того, опираясь на номер, функция вычисляет позицию для счетчика. Сам счетчик

представлен объектом класса TextSprite. Как говорилось выше: это новейший способ в движке Torque 2D вывести текстовую надпись на экран. В параметрах мы указываем номер экранного слоя, физический тип, шрифт (ссылку на описание растрового шрифта), цвет шрифта, размер прямоугольника — подложки, позицию, размер шрифта, расстояние между символами, выравнивание текста. На этом создание счетчика завершается.

Вторая функция класса TextLab служит для обновления счетчика. Она вызывается из многих мест исходного кода игры. В качестве параметра она принимает индекс сокета, в котором надо пересчитать количество кристаллов. Кроме того, этот индекс служит для определения имени того счетчика, который надо обновить:

```
function TextLab::UpdateCounter(%numSock)
{
   if (%numSock > -1)
      (textLab @ %numSock).
setText(Score::fullScore(mySock.getObject(%numSock).
getCount()));
}
```

После конкатенации и получения имени счетчика вызывается метод setText класса TextSprite, который устанавливает для надписи новый текст. В качестве параметра ему передается возвращенное значение функции Score::fullScore, которая возвращает двузначное число с дополнительным нулем слева, когда оно однозначное. На входе функция получает количество фишек в лунке с индексом равному параметру.

17 Главное меню

Главное меню показывается на экране сразу после запуска игры. Но мы его оставили на последнюю очередь, поскольку, по сути, оно не влияет на игровой процесс, а он является самым важным при разработке любой игры. Поэтому разные меню и дополнительные — декоративные окна делаются в последний момент разработки игрового приложения.

Функция создание главного меню — MainMenu::create вызывается из метода MagicMancala::create файла main.cs, с которого, собственно, начинается выполнение игры. В функции создания главного меню создаются 4 спрайта: фон (MMback), передний слой (MMfront), спрайт для слова «Magic» и спрайт для слова «Loading». В конце функции вызывается метод particlesDot для создания еще одного спрайта для украшения:

```
function MainMenu::create()
{
    %back = new Sprite(MMback)
    {
   class = "MainMenu";
   BodyType = static;
   Position = "0 0";
   Size = $cameraSizeWidth SPC $cameraSizeHeight-10;
   SceneLayer = 31;
   Image = "ToyAssets:MM_back";
    };
    myScene.add(%back);

    %back = new Sprite(MMfront)
    {
   class = "MainMenu";
   BodyType = static;
   Position = "0 0";
   Size = $cameraSizeWidth SPC $cameraSizeHeight-10;
   SceneLayer = 30;
   Image = "ToyAssets:MM_front";
    };
    myScene.add(%back);

    %magic = new Sprite(wordMagic)
    {
   BodyType = static;
   Position = "18 23.5";
   Size = "22 7";
   SceneLayer = 30;
   Image = "ToyAssets:wordMagic";
    };
    myScene.add(%magic);

    %load = new Sprite(wordLoad)
    {
   class = "MainMenu";
   BodyType = static;
   Position = "0 -25";
   Size = "20 5";
   SceneLayer = 30;
   Image = "ToyAssets:wordLoading";
   Visible = false;
    };
    myScene.add(%load);

    MainMenu::particlesDot();
}
```

Рис. 10.19. Подложка — фон

Рис. 10.20. Передний слой

Рис. 10.21. Слово Magic

Рис. 10.22. Слово Loading

Все ассеты для загрузки изображений очень простые и не требуют дополнительных комментариев.

Обратите внимание: все спрайты являются статическими, то есть они не могут двигаться — все по плану!

Функция particlesDot имеет такой вид:

```
function MainMenu::particlesDot()
{
   %part = new Sprite(particlesDot)
   {
  BodyType = static;
  Position = "24.3 27";
  Size = "4 4";
  SceneLayer = 30;
  Image = "ToyAssets:particles";
   };
   myScene.add(%part);

   MainMenu::createModes();
}
```

Рис. 10.23. Декорация

В ней создается статический спрайт particlesDot.

В конце этой функции вызывается следующая функция — createModes. Она служит для создания спрайтов — режимов игры. Пока в MagicMancala только 1 режим, поэтому создается 1 спрайт, заметьте: он динамический, поскольку, чтобы выделяться на фоне других объектов, показывая, что его можно выбрать, он должен как-то перемещаться, изображая свою активность:

```
function MainMenu::createModes()
{
   %mode = new Sprite(mode1playerBut)
   {
  class = "Mode1Player";
  BodyType = dynamic;
  Position = "0 -8.5";
  Size = "18 6";
  SceneLayer = 31;
  Image = "ToyAssets:mode1player";
  UseInputEvents = true;
   };
   myScene.add(%mode);

   mode1playerBut::Up();
}
```

Для спрайта объявляется новый — отдельный класс:

```
class = "Mode1Player";
```

чтобы для него можно было написать обработчик события нажатия. А, чтобы спрайт имел возможность получать сообщения о пользовательском вводе, надо установить свойство UseInputEvents в значение true.

В конце функции createModes для созданной кнопки (mode1playerBut) вызывается метод Up, не принимающий никаких параметров. Этот метод занимается тем, что находит позицию и с константной скоростью поднимает надпись вверх на определенное (вычисленное) расстояние. Вместе с тем он высчитывает время, за которое спрайт преодолеет эту

Рис. 10.24. Режим игры: 1 игрок

дистанцию, и через этот временной промежуток планирует вызов метода Down класса кнопки — Mode1Player:

```
function Mode1Player::Up()
{
   %pos = mode1playerBut.Position.X SPC
mode1playerBut.Position.Y + $magicDist;
   mode1playerBut.moveTo(%pos, $magicSpeed);
   %time = $magicDist / $magicSpeed * 1000;
   if (mode1playerBut.Visible)
  schedule(%time, 0, "mode1playerBut::Down");
}
```

Метод Down, в свою очередь, делает похожее действие, разница лишь в том, что он находит позицию внизу, опускает туда спрайт с надписью, вычисляя время, необходимое на прохождение этого пути, и планирует через него вызов метода Up. Таким образом, получается вечный двигатель, который то опускает, то поднимает спрайт с надписью, от чего получается анимационный эффект, выделяющий надпись среди других статических объектов:

```
function Mode1Player::Down()
{
   %pos = mode1playerBut.Position.X SPC
mode1playerBut.Position.Y - $magicDist;
   mode1playerBut.moveTo(%pos, $magicSpeed);
   %time = $magicDist / $magicSpeed * 1000;
   if (mode1playerBut.Visible)
  schedule(%time, 0, "mode1playerBut::Up");
}
```

Как было сказано выше: этот спрайт — надпись является кнопкой, через нажатие на которую должна запуститься одиночная игра: игрок против ИИ. Поэтому нам надо обработать событие нажатия указателем на этом спрайте:

```
function Mode1Player::onTouchDown(%this, %touchID,
%worldPosition)
{
  if (%this.Position.X != 0) return;
  wordLoad.Visible = true;
  schedule($nextEventTime, 0,
"MainMenu::HideAndShow");
}
```

В обработчике onTouchDown надпись «Loading» делается видимой, а через $nextEventTime миллисекунд на выполнение планируется метод HideAndShow класса MainMenu.

В методе HideAndShow все ранее созданные объекты делаются невидимыми:

```
function MainMenu::HideAndShow()
{
   mode1playerBut.Visible = false;
   mode1playerBut.Active = false;
   particlesDot.Visible = false;
   wordMagic.Visible = false;
   MMback.Visible = false;
   MMfront.Visible = false;
   wordLoad.Visible = false;

   MainMenu::StartGame();
}
```

А в конце вызывается метод StartGame. Внутри него вызываются функции и конструкторы для создания игровых объектов:

```
function MainMenu::StartGame()
{
```

Рис. 10.25. Главное меню

```
   createBackground();
   createBoard();
   for (%i = 0; %i < 12; %i++)
  TextLab::createCounter(%i);
   createSockets();
   createGameGUI();
   Window::createWinWin();
   Window::createLoseWin();
   Window::createPauseWin();
}
```

К этому моменту мы рассмотрели создание почти всех игровых объектов, остался только класс Window.

18 Дополнительные окна

Внутри игровые окна представлены классом Sprite и так называются, просто потому, что имеют схожий с ними вид. Как мы увидели выше: все окна создаются при загрузке игры, но делаются невидимыми, чтобы в момент, когда их надо показать, не тратить время на их создание, а просто вывести на экран.

Первое по списку: окно победы, как и остальные окна, относится к классу Window. Функция его создания:

```
function Window::createWinWin()//создать окно победы
{
   %fade = new Sprite(fadeWin)
   {
        Position = "0 0";
        Size = $cameraSizeWidth SPC $cameraSizeHeight;
        SceneLayer = 29;
        Image = "ToyAssets:fade";
        Visible = false;
        BodyType = static;
        UseInputEvents = false;
   };
   myScene.add(%fade);

   %win = new Sprite(winWin)
   {
        Position = "0 0";
        Size = "50 40";
        SceneLayer = 28;
        Image = "ToyAssets:winWin";
        Visible = false;
        BodyType = static;
        UseInputEvents = false;
```

```
   };
   myScene.add(%win);

   %but = new Sprite(winOkBut)
   {
        Position = "1.3 -13.4";
        Size = "10 8";
        SceneLayer = 28;
        Image = "ToyAssets:winOkbut";
        Visible = false;
        BodyType = static;
        UseInputEvents = true;
        class = "butNewGame";
   };
   myScene.add(%but);

    //звезды
   %star = new Sprite(star1)
   {
        Position = "-13.9 -2.4";
        Size = "12 11.5";
        SceneLayer = 28;
        Image = "ToyAssets:star";
        Visible = false;
        BodyType = static;
   };
   myScene.add(%star);
   %star = new Sprite(star2)
   {
        Position = "0.15 1.5";
        Size = "12 11.5";
        SceneLayer = 28;
        Image = "ToyAssets:star";
        Visible = false;
        BodyType = static;
   };
   myScene.add(%star);

   %star = new Sprite(star3)
   {
        Position = "14.6 -2.4";
        Size = "12 11.5";
        SceneLayer = 28;
        Image = "ToyAssets:star";
        Visible = false;
        BodyType = static;
   };
   myScene.add(%star);
}
```

Рис. 10.26. Окно Победы

Рис. 10.27. Чистая победа

Рис. 10.28. Кнопка OK

В этой функции создается 6 спрайтов. Когда отображается одно из 3-х дополнительных окон, игровое поле затеняется — покрывается мглой. Эту роль выполняет дополнительный спрайт — fadeWin с соответствующими окну размерами. На этот спрайт загружается полупрозрачная текстура fade.png. Второй создаваемый здесь спрайт — это winWin, имеющий размеры 50 на 40 и находящийся на слой выше. В качестве текстуры на него загружается изображение winWin.png:

Третий создаваемый спрайт служит кнопкой — winOkBut. На него загружается одноименная текстура.

Кнопка является объектом класса butNewGame, чтобы можно было в рамках последнего написать обработчик нажатия:

```
function butNewGame::onTouchDown(%this, %touchID,
%worldPosition)
{
    newGame();
    $fpMannaCount = $beginMannaCount;
    Score::editScore($beginMannaCount);
    $spMannaCount = 5;
    //обнуление индикаторов лунок
    for (%i = 0; %i < 5; %i++) {
   TextLab::UpdateCounter(%i);
    }
    for (%i = 6; %i < 11; %i++) {
   TextLab::UpdateCounter(%i);
    }
    TextLab::UpdateCounter(5);
    TextLab::UpdateCounter(11);
    //замороженные сокеты
    $freezeSocket1 = -1;
    $freezeSocket2 = -1;

    $semafor = 1;
}
```

В первую очередь внутри обработчика вызывается функция newGame, мы рассмотрим ее поздней. После нее выполняются действия для начала новой игры: значение манны устанавливается в начальное состояние, счетчики кристаллов для лунок обнуляются, семафор устанавливается в единицу.

Функция newGame выполняет еще более тщательную подготовку к началу новой игры:

```
function newGame()
{
   deleteCrystals();

   $freezeSocket1 = -1;
   $freezeSocket2 = -1;

   $endGame = false;//начинаем игру заново
   $startGame = true;//перезапускаем игру
   //флаг, показывающий паузу
   $pause = false;
   //флаг, показывающий выполнение действия
   $action = 0;

   $star1_take5 = false;
   $star2_up2 = false;
   $star3_up3 = false;

   %Y = -10;
   for (%i = 0; %i < 5; %i++) {
   %X = -28.25 + %i * $distBetweenSockets;
      createCrystals(%i, %X, %Y, "right");
   }
   new SimSet(sock5);
   mySock.add(sock5);

   %Y = 10.5;
   for (%i = 6; %i < 11; %i++) {
   %X = (-28.25 + 10 * $distBetweenSockets) -
$distBetweenSockets * %i;
      createCrystals(%i, %X, %Y, "left");
   }
   new SimSet(sock11);
   mySock.add(sock11);

        fadeWin.Visible = false;
        winWin.Visible = false;
        star1.Visible = false;
        star2.Visible = false;
```

```
        star3.Visible = false;
        winOkBut.Visible = false;
        loseWin.Visible = false;
        loseOkBut.Visible = false;
        HidePauseWin();

   //$activeMenu = false;//меню становится неактивно

   $semafor = 1;
   StepOrderWin::createStepOrderWin();
}
```

Она возвращает к первоначальному значению глобальные переменные. Все объекты уже созданы: фон, стол, окна, магия и заклинания, надписи и счетчики, сокеты, и их пересоздавать не нужно, тем не менее надо пересоздать кристаллы. Удаление кристаллов осуществляется в функции deleteCrystals, которая вызывается из функции newGame, ее листинг рассмотрен в разделе 3 Кристаллы. Затем происходит создание кристаллов и недостающих лунок; после чего невидимыми делаются объекты дополнительных окон; семафор устанавливается в единицу; происходит создание окна очередности хода — StepOrderWin.

Вернемся к окну Победы. Здесь у нас есть функция showWinWin. Она делает видимыми созданные в createWinWin спрайты, однако не все. Таким образом, показ звезд зависит от заработанных в процессе игры очков. Каждая звезда показывается только в случае активности (пребывания в значение true) определенной глобальной переменной.

Второе окно показывается при проигрыше игрока. Оно состоит только из 2-х спрайтов, приведем функцию его создания:

```
function Window::createLoseWin()
{
   %win = new Sprite(loseWin)
   {
        Position = "0 0";
```

Рис. 10.29. Звезда

```
        Size = "50 40";
        SceneLayer = 28;
        Image = "ToyAssets:loseWin";
        Visible = false;
        BodyType = static;
        UseInputEvents = false;
    };
    myScene.add(%win);

    %but = new Sprite(loseOkBut)
    {
        Position = "-10.25 -0.3";
        Size = "12 10";
        SceneLayer = 28;
        Image = "ToyAssets:loseOkbut";
        Visible = false;
        BodyType = static;
        UseInputEvents = true;
        class = "butNewGame";
    };
    myScene.add(%but);
}
```

Первый спрайт — основа окна, второй спрайт — кнопка. Как и кнопка окна победы, текущая — тоже объект класса butNewGame, следовательно, у них один и тот же обработчик, значит, одинаковые действия.

Рис. 10.30. Окно проигрыша

Рис. 10.31. Кнопка перезапуска игры

Для отображения окна на экране используется функция showLoseWin:

```
function Window::showLoseWin()
{
   fadeWin.Visible = true;
   loseWin.Visible = true;
   loseOkBut.Visible = true;
   $activeMenu = true;
}
```

Она вызывается из функции calcCollectCrystals. Больше в этом окне нет ничего интересного. Идем дальше!

Последнее дополнительное окно — это окно паузы. Метод его создания выглядит так:

```
function Window::createPauseWin()//создать окно паузы
{
   %win = new Sprite(pauseWin)
   {
        Position = "0 0";
        Size = "50 40";
        SceneLayer = 28;
        Image = "ToyAssets:pauseWin";
        Visible = false;
        BodyType = static;
        UseInputEvents = false;
   };
   myScene.add(%win);

   %but = new Sprite(pauseOkBut)
   {
        Position = "1.3 -13.4";
        Size = "10 8";
        SceneLayer = 28;
        Image = "ToyAssets:pauseOkBut";
        Visible = false;
        BodyType = static;
        UseInputEvents = true;
```

```
        class = "butOkGame";
    };
    myScene.add(%but);

    %but = new Sprite(pauseCancelBut)
    {
        Position = "16 1.1";
        Size = "9 9";
        SceneLayer = 28;
        Image = "ToyAssets:pauseCancelBut";
        Visible = false;
        BodyType = static;
        UseInputEvents = true;
        class = "butCancelGame";
    };
    myScene.add(%but);
}
```

Кнопка OK — зеленая галочка — такая же, как в окне победы.

Как видно: в этой функции создаются 3 спрайта: форма окна, кнопка «OK» и кнопка «Cancel». Являясь кнопками, для которых нужно написать

Рис. 10.32. Форма окна паузы

Рис. 10.33. Кнопка Cancel

разные обработчики событий нажатия, 2 последних спрайта имеют разные классы: butOkGame и butCancelGame.

Обработчик для 1-го класса вызывает функцию continueGame, а 2-го — торковскую функцию quit:

```
function butOkGame::onTouchDown(%this, %touchID,
%worldPosition)
{
   continueGame();
}
function butCancelGame::onTouchDown(%this, %touchID,
%worldPosition)
{
   quit();
}
```

С функцией quit все понятно: она завершает работу движка, с другой стороны функция continueGame выглядит более интересной:

```
function continueGame()
{
   HidePauseWin();
   $pause = false;
   if ($action == 0) {
  StepOrderWin::showStepOrderWin("1");
  if ($semafor == 2)
       schedule(1000, 0, "AI::Click");
   } else
   if ($action == 2) {
  StepOrderWin::showStepOrderWin();
   }
}
```

Она вызывает функцию HidePauseWin, которая скрывает спрайты окна паузы:

```
function HidePauseWin()
```

```
{
   pauseCancelBut.Visible = false;
   pauseOkBut.Visible = false;
   pauseWin.Visible = false;
   fadeWin.Visible = false;
}
```

Затем она отключает режим паузы: $pause = false;. Далее в зависимости от очередности хода: только выводит окно последовательности хода (когда право хода принадлежит игроку) или выводит окно очереди хода и планирует выполнение AI::Click, в случае хода компьютерного оппонента.

Окно паузы отображается на экране в момент, когда пользователь нажимает на шестеренку в левом верхнем углу игрового поля. Отображение окна паузы происходит в функции showPauseWin, которая не требует дополнительных комментариев:

```
function Window::showPauseWin()
{
   fadeWin.Visible = true;
   pauseWin.Visible = true;
   pauseOkBut.Visible = true;
   pauseCancelBut.Visible = true;
   $activeMenu = true;
   $pause = true;
}
```

Тем самым мы рассмотрели весь код для создания и отображения дополнительных окон, использующихся для извещения пользователя о результатах игрового процесса, а так же для управления его течением.

4 Заключение

В финальный раз нажмите в Torsion F5, чтобы сохранить все созданные и измененные файлы, интерпретировать их и запустить проект на выполнение. Если вы следовали всем рекомендациям, прислушивались к советам и правильно выполнили все задания, тогда игра успешно запуститься, и на экране появиться главное меню MagicMancala, откуда можно запустить синглплеерную игру.

Тем не менее сложно разработать приложение или игру свободную от ошибок. Поэтому, если у вас в игре есть какие-то сбойные ситуации (баги) или логические ошибки, воспользуйтесь удобным отладчиком,

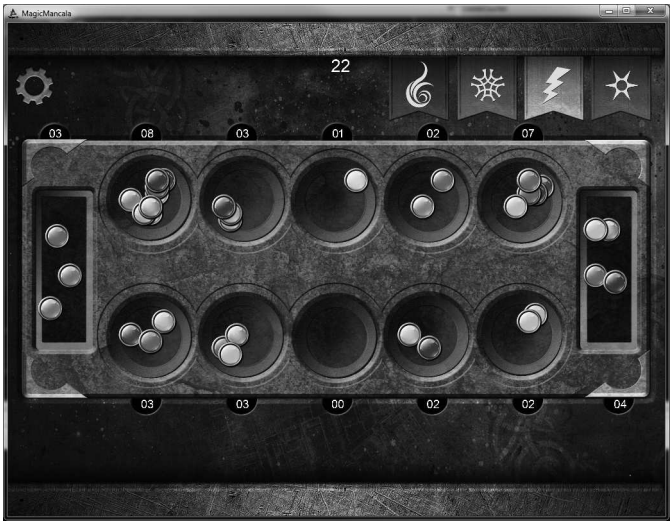

Рис. 10.34. Полностью готовая MagicMancala на PC

встроенным в Torsion, как об этом написано в главе 6. Независимо от размера вашего проекта отладчик выручит в любой ситуации!

Материал главы не простой. В ней мы полностью рассмотрели разработку достаточно крупной мобильной игры. Мы разобрали все детали игрового проекта, решив многие появляющиеся трудности. Однако абсолютно все вопросы мы рассмотреть попросту не смогли, поэтому, в случае их наличия, вы можете связаться с автором по электронной почте.

Разработка игры велась на платформе PC в операционной системе Windows. Между тем с таким же успехом, вы могли разрабатывать MagicMancala на компьютере макинтош в операционной системе OS X. В любом случае сейчас мы имеем полностью работающую игру со всеми запланированными возможностями.

В следующей главе мы создадим билд MagicMancala для мобильной платформы Android, зарегистрируем аккаунт в магазине цифровой дистрибуции Google Play и, наконец, выложим нашу игру в открытый доступ, чтобы любой пользователь устройства на базе ОС Android мог загрузить и поиграть в нашу игру!

Глава 11. MagicMancala в Google Play

Оглавление

1 Подготовка билда: придаем нашей игре товарный вид

Первое, что нам надо сделать, это подготовить билд для платформы Android. Запустите Android Studio и откройте проект из соответствующей поддиректории (в моем случае это папка: c:\Torque\Torque2D-dev-MM-book\engine\compilers\android-studio\).

Наше приложение может представлять собой интересную и увлекательную игру. Но пользователю этого не будет известно, пока он не установить приложение на своё устройство. В Google Play размещено колоссальное количество приложений и игр. Пользователю будет довольно трудно обратить свое внимание именно на нашу игру. Поэтому внимание должна привлекать обертка. То есть картинка, за которой скрывается наша игра, плюс должно быть интересное, но не особо длинное описание. Первые шаги для придания нашему приложению товарного вида осуществляются на локальном компьютере разработчика.

MagicMancala представляет собой совершенную игру в плане управления с устройства, обладающим сенсорным экраном. Если, подготавливая игру Asteroids для запуска на планшете или смартфоне, мы специально добавляли элементы пользовательского интерфейса — виртуальные кнопки для управления игрой, то в случае с MagicMancala ничего добавлять не нужно!

В главе 7 после подготовки игры для запуска на устройстве под управлением ОС Android и ее запуска, первое, с чем мы столкнулись — это стандартный экран — заставка с эмблемой движка Torque 2D. Тогда мы на это не обратили внимания, так как Asteroids мы не собирались выкладывать в Google Play. К MagicMancala у нас требования выше, потому что мы собираемся выложить ее в магазин. Поэтому картинку заставки необходимо заменить на что-то более подходящее к тематике нашей игры. Кроме того, надо изменить иконку приложения, появляющуюся на рабочем столе Android с торковской на нашу специальную.

Сначала сменим картинку на экране — заставке (splash-картинка). Перейдите в подкаталог: engine\compilers\android-studio\app\src\main\assets\ директории с проектом. Здесь находится файл splash.png, представляющий логотип Torque. Подготовьте в каком-нибудь графическом редакторе изображение для заставки MagicMancala размером

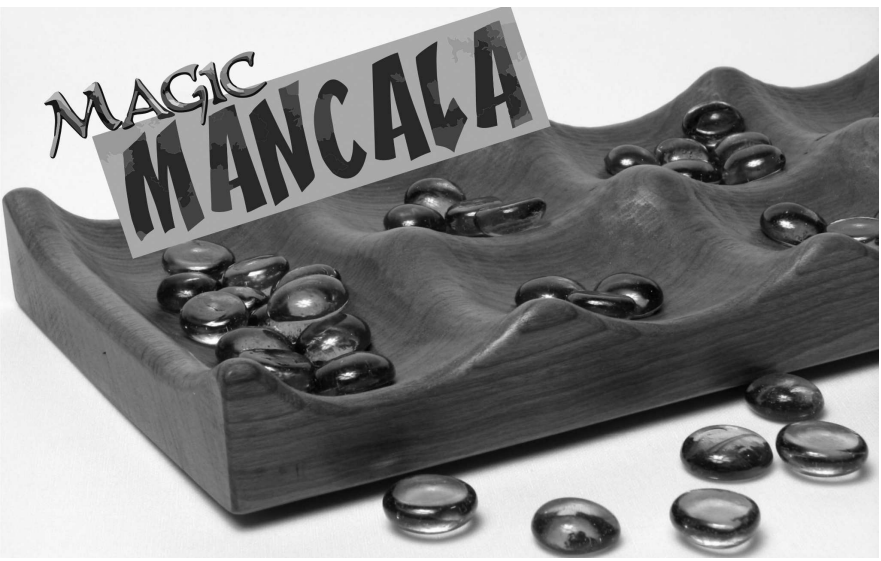

Рис. 11.1. Экран — заставка

2048 × 1496 пикселей или возьмите соответствующее изображение из материалов к книге (рис. 11.1).

Поместите его в указанную выше папку.

Далее нам нужно изображение — пиктограмма для значка приложения на рабочем столе. Перейдите в подпапку engine\compilers\android-studio\app\src\main\res\. В каталогах: drawable-hdpi, drawable-ldpi, drawable-mdpi, drawable-xhdpi находятся соответствующие изображения, отображаемые ОС Android в разных масштабах. По умолчанию — это логотип движка Torque, который нам надо заменить логотипом MagicMancala. Для этого я взял большое splash-изображение, подготовленное на прошлом шаге и уменьшил его до размера 114 × 114 пикселей, при этом, переименовав его в ic_launcher.png. Затем поместил его в каждую из перечисленных папок.

Вернемся в Android Studio, не забудем изменить номера версий sdk в файлах AndroidManifest.xml и build.gradle (Module: app), чтобы освежить память, обратитесь к главе 7. Дополнительно в начале 1-го файла надо изменить имя пакета, во 2-й строке замените:

```
<manifest package="com.garagegames.torque2d"
```

на название своего пакета, к примеру, в моем случае это выглядит так:

```
<manifest package="com.yazevsoft.MagicMancal"
```

Дополнительно надо изменить подпись под значком приложения на рабочем столе, сейчас там отображается Torque 2D, а нам, естественно, надо: MagicMancala. Оставаясь в этом файле, спуститесь чуть ниже до тэга application. В его области найдите строку:

```
android:label="Torque2D"
```

Если подвести к ней курсор, всплывет строка:

```
android:label="@string/app_name"
```

Значит, здесь отображается имя приложения, которое сохранено в списке строк. Изменим значение непосредственно в источнике. Для этого, оставаясь в Android Studio, разверните список app, далее список res, в нем список values, а уже там откройте файл strings.xml. Содержимое открытого файла:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<resources>
   <string name="app_name">Torque2D</string>
</resources>
```

Замените значение переменной app_name:

```
<string name="app_name">MagicMancala</string>
```

После сохранения и возвращения к файлу AndroidManifest.xml мы обнаружим, что указанная выше строка с обозначением подписи к значку приняла нужный вид:

```
android:label="MagicMancala"
```

В файле build.gradle (Module: app) надо заменить значение applicationId, в моем случае стало:

```
applicationId "com.yazevsoft.MagicMancala"
```

Во время подготовки пакета происходит копирование файлов необходимых для работы движка и игры из корневой папки с проектом. Если вы к этому времени скомпилировали движок для Windows (файл с расширением exe), тогда он тоже будет включен в пакет. Тем не менее он совершенно не нужен. Чтобы не приходилось его зря удалять перед сборкой пакета, а потом восстанавливать, лучше добавить исключение на включение исполняемых файлов Windows в скрипт построения пакета. Для этого перейдите в файл build.gradle (свертка Gradle Scripts), пролистайте в нем код до задачи copyGame и после строки exclude '*.torsion' (которая говорит о не включении файла с расширением torsion) добавьте следующую строку:

```
exclude '*.exe'
```

Она вносит коррективы в отборку файлов тем, что запрещает брать файлы с расширением exe.

Теперь можно подключить устройство к компьютеру и запустить MagicMancala на Android для тестирования. Чтобы освежить воспоминания, обратимся к главе 7. После запуска игры на устройстве нас встретит подготовленный splash-экран (рис. 11.2).

Затем после загрузки выполнение продолжится по стандартной схеме: появится главное меню с возможностью выбора режима, после чего запустится игра, где вы сможете играть как на компьютере только за ме-

Рис. 11.2. Заставка

сто мыши использовать сенсорный экран и нажатие с помощью пальцев или другого указателя.

2 Подготовка билда: создание ключа сертификации

В Google Play имеются сотни тысяч приложений, которые могут иметь одинаковые имена, размер, дату выпуска и другие характеристики. Поэтому нужен способ, однозначно идентифицирующий приложение. Для этого в Android есть подписи или, другими словами, ключи сертификации. Они создаются прямо в среде Android Studio. Выбираем в меню Build -> Generate Signed APK. Откроется окно генерации подписанного APK (рис. 11.3).

Поскольку у нас нет ранее созданной подписи, нажмем кнопку Create new. Откроется окно задания параметров нового ключа — New Key Store. Набор параметров достаточно обширный. Наша задача ввести конкретные данные о разработчике. В строке Key store path указывается путь к файлу, в котором будет создана подпись, для выбора директории и задания имени файла можно воспользоваться диалоговым окном. Ниже в поля Password и Confirm вводятся, соответственно, пароль и под-

Рис. 11.3. Окно Generate Signed APK

тверждение. Ниже находится область параметров для ключа: Alias, другими словами, псевдоним, снова пароль и подтверждение и срок годности ключа. Еще ниже расположена область данных о сертификате, которая включает подробные сведения о разработчике, компании и места ее нахождения. Я заполнил это окно следующим образом (рис. 11.4).

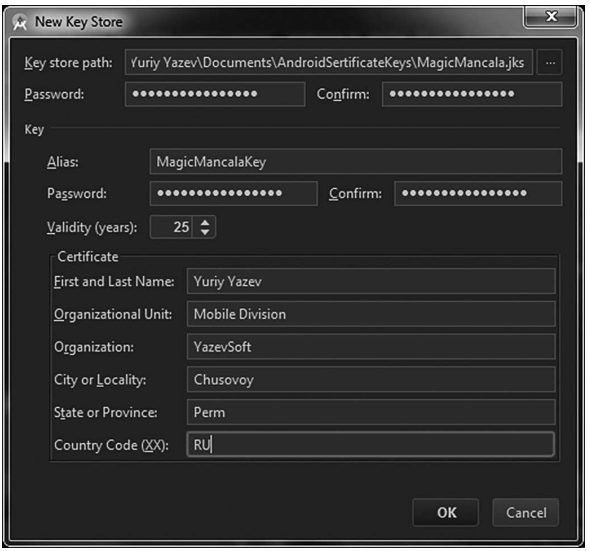

Рис. 11.4. Окно New Key Store

Рис. 11.5. Окно Setup Master Passwords

После нажатия кнопки OK данное окно исчезнет, некоторые введенные в нем данные будут перенесены в строки окна Generate Signed APK. Отметив галочку Remember passwords, перейдем на следующее окно, нажав кнопку Next. Если происходит создание 1-го сертификата, появится окно задания пароля для базы данных паролей вместе с подтверждением (рис. 11.5).

В ином случае появится окно для ввода пароля разблокировки базы данных паролей. После нажатия OK появится новое окно Generate Signed APK, где будут указаны: путь к файлу приложения, тип построения проекта: release и список особенностей — Flavors — можно оставить пустым. В этом окне надо только нажать Finish (рис. 11.6).

После закрытия окна автоматически запустится процесс сборки приложения с подписанным сертификатом. После завершения сборки появится окошко с выбором открыть папку с билдом — APK-файлом, кото-

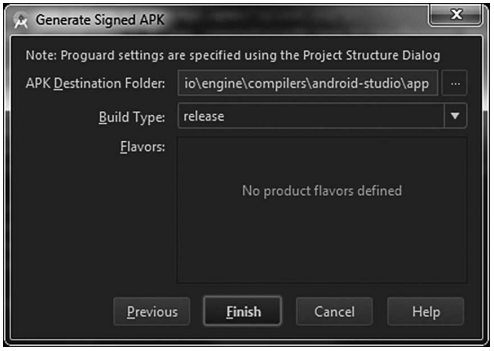

Рис. 11.6. Новое окно Generate Signed APK

рый называется app-release.apk (он находится в подпапке engine\compilers\android-studio\app\).

В результате пакет нашей игры MagicMancala подписан сертификатом, и теперь его можно выкладывать в Google Play.

Примечание. Если вам надо будет перестроить проект, например, после внесения в него каких-либо изменений и/или исправлений, и вы нажмете пиктограмму Run 'app' или команду меню Build -> Make Project, то в результате получите билд, построенный в режиме отладки. Такие пакеты не проходят проверку в Google Play, поэтому нужен билд в режиме выпуска (release). Кроме того, в релизный билд надо включить файл лицензии. Для осуществления этого откройте окно Project Structure (Build -> Edit Build Types в главном меню). В появившемся окне переключитесь на вкладку Signing. На ней мы создадим подпись на основе созданной лицензии, сохраненной в файле. Нажмите зеленый плюс. В списке в центре окна появится новая запись — config. В появившихся справа свойствах можно изменить имя, однако предлагаю оставить значение по умолчанию. Другие свойства необходимо заполнить: Key Alias (псевдоним ключа) — обратите внимание: здесь надо указать тот Alias, который был задан для файла с лицензией, Key Password (пароль ключа) — здесь тоже должен быть пароль, сохраненный в файле с лицензией, Store File (файл лицензии) — здесь, нажав кнопку с многоточием, в открывшемся диалоге надо выбрать файл лицензии — *.jks, который мы создали на предыдущем шаге. Я храню все лицензии (сертификаты) в каталоге: C:\Users\Yuriy Yazev\Documents\AndroidSertificateKeys. Поэтому из него беру файл MagicMancala.jks. В поле Store Password надо ввести пароль, заданный для хранилища подписей (лицензий) (рис. 11.7).

Затем переключитесь на вкладку Flavors. Здесь надо задать конфигурацию подписи, созданную на прошлом шаге. Для этого в ниспадаю-

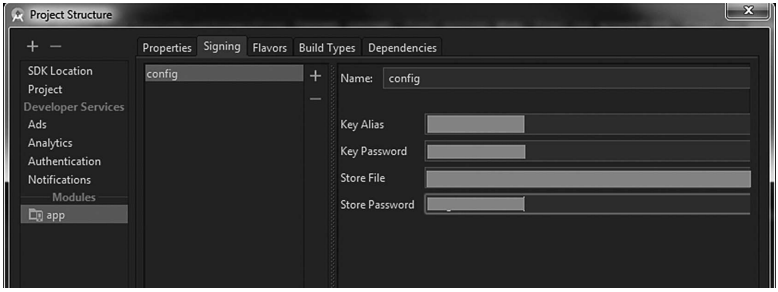

Рис. 11.7. Project Structure — вкладка Signing

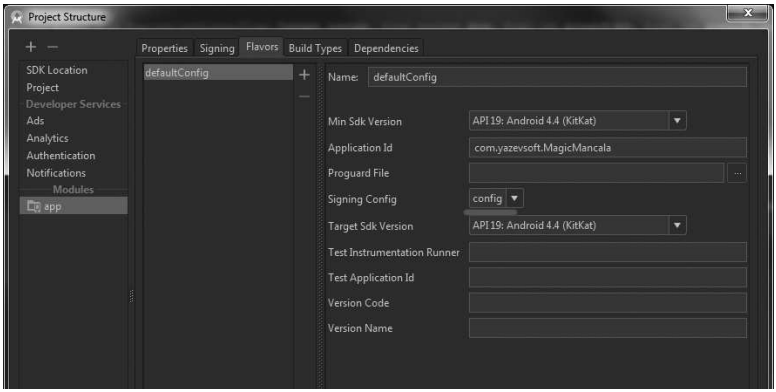

Рис. 11.8. Project Structure — вкладка Flavors

щем списке Signing Config выберите единственный имеющийся там элемент config (рис. 11.8). Остальные параметры можно оставить по умолчанию. Переключитесь на вкладку Build Types и убедитесь, что в центральном списке окна выбрана строка release. После этого можно нажать кнопку OK для применения всех настроек и закрытия окна Project Structure.

Сразу запустится обновление системы построения Gradle. Однако в данный момент Android Studio настроена на построение отладочной версии пакета. Чтобы изменить настройку на построение версии для выпуска, надо вызвать панель Build Variants: Build -> Select Build Variants. Слева от окна редактирования кода и над областью с отладочными сообщениями появится вышеуказанная панель, имеющая 2 столбца: Module и Build Variant. В 1-м — указано приложение — app, во 2-м — способ построения — debug. Чтобы переключить способ построения, надо из ниспадающего списка выбрать release (рис. 11.9).

После переключения надо перестроить проект: Build -> Make Project или Run 'app'. В результате в подкаталоге engine\compilers\android-studio\app\build\outputs\apk\ образуются 2 файла: app-release.apk и app-release-unaligned.apk. Нас интересует 1-й из них. Поэтому его можно прямо сейчас переименовать в MagicMancala.apk. Этот билд абсолютно готов к размещению в Google Play.

Если вы хотите загрузить новый билд в Google Play на место старого, вам так же надо изменить номер версии, чтобы не произошло конфликтов с проверкой при загрузке на сайт. Для этого откройте файл

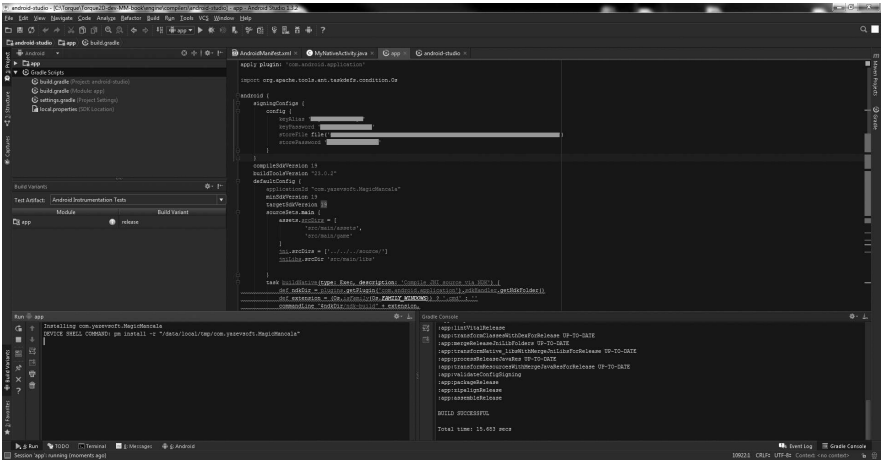

Рис. 11.9. Android Studio — панель Build Variants

AndroidManifest.xml из свитка app -> manifests и в самом его начале после открытия тэга manifest измените номера версий в двух строчках:

```
android:versionCode="5"
android:versionName="5.0"
```

В первой строке указан действительный номер пакета, во второй — декоративный — номер для вывода рядом с именем пакета для пользователя.

3 Google Play

Думаю, что у вас уже есть аккаунт в сервисах Google. Если нет, самое время создать его.

Имея Google-аккаунт можно открыть консоль разработчика Google Play: *https://play.google.com/apps/publish/signup/*. Обратите внимание: это не то же самое, что консоль Google. На 1-й странице консоли предлагается оплатить лицензию разработчика. Как вы знаете, чтобы иметь возможность размещать свои приложения в Google Play, а так же на других площадках цифровой дистрибуции, необходимо оплатить лицензию. В Google Play стоимость лицензии составляет 25 долларов США, что сравнительно ниже, чем у других магазинов, так, например, в App Store (магазине приложений от Apple) аналогичная лицензия стоит

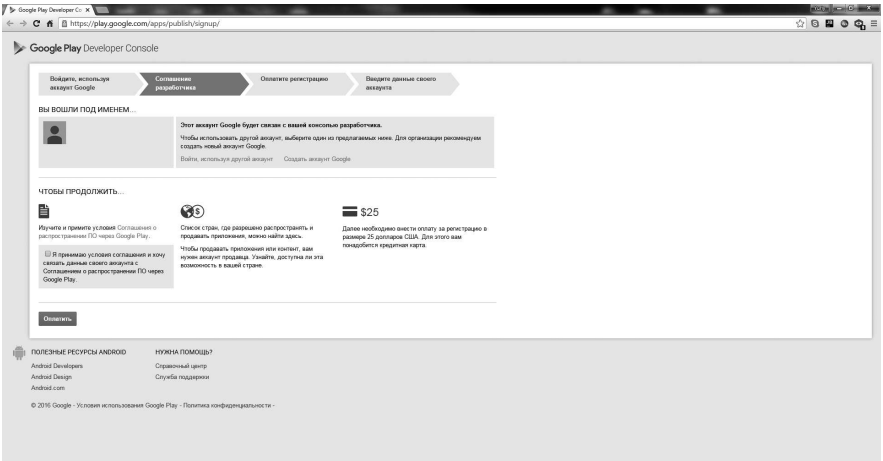

Рис. 11.10. 1-я страница консоли разработчика

$99. При этом в Google Play разработчик, заплатив однажды, может выкладывать свои приложения бесконечно долго, тогда как в App Store указанную сумму надо выплачивать каждый год. Поставьте галочку у пункта о принятии условий соглашения и нажмите кнопку «Оплатить» (рис. 11.10).

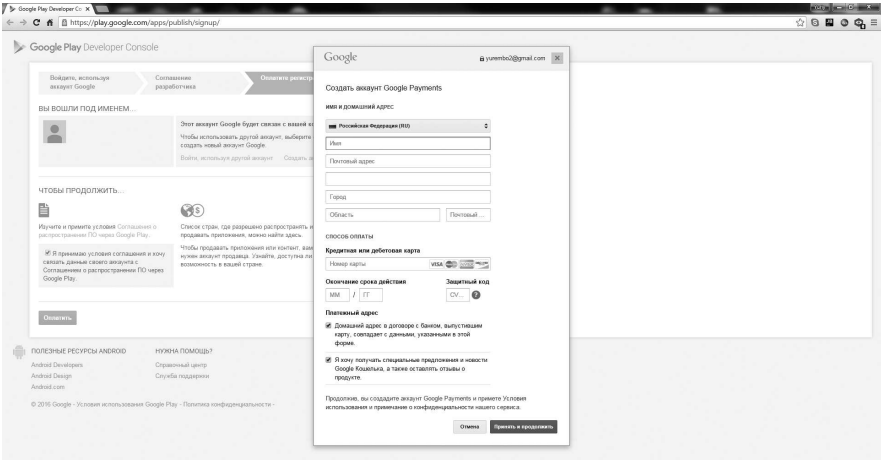

Рис. 11.11. Создание аккаунта в Google Payments

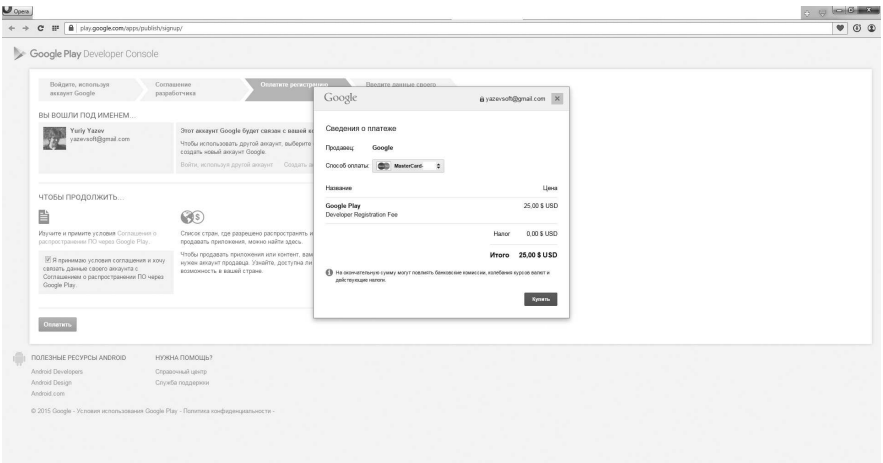

Рис. 11.12. Способ оплаты

Далее появится форма, где надо создать аккаунт Google Payments (ввести соответствующие данные о пользователе и банковской карты: номер, срок окончания, защитный код), другими словами, привязать свою банковскую карту к аккаунту Google; если у вас уже создан аккаунт Google Payments, тогда надо будет указать способ оплаты. В 1-м случае

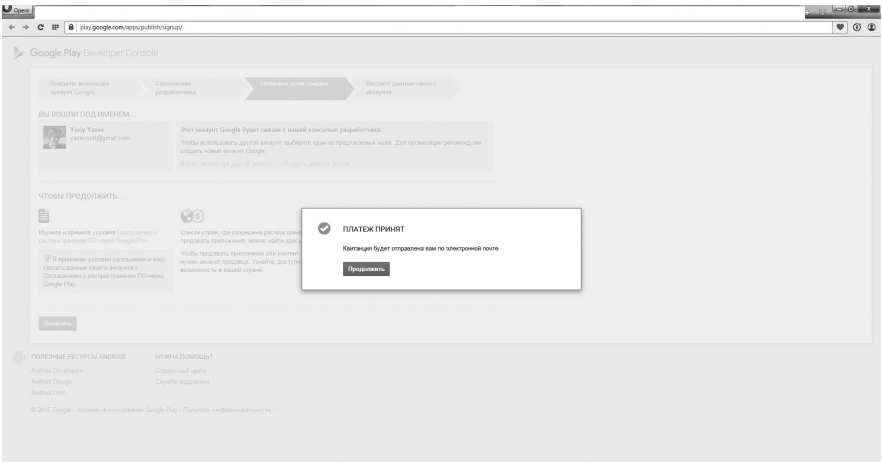

Рис. 11.13. Принятие платежа

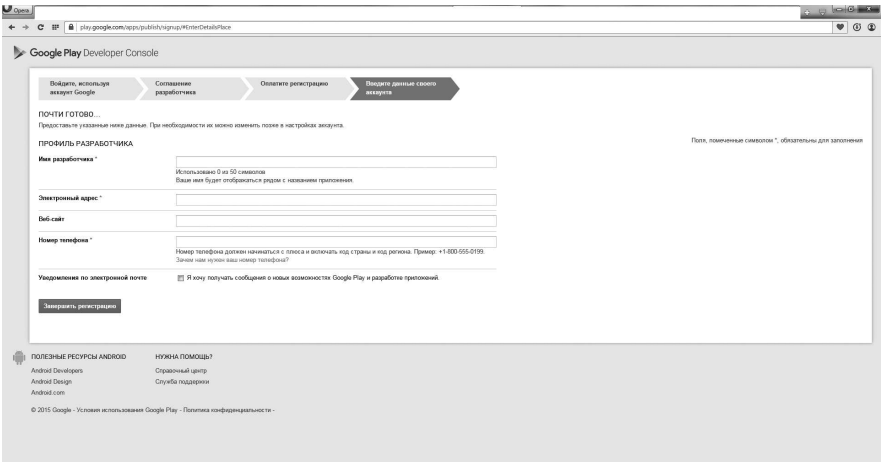

Рис. 11.14. Заполнение профиля разработчика

после завершения создания Google Payments надо нажать «Принять и продолжить» (рис. 11.11), во 2-м — кнопку «Купить».

Отобразится форма о принятии платежа (рис. 11.13).

После нажатия кнопки «Продолжить» отобразится новая форма, в которой надо указать данные для профиля разработчика. Их перечень виден на рисунке (рис. 11.14).

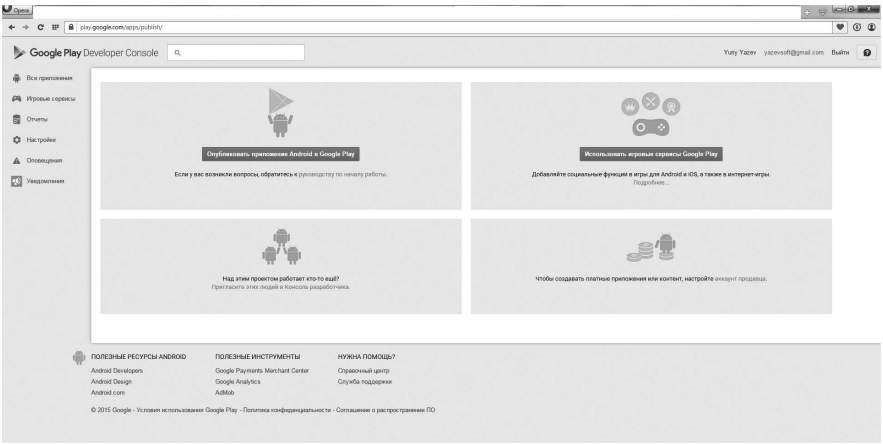

Рис. 11.15. Сервисы для работы с магазином Google

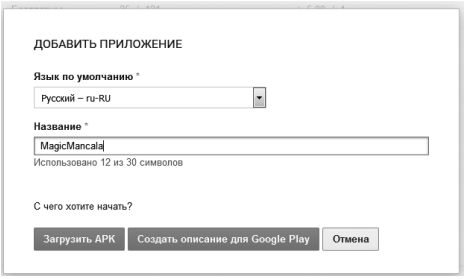

Рис. 11.16. Название приложения

Когда все данные будут указаны, можно нажать кнопку «Завершить регистрацию». Отобразится новая страница с функциями для работы с сервисами, нажмите кнопку «Опубликовать приложение Android в Google Play» (рис. 11.15).

В результате откроется страница для выбора языка и ввода названия для загружаемого приложения (рис. 11.16).

После задания имени откроется страница управления указанным проектом (рис. 11.17).

Далее нажмем кнопку «Загрузить первый APK рабочей версии». Появится форма, в которой выберем кнопку «Обзор файлов» (рис. 11.18).

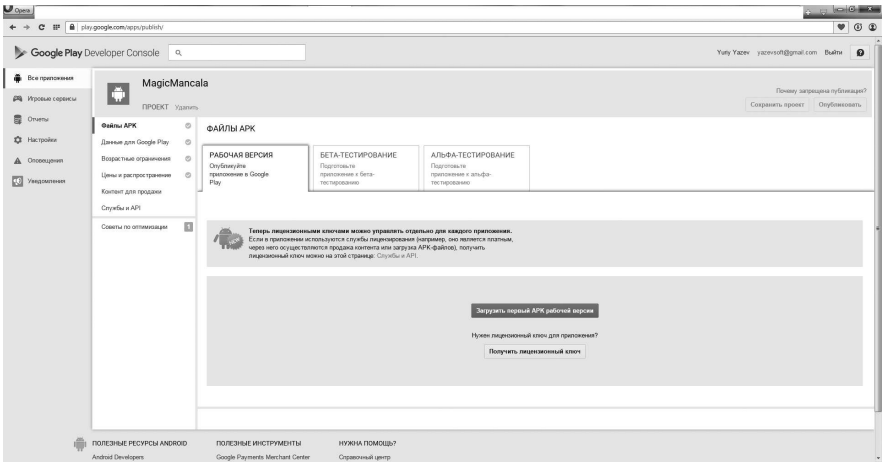

Рис. 11.17. Управление проектом

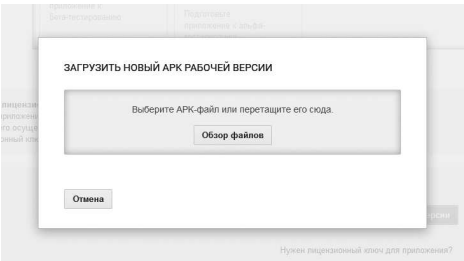

Рис. 11.18. Новая форма

Откроется диалог выбора файла. Напомню: файл с нашей игрой находится в подкаталоге engine\compilers\android-studio\app\ (или engine\compilers\android-studio\app\build\outputs\apk\) под названием app-release.apk. Его можно переименовать: MagicMancala.apk. С помощью диалога открытия файла выберем MagicMancala.apk из указанной выше папки. Начнется загрузка (рис. 11.19).

Обратите внимание: мы загружаем релизную (release) версию нашей игры. В случае если попытаться загрузить отладочную (debug) версию, Google Play сообщит об ошибке, запретив ее загружать. Также ошибка может произойти, когда имя пакета (в моем случае com.yazevsoft.MagicMancala) уже используется в Google Play.

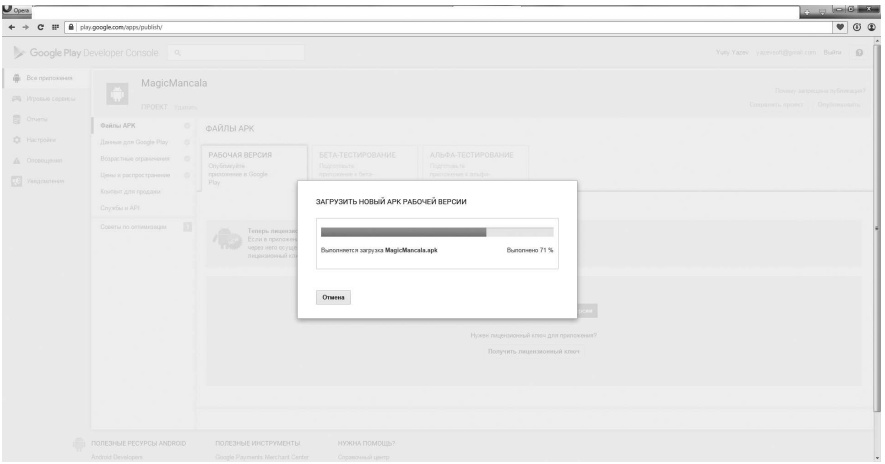

Рис. 11.19. Загрузка пакета MagicMancala.apk в Google Play

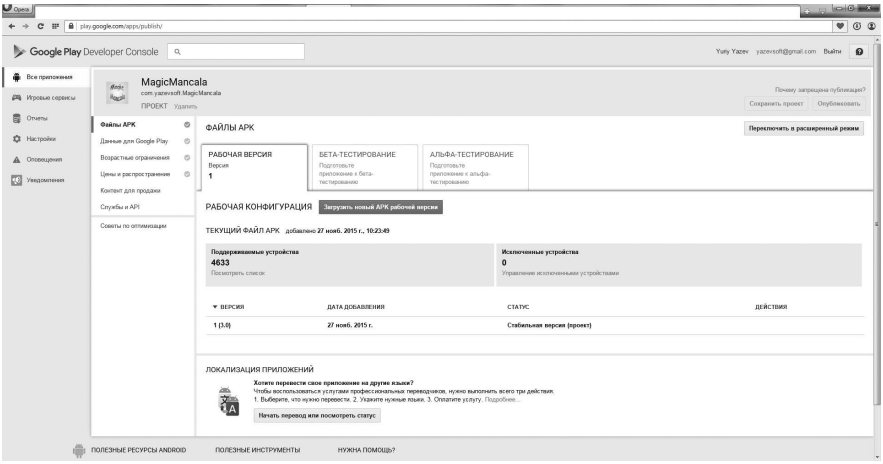

Рис. 11.20. Обновленная страница управления проектом

После успешной загрузки apk-файла отобразится страница управление проектом (рис. 11.20).

На данный момент пакет с игрой только загружен в ваш аккаунт Google Play. Прежде, чем он попадет в открытый доступ, надо совершить настройки и придать проекту товарный вид, используя web-интерфейс консоли разработчика.

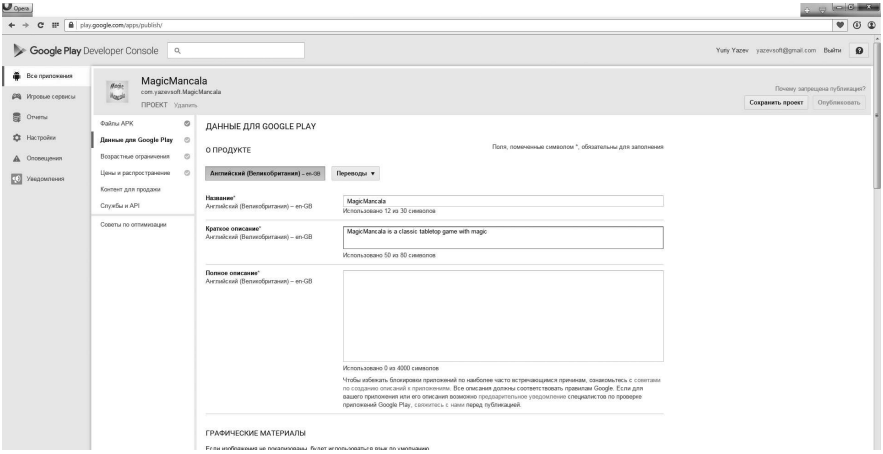

Рис. 11.21. Данные для Google Play

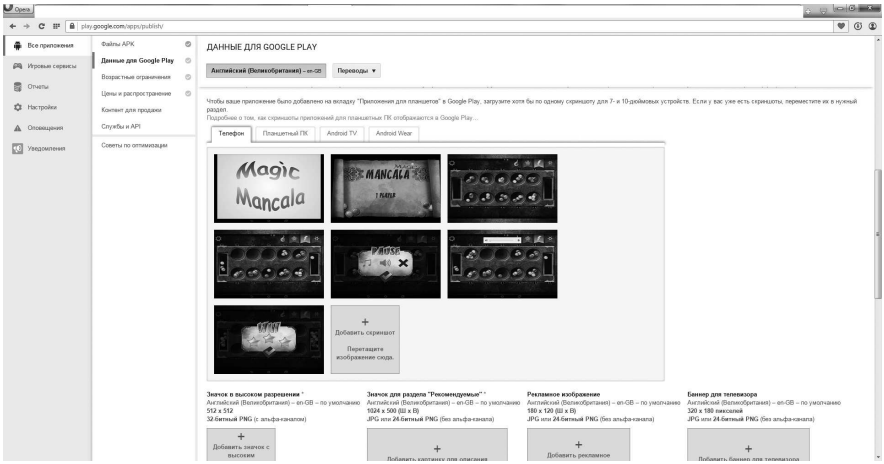

Рис. 11.22. Загрузка картинок

Слева рядом с названием вкладки «Файлы APK» появилась галочка в зеленом кружке, говорящая о том, что файл успешно загружен. Переключитесь на вкладку «Данные для Google Play» (рис. 11.21).

На ней надо ввести название, краткое и полное описание. Плюс здесь же загружаются скриншоты и картинки для значков — пиктограмм. Вы можете создать их сами или воспользоваться подготовленными мной изображениями, которые находятся в материалах к книге в соответствующей главе папке. Скриншоты нужны для каждого поддерживаемого устройства: телефон, планшетный ПК, Android TV, Android Wear. По идее, для каждого устройства скриншоты готовятся индивидуально, но мы воспользуемся одними и теми же. По сути, скриншоты нужны для рекламы игры на витрине магазина. Значки нужны для отображения иконки запуска в разных разрешениях экрана.

Ниже вводится ссылка на промо-ролик (размещенный, естественно, на youtube.com), определяются свойства приложения: тип приложения — игры; категория — настольные игры; возрастные ограничения — для всех возрастных групп. Еще ниже вводится контактная информация с разработчиком: web-сайт, e-mail, политика конфиденциальности (можно поставить галочку «Без политики конфиденциальности») (рис. 11.23).

Далее после заполнения всех нужных полей вверху страницы жмем кнопку «Сохранить проект». Слева рядом с названием вкладки «Данные для Google Play» появится галочка в зеленом кружке. Переключитесь на

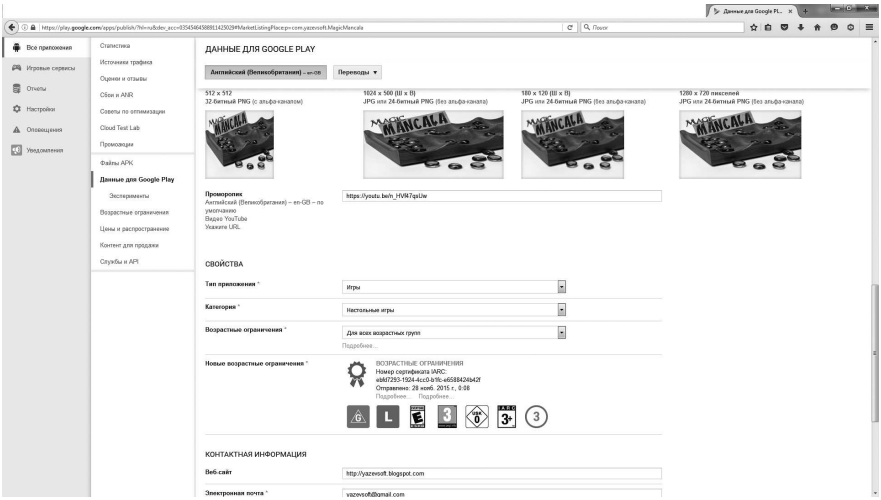

Рис. 11.23. Свойства и контактная информация

следующую вкладку — «Возрастные ограничения». Здесь более детально указываются ограничения в возрасте относительно контента игры: насилие, устрашение, материалы сексуального характера и прочее (рис. 11.24).

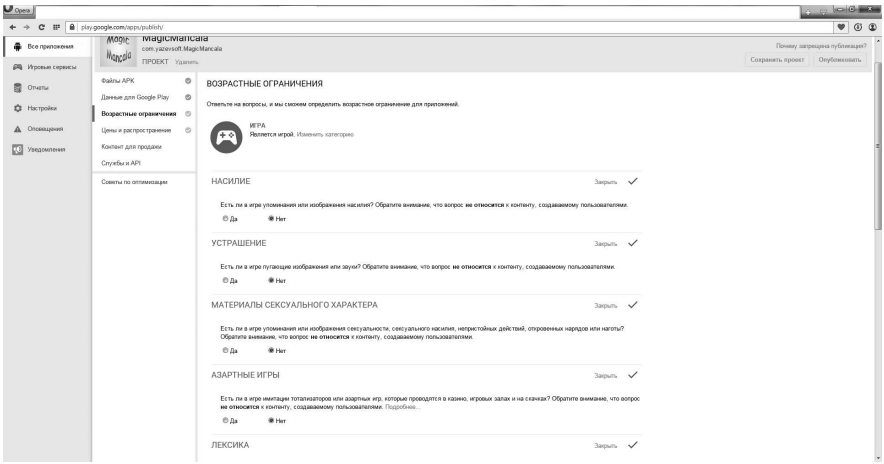

Рис. 11.24. Возрастные ограничения

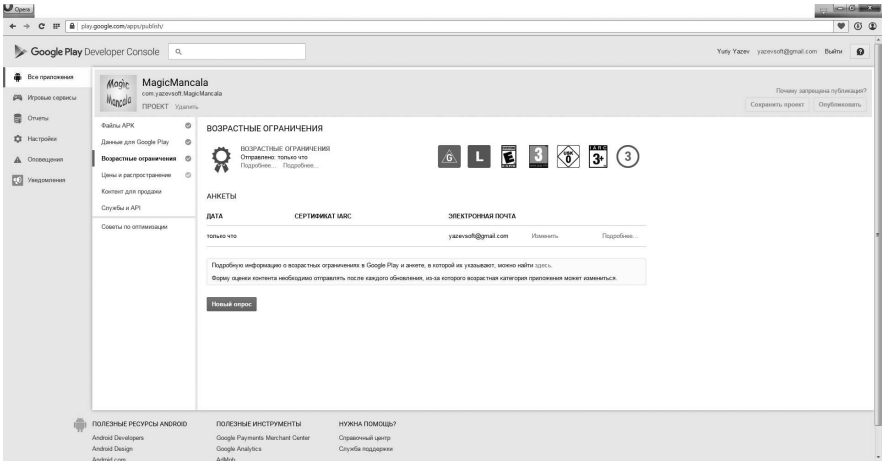

Рис. 11.25. Сертификат IARC

Нажимаем кнопку «Сохранить проект». В результате на основе отвеченных вопросов к вашему приложению будет привязан сертификат IARC (рис. 11.25).

Переходим на следующую вкладку — «Цены и распространение». На новой странице отмечаем, что это приложение бесплатное, доступно во

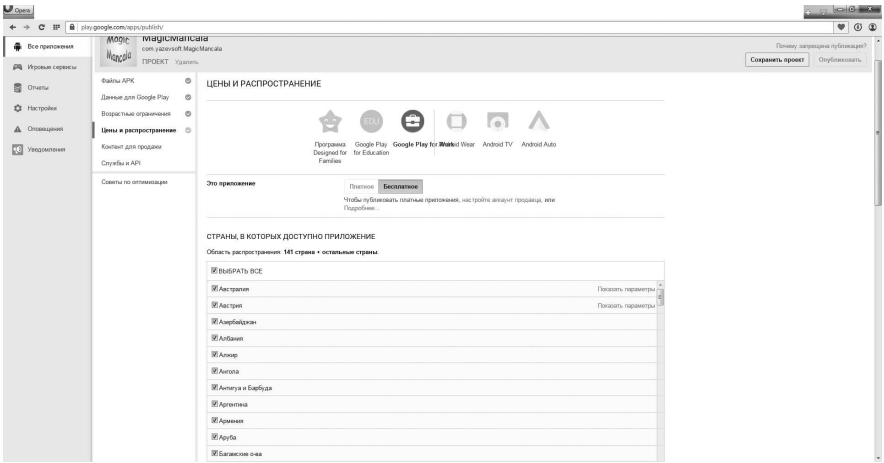

Рис. 11.26. Цены и распространение — 1

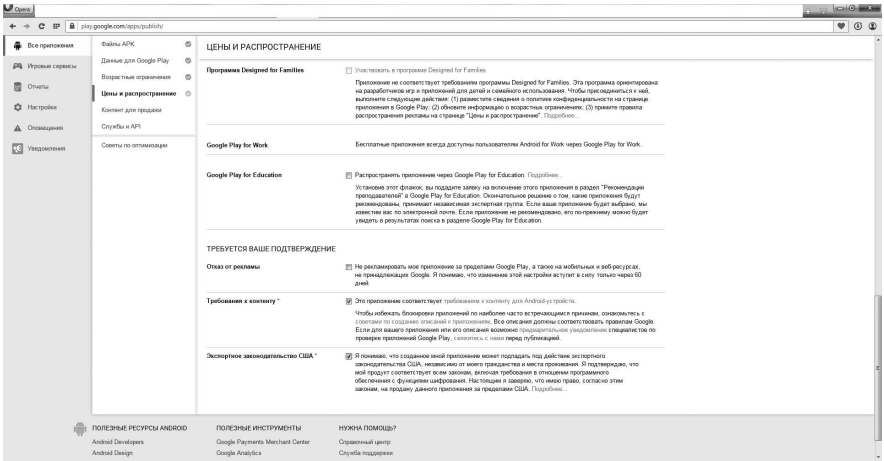

Рис. 11.27. Цены и распространение — 2

всех странах; прокрутите страницу ниже, на вопрос «Есть реклама» ответьте отрицательно. Находящиеся ниже вопросы можно пропустить до «Требования к контенту». Рядом с ним надо поставить галочку, как и с вопросом ниже: «Экспортное законодательство США» — тоже отметить галочкой (рис. 11.26) (рис. 11.27).

Рис. 11.28. Игра доступна для загрузки в Google Play

После этого сохраните проект. В итоге имеем все 4 галочки, находящиеся слева. Значит, можно публиковать проект. Нажимаем кнопку «Опубликовать». Далее Google оповестит, что наше приложение готовиться к публикации и будет готово в течение нескольких часов.

По прошествии 2-х — 3-х часов наша игра будет доступна в Google Play для загрузки (рис. 11.28).

Можно уже сейчас порадовать своих друзей новой игрой!

4 Внедрение рекламы в MagicMancala

Хотя наша игра распространяется бесплатно, она может приносить разработчикам доход — через рекламу. Это самый распространенный способ получения денег через мобильные игры. На самом деле, по статистике прямая продажа приложений и игр приносит меньший доход, чем с помощью рекламы. Поскольку в основном напрямую пользователи покупают игры только от больших, зарекомендовавших себя издателей (Electronic Arts, Microsoft, Ubisoft и др.). Еще один способ монетизации заключается в продаже внутри игровых объектов, например: доспехов, оружия, космических кораблей и многого другого. Тем не менее для нас больше подходит реклама. Есть множество рекламодателей. Мы воспользуемся услугами сервиса AdBuddiz.

AdBuddiz предоставляет удобный SDK для всех распространенных платформ (среди которых: Android, iOS, Unity, Cordova, Corona, Adobe, Xamarin, Marmalade), который легко внедрить в приложение или игру. В обоих случаях он одинаково стабильно работает и не взывает сбоев. Рекламодатели охотнее платят за инсталляции, нежели просмотры и клики. В зависимости от страны, за одну инсталляцию на виртуальный счет разработчика поступает обычно от 1 до 3-х долларов США. Важно заметить, чтобы перевести деньги с виртуального счета на реальный, ваш аккаунт должен набрать 100$ или больше. Перевод осуществляется через 45 дней после окончания месяца, в котором была набрана нужная сумма. AdBuddiz выплачивает деньги посредством системы денежного перевода PayPal или с помощью банковских переводов, при этом первый вариант является более предпочтительным. Статистика по рекламе в ваших приложениях на странице Dashboard обновляется в 0:00 часов (по Гринвичу) каждый день. Выплаты осуществляются в двух валютах (по выбору): доллары США и Евро. Все это можно настроить на сайте adbuddiz.com. Кроме SDK для внедрения рекламы, у AdBuddiz есть API для просмотра статистики, не используя веб-интерфейс.

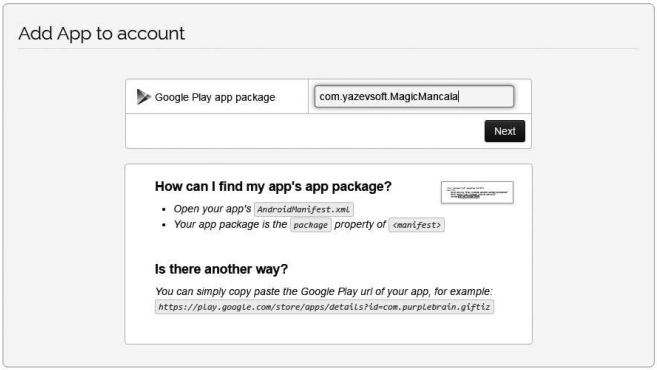

Рис. 11.29. Ввод имени пакета

Чтобы добавить в нашу Android-игру возможность демонстрации рекламы, надо первым делом зарегистрироваться на сайте http://www.adbuddiz.com/. Процедура регистрации вполне стандартная: надо указать e-mail и придумать пароль. Затем нужно перейти в раздел Dashboard (открывается автоматически) и нажать кнопку Add app (добавить приложение). Вам предложат выбрать платформу: Google Play или Apple Store. На следующей странице надо ввести имя пакета и нажать Next (рис. 11.29).

Далее, если приложение уже успешно зарегистрировано в Google Play, то загрузиться новая страница, на которой отобразится иконка

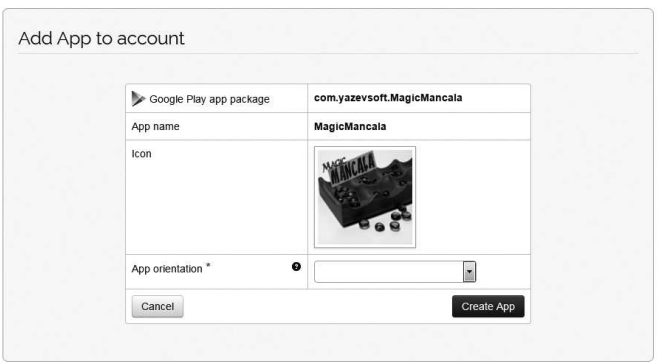

Рис. 11.30. Задание ориентации

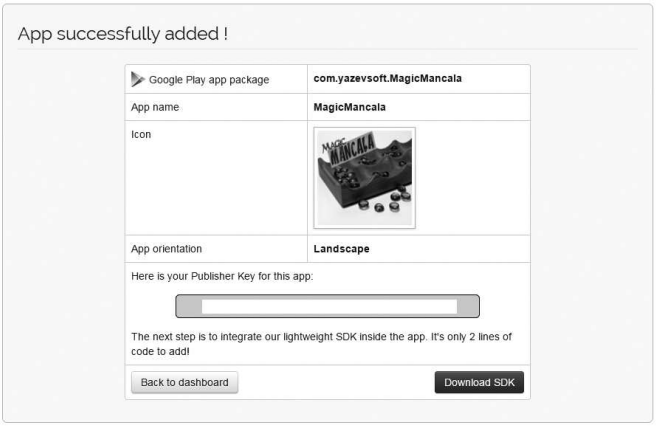

Рис. 11.31. Ключ издателя

приложения и будет предложено выбрать стандартную ориентацию, в которой находится приложение. Наша игра находится в горизонтальном положении. Следовательно, выберем горизонтальную (horizontally) (рис. 11.30).

Жмем кнопку «Create App». На следующей — финальной странице, сообщающей об успешном добавлении приложения, будет выведен ключ издателя (Publisher Key) для нашей игры (рис. 11.31).

Этот же ключ будет находиться в письме, которое придет на ящик электронной почты, указанный при регистрации на http://www.adbuddiz.com/. На этом добавление приложения завершено. Итак, Publisher Key у нас уже есть, совсем скоро он нам понадобится.

Откроем проект игры в Android Studio. Далее откроем файл build.gradle (Module: app), в конец файла после описания параметров Android { … } добавим следующий код:

```
dependencies {
      compile 'com.purplebrain.adbuddiz.sdk:AdBuddiz-
Java:3.+'
   }
   repositories {
      maven {
       url 'http://repository.adbuddiz.com/maven'
      }
   }
```

Затем на инструментальной панели Android Studio надо нажать кнопку «Sync Project with Gradle Files» для обновления проекта. После построения проекта с помощью системы Gradle откроем файл AndroidManifest.xml. В начало файла в раздел задания пользовательских разрешений (uses-permission) добавим следующий код:

```
<!-- Mandatory permission -->
<uses-permission android:name="android.permission.
INTERNET" />
<!-- Highly recommended permission to get more ads and
revenue -->
<uses-permission android:name="android.permission.
ACCESS_NETWORK_STATE" />
```

Кроме этого, в раздел тэга <application …> перед его закрытием вставим такие строчки для добавления активности с ее обязательной темой:

```
<activity android:name="com.purplebrain.adbuddiz.sdk.
AdBuddizActivity"
          android:theme="@android:style/Theme.
Translucent" />
```

Тем самым мы дополняем тэг <activity …>, поскольку он уже написан.

Еще в этом файле надо увеличить номер версии приложения — строчка: android:versionCode=«2» в манифесте. Потому что при загрузке в Google Play приложения с одинаковым именем и той же версией, магазин сообщит о нарушении и откажет размещать.

Следующим шагом надо настроить SDK и включить объект рекламы. В обычных android-приложениях есть java-метод protected void onCreate() { … }, в нем и осуществляется это действо, однако игра на движке Torque 2D не совсем обычное android-приложение, оно полностью написано на C/C++, и хотя в нем имеется код инициализации на java, метод onCreate() там отсутствует. Но есть аналогичный метод, в котором можно произвести настройку рекламы. Откроем файл MyNativeActivity.java из свертка app -> java -> com.garagegames.torque2d и перейдем в метод onWindowFocusChanged. Он выглядит следующим образом:

```
@Override
  public void onWindowFocusChanged(boolean hasFocus) {
    super.onWindowFocusChanged(hasFocus);
    if (hasFocus) {
        getWindow().getDecorView().
setSystemUiVisibility(
              View.SYSTEM_UI_FLAG_LAYOUT_STABLE
```

```
                        | View.SYSTEM_UI_FLAG_LAYOUT_
HIDE_NAVIGATION
                        | View.SYSTEM_UI_FLAG_LAYOUT_
FULLSCREEN
                        | View.SYSTEM_UI_FLAG_HIDE_
NAVIGATION
                        | View.SYSTEM_UI_FLAG_FULLSCREEN
                        | View.SYSTEM_UI_FLAG_IMMERSIVE_
STICKY);}
   }
}
```

После установки свойств окна добавим такую строчку:

```
AdBuddiz.setLogLevel(AdBuddizLogLevel.Info);
```

Тем самым мы включили средства логирования данного SDK.

В случае отсутствия регистрации приложения для проведения проверки на успешность настройки SDK надо включить тестовый режим (даже если приложение уже зарегистрировано, как в нашем случае, эту проверку лучше провести). Это делается такой строчкой:

```
AdBuddiz.setTestModeActive();
```

Затем надо инициализировать SDK для получения рекламы:

```
AdBuddiz.setPublisherKey("полученный Publisher Key");
```

С помощью этого метода мы передаем полученный при регистрации приложения Publisher Key для инициализации ключа, по которому будет определяться приложение в Google Play, и, в соответствии с его свойствами (такими как: тип приложения, возрастные ограничения и прочее) будет подбираться и показываться реклама. Между тем, если приложение не зарегистрировано (отсутствует Publisher Key), и/или нам надо осуществить проверку, тогда в качестве параметра методу надо передать строку: TEST_PUBLISHER_KEY. В этом случае будет выведена стандартная реклама от AdBuddiz.

Следующим методом наше приложение получает рекламу и сохраняет ее в кэше:

```
AdBuddiz.cacheAds(this);
```

Наконец, отображаем рекламу:

```
AdBuddiz.showAd(this);
```

К слову, последний вызов можно поставить в любом месте приложения, когда нужно отобразить рекламу, например, между уровнями или при смерти главного персонажа. Однако мы хотим, чтобы реклама показывалась сразу после загрузки приложения, поэтому вставили вызов рекламы сразу после инициализации рекламного механизма.

В итоге метод onWindowFocusChanged принял следующий вид:

```
@Override
  public void onWindowFocusChanged(boolean hasFocus) {
    super.onWindowFocusChanged(hasFocus);
    if (hasFocus) {
        getWindow().getDecorView().
setSystemUiVisibility(
             View.SYSTEM_UI_FLAG_LAYOUT_STABLE
                    | View.SYSTEM_UI_FLAG_LAYOUT_
HIDE_NAVIGATION
                    | View.SYSTEM_UI_FLAG_LAYOUT_
FULLSCREEN
                    | View.SYSTEM_UI_FLAG_HIDE_
NAVIGATION
                    | View.SYSTEM_UI_FLAG_FULLSCREEN
                    | View.SYSTEM_UI_FLAG_IMMERSIVE_
STICKY);}
      AdBuddiz.setLogLevel(AdBuddizLogLevel.Info);
      AdBuddiz.setTestModeActive();
      AdBuddiz.setPublisherKey("TEST_PUBLISHER_KEY");
      AdBuddiz.cacheAds(this);
      AdBuddiz.showAd(this);
  }
}
```

Чтобы поставить компилятор в известность относительно объекта AdBuddiz и его методах, надо импортировать 2 заголовочных файла. Поднимитесь вверх по коду до области импортирования заголовков и добавьте туда 2 следующие строчки:

```
import com.purplebrain.adbuddiz.sdk.AdBuddiz;
import com.purplebrain.adbuddiz.sdk.AdBuddizLogLevel;
```

После этого надо обновить проект (построить его заново) и можно запустить модифицированную игру на тестирование. Как и ожидается: реклама появится после загрузки приложения, сразу после экрана заставки.

Стандартная реклама от AdBuddiz выглядит следующим образом (рис. 11.32), (рис. 11.33).

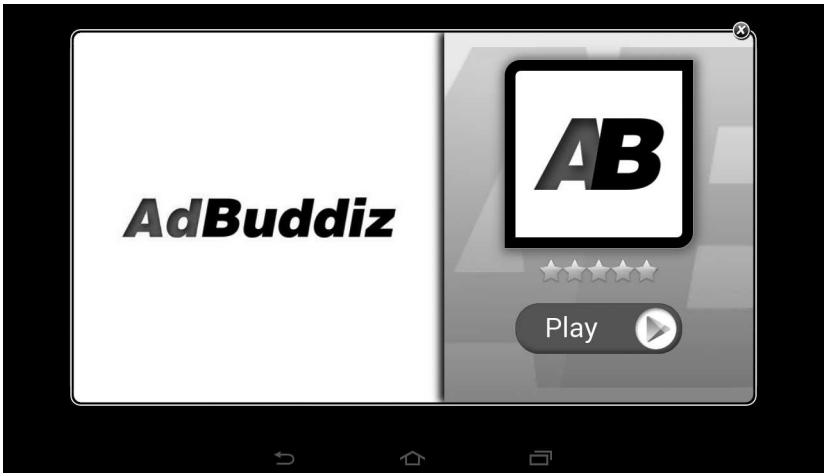

Рис. 11.32. Тестовая реклама — 1

Когда добавленный код будет протестирован, и вы убедитесь в том, что все работает, как надо, удалите строчку: AdBuddiz.setTestModeActive(); А в вызове метода setPublisherKey параметр «TEST_PUBLISHER_KEY» замените на полученный в результате регистрации приложения на http://adbuddiz.com ключ издателя — Publisher Key (рис. 11.34).

Рис. 11.33. Тестовая реклама — 2

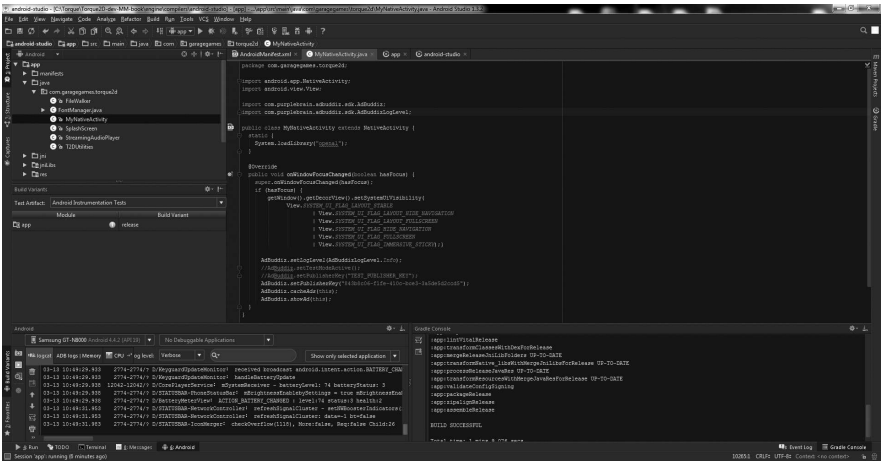

Рис. 11.34. Измененный файл MyNativeActive.java

После регистрации приложения и задания ему выданного Publisher Key в него станет приходить реальная реклама. Таким образом, если наше приложение — игра, то в него будет, в большинстве своем, приходить реклама об играх.

Рис. 11.35.Реальная реклама — 1

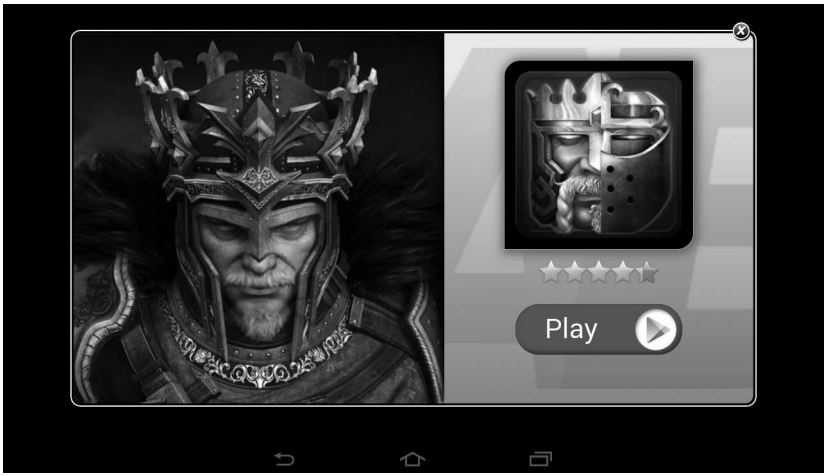

Рис. 11.36. Реальная реклама — 2

Перед тем, как переходить к следующему разделу не забудьте собрать пакет выпуска (release), как это описано в конце 2-го раздела данной главы (в примечании).

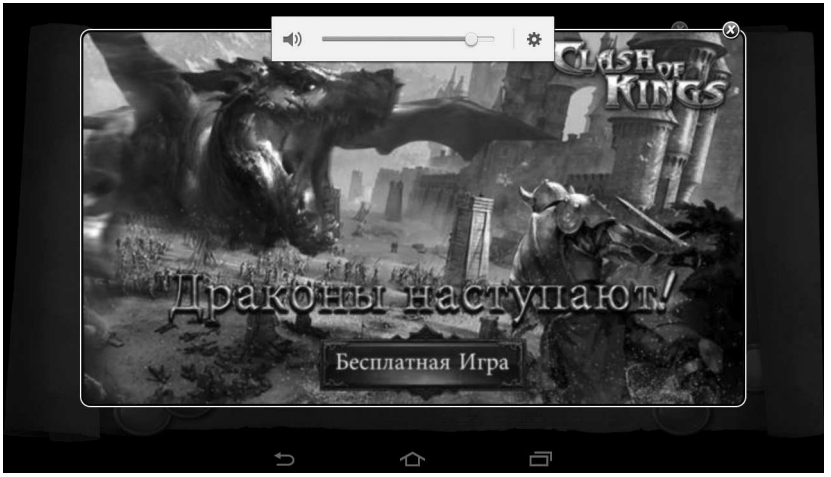

Рис. 11.37. Реальная реклама — 3

Рис. 11.38. Реальная реклама — 4

5 Обновление пакета в Google Play

У нас готов билд со встроенной рекламой от AdBuddiz. Теперь нам надо заменить пакет нашей игры, находящийся в Google Play, нашим новым — улучшенным пакетом.

Откройте в браузере консоль Google Play (play.google.com/apps/publish/). Сразу же откроется вкладка «Все приложения». На ней представлены все ваши опубликованные приложения. Щелчком выберите MagicMancala. В браузер загрузится уже знакомая нам страница «Данные для Google Play», где можно изменить название, описание и графические материалы (скриншоты, пиктограммы). Также сейчас можно посмотреть статистику скачиваний и установок, выставленные оценки, информацию о сбоях и другое. Все эти сведения находятся на соответствующих вкладках, размещенных слева.

Нам надо загрузить новый пакет, поэтому перейдем на вкладку «Файлы APK». Здесь показана информация об уже загруженных пакетах. Нажмем кнопку «Загрузить новый APK рабочей версии». Появится форма для загрузки нового билда, который можно либо выбрать с помощью диалога, либо перетащить из менеджера файлов на поверхность формы.

Начнется загрузка.

Обратите внимание: новая версия пакета, уже имеющегося в Google Play, должна быть подписана тем же сертификатом, что и старая. В про-

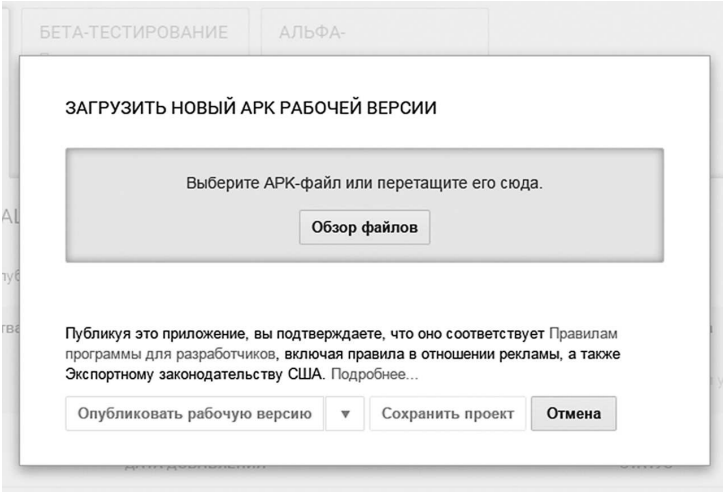

Рис. 11.39. Форма в момент перетаскивания файла

тивном случае новый пакет будет отвергнут. Также напоминаю: версия пакета должна отличаться от предыдущих.

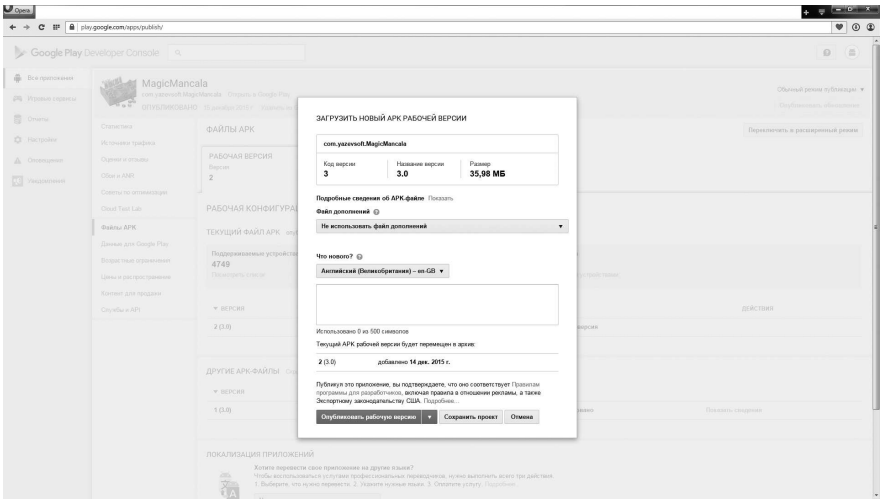

Рис. 11.40. Форма обновления пакета

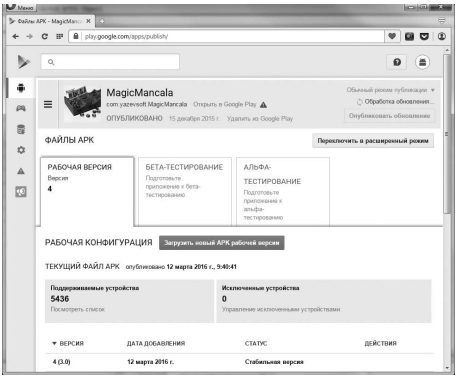

Рис. 11.41. Пакет в процессе обновления

По завершению загрузки появится форма со сведениями о новом пакете. Можно указать подробные сведения об APK файле, которые, затем отредактировать. К примеру, изменить описание, добавить новые скриншоты и т.д. (рис. 11.40).

Когда все будет готово нажимаем кнопку «Опубликовать рабочую версию». Окно исчезнет, а сверху появится надпись о том, что обновление обрабатывается, и новая версия пакета будет доступна через несколько часов (рис. 11.41).

6 Заключение

В этой главе мы узнали минимальный набор действий, необходимый для подготовки к размещению игры в магазине Google Play: мы изменили изображения, подписи, разобрали процедуру копирования игры для пакета. Также мы рассмотрели процедуру создания лицензионного ключа (сертификата) для нашего приложения. В следующем разделе мы занялись непосредственным размещением пакета нашей игры в магазине, пройдя через регистрацию, добавление описания, подготовку и выкладывание картинок, тем не менее к концу 3-го раздела в Google Play уже находилась наша игра, и в нее мог поиграть любой желающий. Но на этом мы не остановились. Мы добавили в нашу игру возможность получения и вывода рекламы от AdBuddiz, благодаря чему мы можем получать доход с нашей игры. Наконец, в последнем разделе мы обновили пакет с нашей игрой и залили его в Google Play, заменив предыдущий.

Этой главой мы заканчиваем развитие нашей логической игры MagicMancala, проведя полный цикл от разработки до издания и доставления продукта пользователям. Дальнейшая судьба игры находится в ваших руках: вы можете придумывать и реализовывать новые уровни, новые игровые механизмы, новые правила, невероятные магические заклинания с потрясающими спецэффектами. А потом обновлять пакет и обновлять его в Google Play для возможности поиграть в вашу игру другим пользователям устройств на базе Android!

Движок Torque 2D идет в ногу со временем, и в нем скоро после выхода появляется поддержка новых устройств, которые можно использовать в игровом процессе. В следующей главе мы рассмотрим: как реализовать в игре управление с помощью устройства Leap Motion.

Глава 12. Использование контроллера Leap Motion

Оглавление

Устройство Leap Motion (скачкообразное движение) Controller берет свое начало из славной семейки беспроводных контроллеров нового поколения, таких как: Wii Remote, PlayStation Move, однако ближайшим родственником является — Xbox Kinect. При этом Leap Motion позволяет управлять компьютерной системой намного более точно, чем Kinect. В отличие от последнего, Leap Motion реагирует на движения исключительно рук, он в 200 раз точнее определяет движения рук и пальцев, чем Kinect, даже если ими быстро двигать в пространстве. Это устройство еще плотнее приближает нас к настоящей виртуальной реально-

сти — к естественному интерфейсу между человеком и машиной — к революционному способу управления цифровым контентом.

1 Обзор устройства

После выхода сенсора Kinect на волне его успеха стали появляться другие устройства бесконтактного управления. Kinect послужил для роста и развития рынка подобных устройств. Инвесторы увидели его перспективу и смысл вкладывать в него средства.

Однако наиболее значимым и, следовательно, успешным стал контроллер Leap Motion. Как и прародитель, последний основан на технологии захвата движения. Это устройство подключается к порту USB и по размеру не превышает пару сложенных флэшки. С технической стороны для захвата проекции пользовательских рук в пространстве устройство Leap Motion использует два оптических сенсора (камеры) и инфракрасный источник света (рис. 12.1).

Как заявляют разработчики: в будущих версиях устройства количество камер может быть изменено. Девайс размещается рабочей поверхностью кверху рядом с экраном для создания ощущения управления объектами на экране с помощью рук. После подключения устройства над ним образуется виртуальная перевернутая пирамида с централь-

Рис. 12.1. Leap Motion в разборе

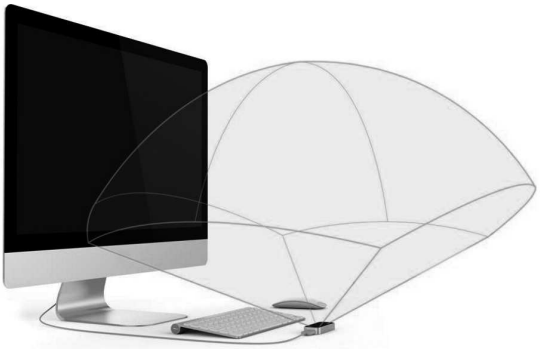

Рис. 12.2. Вид устройства

ной вершиной в устройстве. Наиболее эффективный диапазон распространяется от 25 до 600 мм над контроллером с областью видимости 150 градусов. В области этой пирамиды Leap Motion «видит» все движения и пересылает их программному обеспечению, которое преобразует данные и сигналы в координаты и сообщения. Программное обеспечение способно распознать как простые команды — жесты: виртуальные прикосновения, нажатия, так и сложные продолжительные движения: масштабирование, перемещение, вращение, рисование различных геометрических фигур. Таким образом, само устройство не выполняет никаких вычислений и преобразований, отдавая все на откуп программе, работающей на компьютере, которая, удаляя шумы изображения, строит модели рук и пальцев — указателей (рис. 12.2).

Имея начало координат в центре устройства, Leap Device интерпретирует оси координат следующим образом: отрицательная X расположена слева от устройства, соответственно, положительная — справа. Координата Y растет вверх и не имеет отрицательных значений, ибо Leap Motion «видит» объекты, начиная с 25-и миллиметров выше себя. Положительная Z располагается в направление к пользователю, тогда как отрицательная — к экрану.

В настоящее время контроллер Leap Motion используется в исследовательских целях и для продвижения других, связанных с дополненной реальностью проектах. Сенсор устанавливается на переднюю часть шлема виртуальной реальности Oculus Rift (рис. 12.3), благодаря чему он «видит» перед собой руки использующего его персонажа, в смысле, человека.

Рис. 12.3. Leap Motion + Oculus Rift

Таким образом, происходит полное погружение в виртуальный мир через осязание, игрок может управлять виртуальным пространством с помощью своих рук. Здесь многое зависит от программного обеспечения, потому что оно осуществляет все основные вычисления и управление. Существуют так же другие исследовательские проекты, например, воспроизведение звуков и музыки с помощью Leap Motion, подобно электронным музыкальным инструментам (таким как: терменвокс (https://ru.wikipedia.org/wiki/Электронные_музыкальные_инструменты)). Данные с сенсора можно обрабатывать в разных целях, поэтому Leap Motion может быть использован в широчайшем спектре приложений. Главное, чтобы программное обеспечение корректно анализировало получаемую с сенсора информацию, которая имеет довольно широкий диапазон различных состояний.

Кроме того, после выхода устройства разработчики компьютерных игр сразу обратили на него внимание, поскольку было очевидно, что Leap Motion, являясь продолжателем дела сенсора Kinect, станет таким же успешным и популярным в среде игроков. А значит, нужны игры, поддерживающее это устройство.

Для работы с сенсором Leap Motion подойдет компьютер со следующей конфигурацией:

— Процессор: AMD Phenom II или Intel Core i3/i5/i7
— 2 ГБ оперативной памяти
— порт USB 2-й версии

— операционная система: Windows 7/8/10 или Mac OS X 10.7 и новее

Обратите внимание, когда работает контроллер Leap Motion, операционная система считает, что работает web-камера.

2 Программное обеспечение для Leap Motion

Leap Motion SDK имеет исключительно бурное развитие: новые версии выходят с завидной регулярностью, за сравнительно недолгую историю своего существования уже вышла полноценная вторая версия инструментального набора, а так же его модификации. Мы будем использовать самую последнюю на момент написания книги версию sdk.

Основное отличие второй версии sdk от первой, это новая система слежения за скелетом верхних конечностей, в эту систему включена обработка дополнительной информации о костях рук и пальцев, возможность предсказания расположения невидимых для устройства костей и построение моделей рук, когда полностью конечности не видны.

Как следовало того ожидать: Leap Motion SDK работает на всех распространенных платформах: Windows NT, OS X и Linux. Так как, читаемая вами в данный момент книга в большинстве своем посвящена разработке игр для Windows, мы будем рассматривать применения устройства Leap Motion в этой операционной системе. Тем не менее с таким же успехом вы можете применять Leap Motion Controller в двух других операционных системах.

Для начала работы с контроллером Leap Motion, предварительно зарегистрировавшись на сайте производителя устройства (https://www.leapmotion.com), из раздела Developers/Desktop SDK (https://developer.leapmotion.com/v2) скачайте zip-архив Leap_Motion_SDK_Windows_2.3.1.zip. После распаковки архива вы обнаружите папку LeapDeveloperKit_2.3.1+31549_win, в которой находятся файл Leap_Motion_Installer_release_public_win_x86_2.3.1+31549_ah1886.exe, текстовый файл и каталог LeapSDK, включающий все необходимые библиотеки и api для разработки приложений, работающих с устройством Leap Motion. Вдобавок в этой папке находятся документация и сэмпл с проектами под разные языки и системы программирования. Файл Leap_Motion_Installer_release_public_win_x86_2.3.1+31549_ah1886.exe содержит программу для установки драйверов, документации, примеров и приложения Leap Motion App Home. Мастер установки включает пару простых шагов (рис. 12.4).

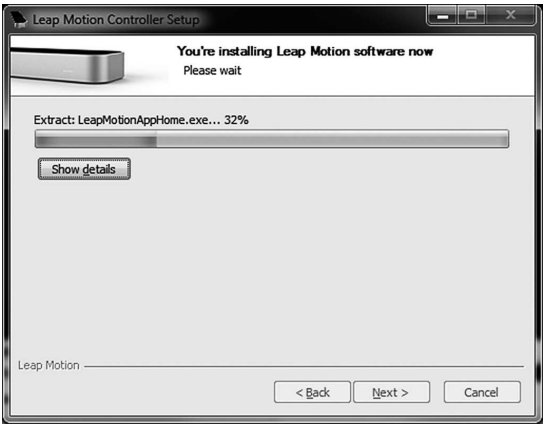

Рис. 12.4. Установка Leap Motion SDK

В конце установки будет предложено запустить Leap Motion Experience. После этого если у вас открыт браузер FireFox, его будет предложено закрыть для включения поддержки Leap Motion в web-приложениях. Затем перед запуском Leap Motion App Home появится вопрос: в каком режиме мы собираемся использовать сенсор: в настольном или режиме виртуальной реальности.

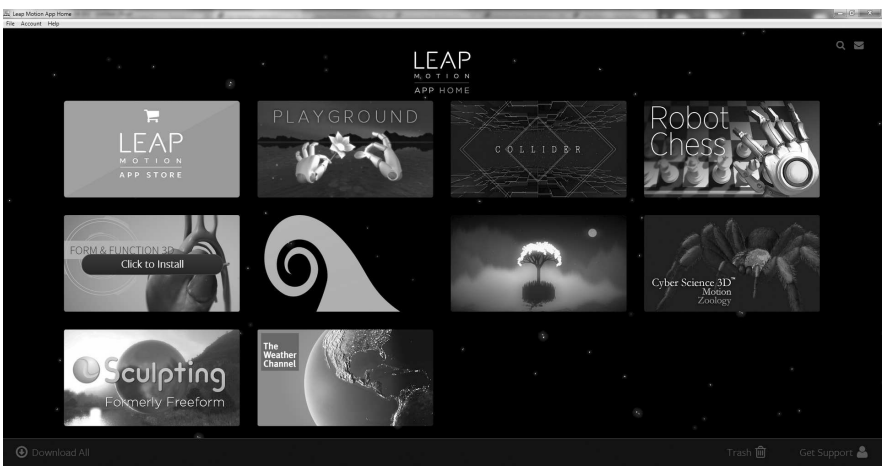

Рис. 12.5. Leap Motion App Home

Leap Motion App Home представляет собой клиент для магазина приложений, созданных для работы с контроллером Leap Motion. Как и на других площадках цифровой дистрибуции, в App Home вы можете размещать свои приложения и распространять их оттуда на бесплатной или платной основах.

Также после завершения установки программного обеспечения для Leap Motion, автоматически запуститься драйвер и в виде сервиса «поселится» на панели задач. В главном меню появится папка, содержащая 3 новых приложения: Leap Motion Control Panel — приложение для настойки драйвера устройства, демо-программа Leap Motion Visualizer (рекомендую начать с нее, чтобы проверить работоспособность сенсора и его возможности по слежению за руками) и клиент для магазина Leap Motion App Home.

Если ранее контроллер не был подключен, самое время подключить его. Значок на панели задач подсветится зеленым цветом. В результате щелчка правой кнопкой на нем откроется меню, содержащее 5 пунктов. Первый пункт — Launch App Home запускает одноименного оконного клиента. По умолчанию в нем присутствуют 9 демо-приложений и одна ссылка, переводящая в Leap Motion App Store. Каждая из демонстраций раскрывает возможности Leap Motion:

1. В PlayGround пользователь прикрепляет кубики к роботам в качестве головы;

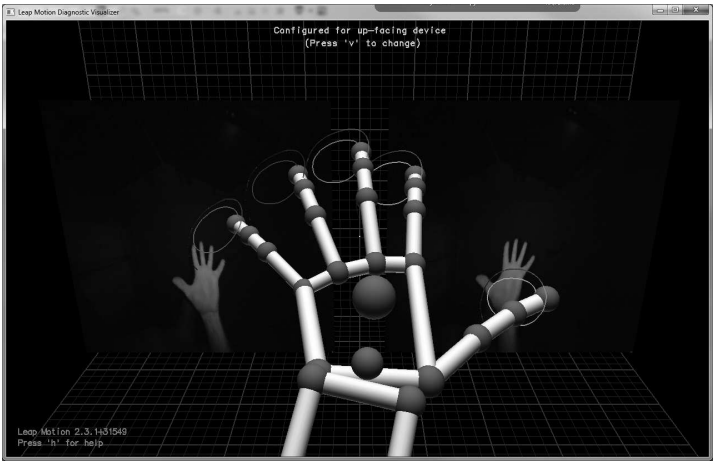

Рис. 12.6. Visualizer

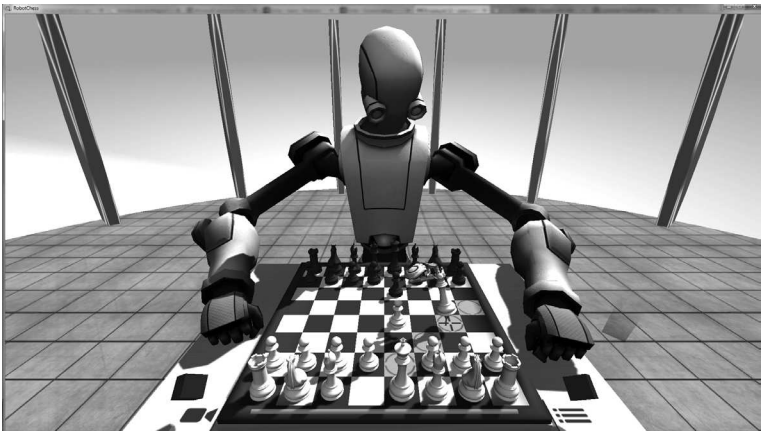

Рис. 12.7. RobotChess

2. Collider — перемещение в трехмерном пространстве, при выполнении трюков руками, показанными на экране, движение ускоряется;

3. RobotChess — великолепная игра в шахматы с роботом, перемещение фигурок осуществляется с помощью контроллера, как будто вы играете в настольную игру по-настоящему;

4. Form & Function 3D — приложение для изучения анатомии;

5. Water Waves Screensaver — представляет собой хранитель экрана, который из меню можно установить в Windows;

6. Kyoto — приложение, в котором под спокойную музыку, вам предлагается взаимодействовать с природой с помощью сверхъестественных возможностей, например, поймать звезду;

7. В Cyber Science 3D пользователю предоставляется возможность рассмотреть паука — тарантула со всех углов с возможностью приближения и отдаления камеры;

8. С помощью приложения Sculpting Formerly Freeform можно из глины лепить объекты, при этом у вас есть большой выбор инструментов для создания красивых скульптур. Вдобавок многие параметры приложения легко настраиваются: фон, на котором происходит создание скульптуры, размер, скорость вращения моделируемого объекта и другое.

9. The Weather Channel — позволяет смотреть погоду на карте мира, точнее, на глобусе — модели планеты, вращая, увеличивая и уменьшая масштаб последней.

Следующий пункт меню — Settings открывает окно для настройки устройства. Это окно включает 4 вкладки: на странице Generals производятся основные настройки:

— Allow Apps — разрешить или запретить устройству взаимодействовать с web-приложениями, которые поддерживают Leap Motion (для этого используется HTML5+JavaScript);

— Allow Background Apps — включить или выключить возможность получения сигналов от устройства приложениями, работающим в фоне;

— Allow Images — включить отправление изображений для приложений, запросившие их; этот режим нужен для приложений виртуальной реальности, использующие режим изображения рук;

— Automatic Power Saving — включить (при возможности) переход в энергосберегающий режим;

— Send Usage Data — автоматическая передача статистики устройства;

— Launch on Startup — во включенном состоянии позволяет запускать сервис Leap Motion Controller и апплет Leap Motion Control Panel во время загрузки операционной системы;

— Automatic Interaction Height — настройка наименьшей высоты над устройством, при котором оно «видит» руки и пальцы (указатели);

— Automatically Install Updates — согласиться на автоматическое обновление.

На странице «Tracking» присутствуют 4 параметра:

— Robust Mode — во включенном состоянии позволяет производить лучшее слежение в освещенных условиях;

— Tool Tracking — слежение за указателями (отличными от пальцев) в области вида устройства;

— Hand Tracking — слежение за руками и пальцами в области вида устройства;

— Auto-orient Tracking — автоматически выбирает ориентацию устройства в зависимости от положения рук в области видимости;

Следующая вкладка посвящена диагностированию и устранению неполадок, здесь присутствуют кнопки для вызова функций:

— просмотра лога ПО;

— диагностический визуализатор;

— повторная калибровка устройств;

— вывести отчет о проблемах ПО;

— возврат к настройкам по умолчанию;

Ниже находятся 2 переключателя:

— Low Resource Mode — включение этого флажка уменьшает производительность и полосу пропускания контроллера Leap Motion для повышения надежности для слабых компьютеров;

— Avoid Poor Performance — в этом режиме слежение приостанавливается, когда появляются проблемные условия;

Последняя вкладка просто сообщает сведения об устройстве и обслуживающим его программном обеспечении.

Щелчком по пункту Visualizer открывается демонстратор, в нем можно посмотреть, как устройство «видит» конечности. То есть если переместить руки над активной областью устройства, приложение отобразит их в виртуальном пространстве.

Пункт Pause Tracking приостанавливает слежение.

Quit — прекращает исполнение сервиса.

Leap Motion SDK написан на C++, но, благодаря SWIG, имеет поддержку многих распространенных компилируемых и интерпретируемых языков, среди которых: C# (вместе с фреймворками .Net и Mono плюс движком Unity 3D), Objective-C, Java, Python, JavaScript. SWIG, являясь свободным инструментом с открытым исходным кодом, играет роль генератора связующего кода между C++ и другими языками.

Для взаимодействия между клиентским компьютером и контроллером используется TCP соединение, при котором открываются порты 6437, 6438, 6439, для корректной работы устройства необходимо проследить, чтобы они не блокировались фаерволом.

Leap Motion SDK позволяет разрабатывать приложения двух видов: использующие нативный интерфейс (клиентские приложения) и через интерфейс WebSockets (соответственно, web-приложения, работающие в среде браузера). Первые — для работы — получения данных от контроллера используют динамическую библиотеку — конкретную для определенной операционной системы, она подключается к драйверу устройства и предоставляет сервис верхнему уровню. Тогда как вторые — получают данные через сервер WebSockets локального компьютера в виде сообщений формата JSON. В этом случае используется JavaScript + Open Source надстройка LeapJS, и для управления устройством приложение может передавать конфигурационные сообщения через сервер WebSockets обратно контроллеру.

Между тем кроме Leap Motion SDK для сенсора Leap Motion существует еще один пакет программ — Orion. По сути, это развитие Leap Motion SDK, только с другим названием. Он, в большей степени, предназначен для разработки приложений виртуальной реальности. Это тот случай, когда сенсор Leap Motion устанавливается на переднюю панель шлема виртуальной реальности Oculus Rift.

На момент написания книги, Orion находился в стадии бета тестирования, о чем говорит слово Beta в названии. Кроме того, в данное время Orion предназначен только для операционной системы Windows. Чтобы скачать дистрибутив перейдите на страницу Get Started (https://developer.leapmotion.com/get-started) и щелкните по кнопке «Orion Beta (VR)». Начнется скачивание zip-архива. В нем находится папка LeapDeveloperKit_3.1.0+39495_win с аналогичным прошлому пакету содержимым, в том числе с программой — инсталлятором для установки драйверов, документации и примеров SDK. Для начала я рекомендую удалить прошлую версию SDK. Уже затем приступить к установке новой. Ее инсталляция ничем не отличается от предыдущей.

Вот как выглядит Visualizer из новой версии SDK (рис. 12.8).

В новой версии Visualizer не ожидает, что руки пользователя буду над устройством, здесь он полагает, что расположен примерно на голове у пользователя, поэтому кости рук строит с вида сверху. На скриншоте показан вывод, того, что видит сенсор. На самом деле, руки проецируются в виртуальную реальность, представляемую шлемом Oculus Rift.

Рис. 12.8. Visualizer-VR

Как мы обсуждали выше: нововведения новой версии SDK предназначены для создания приложений виртуальной реальности, при этом возможности десктопной версии остались прежними. К сожалению, на момент написания книги, Oculus Rift по-прежнему не поступил в продажу в потребительской версии, поэтому у меня его нет. Из-за этого я собственноручно не тестировал Leap Motion со шлемом виртуальной реальности.

Для наших целей Desktop-версии хватит с избытком.

3 Использование Leap Motion в играх

Именно в игровых приложениях использование Leap Motion находит наилучшее применение. Спустя некоторое время после выхода устройства, увидев его потрясающие возможности, а главное заметный интерес среди пользователей, разработчики ведущих игровых движков стали добавлять к своим основным продуктам поддержку контроллера. Первым движком, поддерживающим Leap Motion, стал Unity 3D. Затем поддержкой обзавелся Torque 3D, уже являясь Open Source решением. После этого на основе данного решения была реализована поддержка в движке Torque 2D. А некоторое время спустя, с помощью Leap Motion стало возможно управлять играми, созданными на движке Unreal Engine 4.

Для меня, а так же многих других инди игроделов, семейство движков Torque является оптимальным решением. И в свете современных событий, таких как: распространение гейминга на мобильных платформах, более востребованными стали двумерные игры, так как их больше любят казуальные игроки, составляющие большую часть играющих на миниатюрных устройствах. Много жанров двумерных игр можно легко адаптировать для управления с помощью контроллера Leap Motion.

Hidden Objects

Например, можно проапгрейдить жанр Hidden Objects. В магазине цифровой дистрибуции Leap Motion App Sore подобных игр мало, отсюда вытекает привлекательная ниша, которую можно с успехом занять. Первая волна популярности этого жанра пришлась на 90-е годы прошлого века, когда поиск и кликанье объектов было реализовано в популярных тогда приключенческих играх (типа: «Сломанный меч», «Neverhood», «Братья пилоты», «Петька», «Как достать соседа» и многие другие). Потом с началом следующего десятилетия первое место заняли трехмерные боевики, а приключенческий жанр вместе с Hidden Objects

утратили свою былую популярность. Но еще через десятилетие вместе с айфонами и другими мощными мобильными устройствами Hidden Objects выделился в отдельный жанр и вернулся с новой силой и популярностью. Особенно стоит выделить такие компании: BigFish, Playrix и GameInsight, которые начали свой игровой бизнес с таких игр и уверенно выросли до размеров гигантов индустрии, продолжая в увеличивающихся количествах билдить Hidden Objects игры и оставаться в лидерах. Однако в мобильном гейминге наблюдается подобный произошедшему на PC процесс: мобильный игровой рынок потихоньку пока неуверенно занимают трехмерные игры.

Однако появление новых видов устройств вполне может поспособствовать продолжению жизни жанра на платформе, для которой эти устройства выпускаются, в данном случае: PC и Mac.

Как раз с помощью Torque 2D можно подготовить один исходник, который с минимальными изменениями можно собрать на другой программно-аппаратной платформе.

4 Разработка игры с Leap Motion

С игровым жанром и используемыми инструментами определились. В таком случае начнем разрабатывать нашу новую игру с поддержкой контроллера Leap Motion. Давайте графические материалы возьмем из стандартного примера, идущего в составе Torque 2D, про аквариум с рыбками.

Для этого я подготовил новый пример LeapObjects (см. в материалах к книге). На этот раз нам подойдет не модифицированная копия движка. Арт находится в подпапке Content каталога modules корневой папки игры, а игровые скрипты — в подкаталоге LeapObjectsGame. В поддиректории AppCore по традиции лежат скрипты для инициализации графической, звуковой подсистем, объявления параметров по умолчанию и констант. Инициализация и подготовка происходят по стандартному сценарию, описанному в заготовке игры.

4.1 LeapObjectsGame

Создание объекта игры происходит в методе LeapObjectsGame::create, находящемся в файле main.cs на уровне выше каталога LeapObjectsGame.

```
function LeapObjectsGame::create( %this )
{
   exec("./scripts/common/scenewindow.cs");
```

```
    exec("./scripts/common/scene.cs");
    exec("./scripts/gameObjs/background.cs");
    exec("./scripts/gameObjs/cursor.cs");
    exec("./gui/guiProfiles.cs");
    exec("./scripts/InputManager.cs");
    exec("./scripts/gameObjs/aquarium.cs");
    exec("./scripts/gameObjs/fish.cs");

    createScene();
    createSceneWindow();
    mySceneWindow.setScene(myScene);
    new ScriptObject(InputManager);
    mySceneWindow.addInputListener(InputManager);
    InputManager.Init_controls();

   LeapObjectsGame.maxFish = 10;

   myScene.setLayerSortMode( 15, batch );

   LeapObjectsGame.reCreate();

   %this.selectedObjects = new SimSet();
   %this.add(%this.selectedObjects);
}
```

Внутри данного метода подключаются файлы с исполняемым кодом, затем происходят вызовы функций для создания среды игры (этот код мы рассматривали при обсуждении заготовки), в том числе создается элемент управления — InputManager. Свойство максимального количества рыб устанавливаем в значение 10. С помощью метода setLayerSortMode объекта myScene задаем порядок «пакетной» сортировки для вывода визуальных элементов сцены и тем самым уменьшения количество вызовов перерисовки. Далее создаем контейнер для выбранных объектов — selectedScene:

```
%this.selectedObjects = new SimSet();
%this.add(%this.selectedObjects);
```

В деструкторе класса LeapObjectsGame уничтожается объект управления:

```
function LeapObjectsGame::destroy( %this )
{
   InputManager.destroy();
   InputManager.delete();
}
```

Из конструктора класса LeapObjectsGame вызывается функция reCreate. В ней происходит подготовка игры, создаются геймплейные объекты:

```
function LeapObjectsGame::reCreate(%this)
{
   // очистить сцену
   myScene.clear();

   //создать курсор
   $cursor = cursor::createCursor(%this);//глобальный
//объект
   Canvas.hideCursor();
   buildAquarium(myScene);//создание аквариума
   // сбросить счетчик
   %this.currentFish = 0;

   LeapObjectsGame.createCircleSprite(%this);
   // запустить таймер
   LeapObjectsGame.startTimer( "spawnFish", 500 );
}
```

Очищается сцена, создается пользовательский курсор, который заменяет собой системный, делая последний невидимым. Вызывается buildAquarium для построения аквариума. Число рыб обнуляется. С помощью функции createCircleSprite создается спрайт для круга. Следующим действием запускается таймер. Это выполняется с помощью метода startTimer глобального объекта игры. В параметрах ему передаются: имя метода, количество миллисекунд, через которое этот метод надо периодически вызывать.

Рассмотрим метод createCircleSprite. Как упоминалось выше: в нем создается спрайт, который образуется в результате действия пользователя — кругового вращения пальцем. Метод выглядит весьма типично:

```
function LeapObjectsGame::createCircleSprite( %this )
{
   %circle = new Sprite();
   %circle.Position = "-10 5";
   %circle.setBodyType("Kinematic");
   %circle.Size = 5;
   %circle.Image = "Content:circleThree";
   %circle.Visible = false;
   %circle.AngularVelocity = 180;
   %this.circleSprite = %circle;
   myScene.add( %circle );
}
```

Здесь есть 2 момента, которые стоит отметить: задание типа «Kinematic» для физического взаимодействия и присвоение кастомному свойству circleSprite самого себя (создаваемого объекта) для дальнейшего использования.

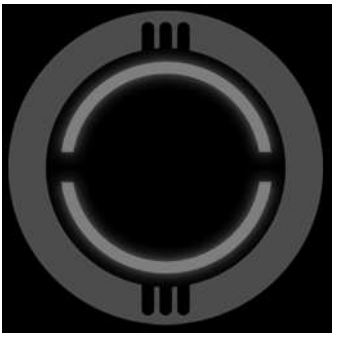

Рис. 12.9. Круг

Метод для создания рыб — spawnFish вызывается по тику таймера — через каждые 500 миллисекунд. Этот метод является частью класса LeapObjectsGame по той причине, что метод startTimer может принять ссылку только на функцию того же класса, для которого вызывается.

```
function LeapObjectsGame::spawnFish(%this)
{
   %position = getRandom(-55, 55) SPC getRandom(-20,
20);
   %index = getRandom(0, 5);
   %anim = getUnit(getFishAnimationList(), %index,
",");
   %fishSize = getFishSize(%anim);
   %fish = new Sprite()
   {
      Animation = %anim;
      class = "FishClass";
      position = %position;
      size = %fishSize;
      SceneLayer = "15";
      SceneGroup = "14";
      minSpeed = "5";
      maxSpeed = "15";
      CollisionCallback = true;
   };
   %fish.setCollisionGroups( 15 );
   %fish.createPolygonBoxCollisionShape(%fishSize);
   %fish.setDefaultDensity( 1 );
   %fish.setBodyType("dynamic");
   myScene.add( %fish );
   %this.currentFish++;
   if ( %this.currentFish >= %this.maxFish)
   {
      LeapObjectsGame.stopTimer();
   }
}
```

Метод начинается с получения случайных координат для последующего размещения спрайта рыбы. В переменную %index сохраняется случайное число из диапазона от 0 до 5. На следующем шаге в переменную %anim помещается имя анимации, взятое из списка, при этом для получения элемента списка используется функция getUnit, она принимает 3 параметра: список элементов, индекс элемента, набор символов, встретив один из которых, функция прекратит выполнение, возвращая набор символов от начала элемента, до встретившегося контрольного символа. Нужный номер элемента находится путем перебора всей строки символов, где контрольный символ считается разделителем между элементами. Список элементов возвращает функция getFishAnimationList:

```
function getFishAnimationList()
{
   %list = "Content:angelfish1Anim" @ "," @
"Content:angelfish2Anim" @ "," @
"Content:butterflyfishAnim";
   %list = %list @ "," @ "Content:pufferfishAnim" @ ","
@ "Content:rockfishAnim" @ "," @ "Content:seahorseAnim";
   %list = %list @ "," @ "Content:triggerfish1Anim" @
"," @ "Content:eelAnim";
}
```

Индекс — это случайно найденное выше число, а контрольный символ — это «,». В итоге получаем одно из имен ассетов анимаций. Затем с помощью функции getFishSize, передав ей имя ассета, получаем размер рыбки — анимационного кадра: %fishSize = getFishSize(%anim);. В функции прописаны заранее вычисленные размеры для всех ассетов, поэтому, получая имя последнего, ей остается только выбрать параметры для ассета с подходящим названием.

```
function getFishSize(%anim)
{
   switch$(%anim)
   {
      case "Content:angelfish1Anim":
      %fishInfo = "15 15";
      case "Content:angelfish2Anim":
      %fishInfo = "15 15";

      case "Content:butterflyfishAnim":
      %fishInfo = "15 15";
```

```
      case "Content:pufferfishAnim":
      %fishInfo = "15 15";

      case "Content:rockfishAnim":
      %fishInfo = "15 7.5";

      case "Content:seahorseAnim":
      %fishInfo = "7.5 15";

      case "Content:triggerfish1Anim":
      %fishInfo = "15 15";
      case "Content:eelAnim":
      %fishInfo = "7.5 3.75";
   }
   return %fishInfo;
}
```

Теперь можно переходить к созданию спрайта, используя найденные параметры: анимационный ассет, позицию, размер, а так же дополнительные: класс — «FishClass», слои уровня и группы, минимальная, максимальная скорости, обратный вызов при столкновении — true. Также мы устанавливаем для спрайта рыбы другие свойства: группу коллизии, тип взаимодействия — динамический, плотность объекта и создаем полигон столкновения равный размеру спрайта. После чего добавляем созданный спрайт в стек сцены. Потом увеличиваем глобальную переменную currentFish на 1. В конце функции проверяем: равна ли глобальная переменная %maxFish глобальной переменной %currentFish. Если да, значит, достигнут лимит рыб, и нам нужно отключить таймер, который вызывает рассматриваемую функцию:

```
LeapObjectsGame.stopTimer();
```

Для анимаций рыб служат 8 изображений, каждое из которых поделено на 4 кадра (рис. 12.10).

Для загрузки изображения используется такой taml-код:

```
<ImageAsset
   AssetName="angelFishImage"
   ImageFile="angelfish1.png"
   CellCountX="2"
   CellCountY="2"
   CellWidth="256"
   CellHeight="256" />
```

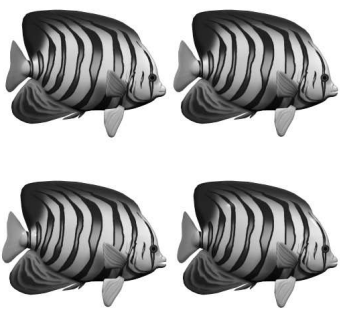

Рис. 12.10. Изображение одной из рыб

В свою очередь, этот ассет используется анимационным ассетом:

```
<AnimationAsset
   AssetName="angelfish1Anim"
   Image="@asset=Content:angelFishImage"
   AnimationFrames="0 1 2 3 3 2 1 0"
   AnimationTime="0.4" />
```

Как видите: по нумерации воспроизводимых кадров, сначала идет показ фреймов в порядке возрастания — от 0 до 3, затем в порядке убывания — от 3 до 0. Каждый следующий кадр показывается через 0,4 секунды.

Таким образом загружаются все рыбы в игру.

4.2 Аквариум

Посмотрим, как организована графически-анимационная часть игры. Она состоит из изображений фона, переднего плана, курсора, рыб и анимации последних. По традиции фон представлен классом background и находится в файле background.cs подкаталога gameObjs. Метод для его создания — createBackground выглядит следующим образом:

```
function background::createBackground()
{
   %background = new Sprite();
   %background.class = "background";
   %background.setBodyType( static );
   %background.Position = "0 0";
   %background.Size = "100 75";
   %background.Image = "Content:background";
   %background.setCollisionSuppress();
   %background.setAwake( false );
   %background.setActive( false );
   %background.setSceneLayer(30);
   %background.setUseInputEvents(true);
   myScene.add(%background);

   return %background;
}
```

Как видно: фон представляет собой спрайт, и в этом нет ничего удивительного и нового. Все свойства стандартные, и с ними мы уже встречались: статический тип, позиция — в центре экрана, размер: 100 × 75, слой — 30, изображение загружается из ассета «Content:backround», который загружает соответствующую текстуру:

```
<ImageAsset
   AssetName="background"
   ImageFile="background.jpg" />
```

Функция обратного вызова onTouchMoved, описанная в файле background.cs, вызывается из движка во время движения курсора мыши или указателя (пальца), когда управление происходит с помощью Leap Motion, тем самым она передвигает кастомный курсор на экране.

```
function background::onTouchMoved(%this, %touchID,
%worldPosition)
{
   $cursor.setPosition(%worldPosition.x + 1.8 SPC
%worldPosition.y - 1.8);
}
```

Фон вместе с передним слоем и границами составляют аквариум, описанный в файле aquarium.cs. Вызов метода для построения фона — createBackground класса background происходит из функции buildAquarium. Она вызывается из метода reCreate главного класса игры LeapObjectsGame. Кроме того, buildAquarium принимает ссылку на объект класса Scene:

```
function buildAquarium(%scene)
{
   // фон
   %background = background::createBackground();

   // дальние камни
   %farRocks = new Sprite();
   %farRocks.setBodyType( "static" );
   %farRocks.setPosition( 0, -7.5 );
   %farRocks.setImage( "Content:rocksfar" );
   %farRocks.setSize( 100, 75 );
   %farRocks.setCollisionSuppress();
   %farRocks.setAwake( false );
   %farRocks.setActive( false );
   %farRocks.setSceneLayer(20);
   %scene.add( %farRocks );

   // ближние камни
   %nearRocks = new Sprite();
   %nearRocks.setBodyType( "static" );
   %nearRocks.setPosition( 0, -8.5 );
   %nearRocks.setImage( "Content:rocksnear" );
   %nearRocks.setSize( 100, 75 );
```

```
   %nearRocks.setCollisionSuppress();
   %nearRocks.setAwake( false );
   %nearRocks.setActive( false );
   %nearRocks.setSceneLayer(10);
   %scene.add( %nearRocks );

   addAquariumBoundaries( %scene, 100, 75 );
}
```

После создания объекта background происходят создания отдаленного и ближнего слоев камней. Они являются спрайтами, на которые загружаются текстуры, имеющие полупрозрачные области. Также для них устанавливаются 20-й и 10-й слои соответственно. При этом вызов функции setCollisionSuppress исключает объект, для которого она была вызвана (в данном случае спрайты камней) из симуляции столкновения. Ассеты для загрузки текстур отличаются только именами загружаемых файлов. После создания каждого из спрайтов они добавляются в сцену, посредством ссылки на нее.

В конце метода вызывается функция addAquariumBoundaries, которая принимает ссылку на объект класса Scene и размеры области вывода по обеим осям. В ней происходит создание границ экрана:

```
function addAquariumBoundaries(%scene, %width,
%height)
{
//вычисление ширины и высоты для построения границ
   // границы должны быть больше аквариума
   %wrapWidth = %width * 1.5;
   %wrapHeight = %height * 1.5;
   %scene.add( createOneAquariumBoundary( "left",
-%wrapWidth/2 SPC 0,  5 SPC %wrapHeight) );
   %scene.add( createOneAquariumBoundary( "right",
%wrapWidth/2 SPC 0,   5 SPC %wrapHeight) );
   %scene.add( createOneAquariumBoundary( "top", 0 SPC
-%wrapHeight/2, %wrapWidth SPC 5 ) );
   %scene.add( createOneAquariumBoundary( "bottom", 0
SPC %wrapHeight/2,  %wrapWidth SPC 5 ) );
}
```

В начале функции вычисляются ширина и высота, используемые для границ. Для этого начальные ширина и высота умножаются на 1.5. Границы строятся так, чтобы быть немного дальше реальных размеров области вывода (окна). Границы нужны, чтобы определять столкновения с рыбками, после чего разворачивать их в обратную сторону. И, чтобы это

происходило не на виду у пользователя, границы строятся дальше пределов обзора.

Отдельная граница строится с помощью функции createOneAquariumBoundary. Четыре вызова этой функции находятся в конце addAquariumBoundaries. Когда результат, возвращенный createOneAquariumBoundary, чем является созданная граница, является входящим параметром метода add объекта Scene.

Функции createOneAquariumBoundary передаются 3 параметра: название стороны, позиция и размер:

```
function createOneAquariumBoundary(%side, %position,
%size)
{
   %boundary = new SceneObject()
{
class = "AquariumBoundary";
};

   %boundary.setSize( %size );
   %boundary.side = %side;
   %boundary.setPosition( %position );
   %boundary.setSceneLayer( 1 );
   %boundary.createPolygonBoxCollisionShape( %size );
   %boundary.setSceneGroup( 15 );
   %boundary.setCollisionCallback(false);
   %boundary.setBodyType( "static" );
   %boundary.setCollisionShapeIsSensor(0, true);
   return %boundary;
}
```

В теле функции создается объект класса SceneObject. Это самый простой класс, который может быть добавлен в стек сцены, плюс объекты, созданные на его основе, являются невидимыми. Зато порожденные объекты обладают физическими свойствами. Это нам как раз и надо. Для вновь созданного объекта задаем новый класс «AquariumBoundary», устанавливаем размер, добавляем переменную side, значение которой берется из параметра, настраиваем позицию, номер слоя и группы сцены, ставим статический тип, выключаем возможность реакции на столкновение, при этом с помощью метода setCollisionShapeIsSensor оставляем ему возможность сообщать о произошедшем столкновении.

Когда границы будут сформированы, создание аквариума завершается.

4.3 FishClass

В разделе 1 мы написали функцию для создания спрайтов рыб, которым присваивается класс FishClass. После создания объектов — рыб этот класс занимается их управлением.

После появления на экране для каждого объекта класса FishClass вызывается функция onAdd:

```
function FishClass::onAdd(%this)
{
   %this.setSpeed();

   if (getRandom(0, 10) > 5)
   {
      %this.setLinearVelocityX(%this.speed);
      %this.setFlipX(false);
   }
   else
   {
      %this.setLinearVelocityX(-%this.speed);
      %this.setFlipX(true);
   }
}
```

Для текущего объекта из нее вызывается метод setSpeed, который между значениями minSpeen и maxSpeed случайным образом выбирает значение для скорости перемещения рыбы.

```
function FishClass::setSpeed(%this)
{
    %this.speed = getRandom(%this.minSpeed, %this.
maxSpeed);
}
```

Возвратившись в метод onAdd, мы встречаем условный оператор. Сначала в нем случайно выбирает число от 0 до 10, если выбранное значение больше 5, тогда текущему спрайту назначается движение с положительным приращением по оси X:

```
%this.setLinearVelocityX(%this.speed);
```

Вдобавок объект возвращается к начальному повороту вдоль оси X (по умолчанию на всех изображениях рыбы направлены слева направо), для этого в метод поворота передается нулевое значение:

```
%this.setFlipX(false);
```

Когда значение меньше или равно 5, происходят обратные действия: скорость движения объекта по оси X — отрицательная, а картинка объекта зеркально отражается вдоль оси X:

```
%this.setLinearVelocityX(-%this.speed);
%this.setFlipX(true);
```

После этого рыбка начинает плавно перемещаться по экрану — аквариуму.

Когда спрайт перемещается по экрану, он отображает анимацию, и с ним не происходят никакие события, пока он не достигает границы аквариума. В таком случае происходит обратный вызов onCollision. Мы уже неоднократно сталкивались с обработчиком этого события. Он принимает: указатели на столкнувшие объекты и список подробностей столкновения. Внутри обработчика проверяется класс 2-го объекта (поскольку класс 1-го — определенно, FishClass), если он является «AquariumBoundary» (граница аквариума), тогда для текущей рыбы выполняется метод recycle. Он получает имя стороны, с которой произошло столкновение.

```
function FishClass::recycle(%this, %side)
{
   %this.setSpeed();
   %layer = getRandom(0, 5);
   %this.setLinearVelocityY(getRandom(-3, 3));
   %this.setPositionY(getRandom(-15, 15));
   %this.setSceneLayer(%layer);
   if (%side $= "left")
   {
      %this.setLinearVelocityX(%this.speed);
      %this.setFlipX(false);
   }
   else if (%side $= "right")
   {
      %this.setLinearVelocityX(-%this.speed);
      %this.setFlipX(true);
   }
}
```

В теле обработчика с помощью метода setSpeed случайно выбирается скорость, кроме того, случайным образом выбирается число от 0 до 5, которое послужит номером слоя для текущего объекта; выбираем и устанавливаем приращение по оси Y от -3 до 3, также случайно меняем позицию спрайта рыбы по оси Y (от -15 до 15), устанавливаем номер

экранного слоя. Затем выполняется проверка: с какой стороной: left или right произошло столкновение. В первом случае выбранная скорость устанавливается в качестве положительного приращения по оси X (плывет слева направо), при этом отменяется зеркальное отображение (картинка возвращается в начальный вид), во втором случае устанавливается отрицательное приращение (справа налево), а изображение на спрайте зеркально отражается.

Это вся работа, возложенная на класс управление рыбками — FishClass.

4.4 Cursor

Следующим классом визуальных объектов является cursor. Он располагается в файле cursor.cs и содержит только одну функцию — для создания:

```
function cursor::createCursor(%this)
{
   %size = 4;
   %cur = new Sprite()
   {
  class = "cursor";
  Image = "Content:cursor";
  size = %size SPC %size;
  SceneLayer = 0;
  SceneGroup = 0;
  Position = "0 0";
  BodyType = static;
   };
   myScene.add(%cur);

   return %cur;
}
```

Думаю, в дополнительных комментариях этот код не нуждается — все очень знакомо.

4.5 Менеджер управления

Управление для LeapObjects мы организуем с 2-х устройств: клавиатуры и сенсора Leap Motion. Модуль, в котором находится код для реализации управления, называется InputManager.cs.

Какие геймплейные функции мы можем добавить?

1. Остановку рыб. Другими словами, нам нужен способ для остановки движения рыб.

2. Дополнительно нам нужен способ для удаления остановленных рыб.

Какие задачи мы возложим на управление? Для клавиатуры мы реализуем только реакцию на нажатие клавиши Esc — закрытие приложение — все как обычно. На Leap Motion мы возложим следующие задачи:

1. Рисование пальцем (или другим указателем) круга (в воздухе), экземпляр которого будет создан в игре — на экране монитора.

2. Обработка виртуального нажатия в результате чего остановленные рыбы будут удалены.

Функция, которая служит для инициализации элемента управления, называется InputManager::Init_controls, она вызывается из конструктора класса LeapObjectsGame. Сама функция выглядит так:

```
function InputManager::Init_controls(%this)
{
   new ActionMap(keyInput);
   new ActionMap(leapInput);
   keyInput.bindCmd(keyboard, "escape", "quit();",
"");
   leapInput.bindObj(leapdevice, circleGesture,
reactToCircleBuilder, %this);
   leapInput.bindObj(leapdevice, screenTapGesture,
reactToScreenTapBuilder, %this);
   leapInput.bindObj(leapdevice, swipeGesture,
reactToSwipeBuilder, %this);
   leapInput.bindObj(leapdevice, keyTapGesture,
reactToKeyTapBuilder, %this);
   keyInput.push();
   leapInput.push();

   %this.handPosDeadzone = "-1.0 1.0";
   %this.handRotDeadzone = "-1.0 1.0";
   %this.fingerPosDeadzone = "-1.0 1.0";
   %this.enableSwipeGesture = true;
   %this.enableCircleGesture = true;
   %this.enableScreenTapGesture = true;
   %this.enableKeyTapGesture = true;
   %this.enableHandRotation = true;
   %this.enableFingerTracking = true;

   // инициализировать Leap Motion
  initLeapMotionManager();
  enableLeapMotionManager(true);
  enableLeapCursorControl(true);

  configureLeapGesture("Gesture.Circle.MinProgress",
```

```
1);
   configureLeapGesture("Gesture.ScreenTap.
MinForwardVelocity", 0.1);
   configureLeapGesture("Gesture.ScreenTap.MinDistance",
0.1);
   configureLeapGesture("Gesture.KeyTap.
MinDownVelocity", 20);
   configureLeapGesture("Gesture.KeyTap.MinDistance",
1.0);
}
```

В ее начале создаются 2 контроллера: для клавиатуры и сенсора Leap Motion:

```
new ActionMap(keyInput);
new ActionMap(leapInput);
```

Оба они являются картами активностей — ActionMap. Для карты клавиатуры определяется реакция на нажатие клавиши Escape:

```
keyInput.bindCmd(keyboard, "escape", "quit();", "");
```

В результате чего вызывается функция quit.

Для карты активности Leap Motion определяются 4 события — жеста, забегая вперед, фактически используются только 2, но для удобства лучше определить остальные. Движок Torque 2D поддерживает 4 базовых жеста:

— «круг», нарисованный пальцами или подручными средствами (т.к. карандаш) в воздухе;

— «экранное нажатие» — длинное движение руки вперед;

— «нажатие клавиши» — жест, подобный нажатию клавиши на клавиатуре или кнопки — элемента управления на экране;

— «быстрое проведение — свайп», подобно нажатию элемента графического интерфейса на сенсорном экране с последующим проведением указателя, т.е. нажатие вперед и быстрое проведение указателя;

На основе этих базовых жестов можно создавать сложные движения и жесты, компонуя и совмещая в последовательности базовые. Сформируем карту активностей, определяя обработчики событий с помощью метода bindObj объекта leapInput класса ActionMap, в качестве параметров этому методу передаются: символьное имя устройства — leapdevice, название события, имя функции — обработчика, последним параметром передается ссылка на объект менеджера управления:

```
leapInput.bindObj(leapdevice, circleGesture,
reactToCircleBuilder, %this);
leapInput.bindObj(leapdevice, screenTapGesture,
reactToScreenTapBuilder, %this);
leapInput.bindObj(leapdevice, swipeGesture,
reactToSwipeBuilder, %this);
leapInput.bindObj(leapdevice, keyTapGesture,
reactToKeyTapBuilder, %this);
```

Таким образом, определяются обработчики для 4-х событий:

— circleGesture — рисование круга;

— screenTapGesture — экранное нажатие;

— swipeGesture — жест быстрого проведения пальцем и/или указателем — свайп;

— keyTapGesture — нажатие виртуальной кнопки;

После формирования карт активностей они запихиваются в стек:

```
keyInput.push();
leapInput.push();
```

Далее происходит задание значений переменным менеджера управления, а именно включение определенных реакций. То есть их значения используются в обработчиках событий для проверки, включена ли обрабатываемая реакция и, если нет, тогда обработчик не выполняется.

Torque 2D скрывает многословные инициализацию и использование ПО Leap Motion, экспортируя в скриптовый код стройные вызовы и обработчики. Чтобы инициализировать устройство вместе с ПО, включить менеджер и запустить использование данных, получаемых с сенсора для управления курсором достаточно 3 строчки:

```
initLeapMotionManager();
enableLeapMotionManager(true);
enableLeapCursorControl(true);
```

На самом деле, для организации управления посредством контроллера в играх на Torque 2D не нужен Leap Motion SDK. Движок в своем составе имеет динамическую библиотеку Lead.dll, которая осуществляет связь игры с драйвером устройства.

Под конец конструктора происходят 5 вызовов функции configureLeapGesture. Она служит для настройки переменных-членов для определенного жеста в движке. Первым параметром эта функция принимает имя переменной, которое состоит из: ключевого слова Gesture, затем через точку указывается имя жеста и через точку имя

свойства, вторым параметром функция получает значение, устанавливаемое для указанного в 1-м параметре свойства определенного жеста.

```
configureLeapGesture("Gesture.Circle.MinProgress", 1);
configureLeapGesture("Gesture.ScreenTap.
MinForwardVelocity", 0.1);
configureLeapGesture("Gesture.ScreenTap.MinDistance",
0.1);
configureLeapGesture("Gesture.KeyTap.MinDownVelocity",
20);
configureLeapGesture("Gesture.KeyTap.MinDistance",
1.0);
```

У жеста «круг» есть свойство MinProgress, которое отвечает за полноту рисования круга для срабатывания события. То есть значение 1 означает полный круг, однако, если задать 0.5, то для срабатывания события достаточно нарисовать половину круга. При полном круге мы избегаем срабатывания случайных событий. У «экранного нажатия» есть: минимальная скорость движения руки вперед, минимальная дистанция для движения. У жеста «нажатие виртуальной кнопки»: минимальная скорость движения сверху вниз и минимальная дистанция.

В деструкторе класса InputManager уничтожаются карты активностей:

```
function InputManager::destroy()
{
   keyInput.pop();
   keyInput.delete();
   leapInput.pop();
   leapInput.delete();
}
```

После создания InputManager ожидает возникновения событий. Когда движком определен жест «круга», управление передается в функцию reactToCircleBuilder:

```
function InputManager::reactToCircleBuilder(%this,
%id, %progress, %radius, %isClockwise, %state)
{
  if (!%this.enableCircleGesture)
    return;
  if (isLeapCursorControlled())
  {
    if (%progress > 0 && %state == 3)
    {
```

```
          LeapObjectsGame.grabObjectsInCircle();
          LeapObjectsGame.schedule(300,
"hideCircleSprite");
     }
     else if (%progress > 0 && %state != 3)
     {
          LeapObjectsGame.showCircleSprite(%radius/5,
%isClockwise);
     }
   }
}
```

Функция получает 6 параметров: %this — вызвавший объект (в данном случае, InputManager), %id — номер пальца, %progress — на сколько полно нарисован круг, %radius — радиус воображаемого круга, %isClockwise — направление рисования круга: по часовой стрелке или против, %state — состояние рисования: 1 — начало, 2 — обновление (т.е. рисование по 2-му разу), 3 — рисование окончено. Внутри функции проверяется: включен ли режим рисования, если нет, покидаем функцию, иначе с помощью системной функции isLeapCursorControlled происходит проверка: контролируется ли курсор с помощью контроллера (мы включаем этот режим при инициализации менеджера управления), если да, то выполняется следующая проверка, в обратно случае — идем на выход. Успех следующей проверки заключается в том, чтобы %progress был больше 0 и %state был равен 3 — конец рисования. В таком случае вызывается метод grabObjectsInCircle класса LeapObjectsGame:

```
function LeapObjectsGame::grabObjectsInCircle( %this )
{
   %worldPosition = %this.circleSprite.getPosition();
   %size = %this.circleSprite.getSize();
   %radius = (%size._0 * 0.5);
   echo("Radius:" SPC %radius);
   %picked = myScene.pickCircle(%worldPosition,
%radius);
   // выйти из функции, если ничего не выбрано
   if ( %picked $= "" )
     return;
   // получить количество выбранных объектов
   %pickCount = %picked.Count;
   for( %n = 0; %n < %pickCount; %n++ )
   {
     // извлечь 1 из выбранных объектов
     %pickedObject = getWord( %picked, %n );
```

```
     // пропустить объект, если он статический или
//кинематический
     if ( %pickedObject.getBodyType() $= "static" ||
%pickedObject.getBodyType() $= "kinematic")
          continue;

     if (%pickedObject.class $= "fishClass") {
          %pickedObject.setLinearVelocityX(0);
          %pickedObject.setLinearVelocityY(0);
          %this.selectedObjects.add(%pickedObject);
     }
   }
}
```

В 1-й строчке кода метода координаты спрайта круга сохраняются в переменной %worldPosition. Мы помним, что ссылка на этот спрайт сохраняется в переменной-члене circleSprite класса LeapObjectsGame сразу после создания спрайта. Таким же подходом сохраняется размер спрайта в переменной %size. Затем вычисляется радиус. Когда у нас есть позиция круга и его радиус, мы можем определить: какие объекты попали внутрь круга, для этого служит метод pickCircle глобального объекта сцены myScene класса Scene. В параметрах метод получает позицию центра круга и его радиус. Если ни один объект не попал в круг, переменная %picked имеет значение пустой строки, в таком случае покидаем функцию. В ином случае получаем количество объектов и запускаем цикл от нуля до этого количества. В теле цикла последовательно получаем каждый объект, так как массив объектов %picked — это строка, в которой id объектов разделены пробелами, мы выбираем отдельное слово с помощью функции getWord, передавая ей строку — %picked и номер счетчика цикла. Получив объект, проверяем его физический тип, если он не равен «dynamic», выходим. В противном случае проверяем класс этого динамического объекта, если это рыба — FishClass, тогда происходит остановка ее движения по обеим осям, плюс она добавляется в контейнер selectedObjects с помощью метода add последнего: %this.selectedObjects.add(%pickedObject);

После завершения метода grabObjectsInCircle управление возвращается в функцию reactToCircleBuilder, где через 300 миллисекунд на выполнение планируется метод hideCircleSprite класса LeapObjectsGame. Этот метод только скрывает спрайт с кругом:

```
function LeapObjectsGame::hideCircleSprite( %this )
{
```

```
   // скрыть спрайт
   %this.circleSprite.visible = false;
}
```

Обратите внимание: хотя круг висит на экране 0,3 секунды, объекты, попавшие в него, останавливаются только в момент его появления на экране, что случается после выполнения пальцем жеста «круг».

Далее в методе reactToCircleBuilder завершается 1-я ветка условного оператора. Вторая ветка уловного оператора выполняется при условии, когда %progress больше нуля, при этом %state не равен 3. В таком случае вызываем метод showCircleSprite класса LeapObjectsGame. Этот метод отображает спрайт с кругом на экране. Кроме этого, в нем изменяется размер спрайта, позиция и задается скорость вращения.

```
function LeapObjectsGame::showCircleSprite( %this,
%radius, %isClockwise )
{
   // если круг невидим, показать его
   if (!%this.circleSprite.visible)
   {
      %this.circleSprite.visible = true;

      // найти координаты курсора
      %worldPosition = mySceneWindow.
getWorldPoint(Canvas.getCursorPos());

      // переместить круг в эту позицию
      %this.circleSprite.position = %worldPosition;
   }

   // изменить размер спрайта с кругом
   %this.circleSprite.size = %radius;

   // повернуть круг по часовой стрелки или против
   if (%isClockwise)
      %this.circleSprite.AngularVelocity = -180;
   else
      %this.circleSprite.AngularVelocity = 180;
}
```

Кроме указателя на объект — синглтон LeapObjectsGame, метод получает: %radius — размер спрайта, %isClockwise — флаг указывающий направление вращения: по часовой стрелки или против.

В начале метода проверяется: виден ли круг, если нет, то делаем его видимым, получаем координаты виртуального курсора — указателя

Leap Motion и устанавливаем эти координаты в качестве позиции для спрайта. После конца условного оператора спрайту присваивается новый размер. В завершение метода проверяется флаг %isClockwise: если он равен 1, тогда спрайт вращается по часовой стрелки — свойство AngularVelocity принимает значение -180, в обратном случае вращается против часовой стрелки, принимая значение 180. Обратите внимание: положительное ускорение вращения поворачивает объект против часовой стрелки.

На этом метод reactToCircleBuilder подходит к концу.

Следующие 3 обратных вызова: reactToSwipeBuilder (вызывается, когда осуществлен жест «свайп»), reactToScreenTapBuilder (жест «экранное нажатие»), reactToKeyTapBuilder (жест «нажатие кнопки») делают одно и то же: они только проверяются возможность своего выполнения, в случае чего очищают контейнер selectedObjects, вызывают соответствующий метод:

```
LeapObjectsGame.deleteSelectedObjects();
```

Метод deleteSelectedObjects имеет следующий вид:

```
function LeapObjectsGame::deleteSelectedObjects( %this )
{
   echo("count = " @ %this.selectedObjects.
getCount());
  while (%this.selectedObjects.getCount() > 0) {
     %obj = %this.selectedObjects.getObject(0);
//получение ссылки на объект
     %this.selectedObjects.remove(%obj);//изъятие
//объекта из контейнера
     %obj.delete();//фактическое удаление объекта
  }
}
```

То есть он пробегает по всем элементам контейнера, последовательно удаляя каждый из них в цикле while.

Это была последняя не рассмотренная функция нашей игры, теперь все готово.

5 Заключение

В течение главы мы разработали небольшую игру, управление игровым процессом в которой происходит посредством контроллера Leap

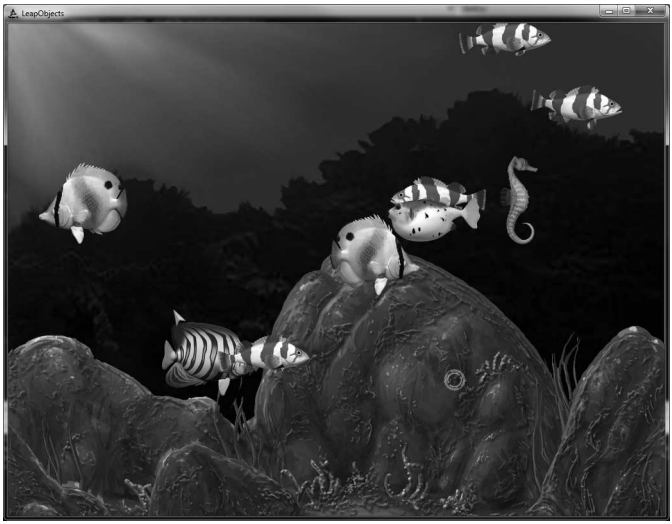

Рис. 12.11. Скриншот игры LeapObjects

Motion. Он прекрасно подходит для управления игровым процессом, надо только проапгрейдить старые жанры, чтобы они отвечали современным требованиям и удачно сочетались с новыми устройствами управления.

Игровой движок Torque 2D максимально упростил процесс создания не только визуальной и геймплейной частей игры, но и существенно облегчил ее адаптацию к управлению контроллером Leap Motion. Говоря иначе, T2D взял на себя все подготовительные мероприятия для управления игрой с помощью сенсора, предоставив нам удобный механизм для реализации управления игрой.

Leap Motion революционный контроллер, он не только заменил сенсорный экран, он предоставил управление пространством, сделав еще более прозрачной границу между реальным миром и виртуальной реальностью. На уровне разработчика ПО предоставляется удобный программный интерфейс, позволяющий управлять всеми возможностями сенсора. Кроссплатформенные инструменты разработчика дают последнему доступ к устройству на множестве языков программирования. Кроме того, api имеет стройную и понятную структуру: в каждый момент времени контроллер снимает изображение, формирует на основе его кадр и посылает на верхний уровень — в прикладную про-

грамму, где программист, распарсив кадр, работает с такими сущностями, как: руки, пальцы, указатели (инструменты) и другое.

Из-за наличия в устройстве двух камер, оно часто монтируется на шлема виртуальной реальности для создания эффекта дополненной реальности, что осуществляется благодаря наличию в изображениях, снимаемых камерами, измеренных значений яркости инфракрасного излучателя, а так же калибровочных данных необходимых для коррекции сложного объектива.

Итоги

Подошло к концу наше занимательное путешествие в виртуальный мир волшебства и чудес. Мир, создателями которого мы стали. Потому что любая игра несет в себе замыслы создателя, каждая игра становится достоянием виртуального мира. С каждой новой игрой происходит расширение виртуального пространства, от чего реальный мир становится чуточку интереснее.

В результате прочтения книги и выполнения представленных в ней заданий — разработки предлагаемых игр, вы не только существенно подняли навыки разработки игр, но вместе с этим изучили мощный кроссплатформенный и удобный для использования игровой движок Torque 2D, свободно распространяемый ныне с открытыми исходными кодами. В рамках этого движка вы так же научились программировать на его скриптовом языке Torque Script, изучили понятие ассетов вместе с языком TAML, научились их подготавливать для файлов разных форматов, в том числе: изображений, анимаций и частиц. Кроме того, Torque Script имеет удобную связь с C++ кодом движка, и вы можете легко добавлять новую функциональность в код движка для дальнейшего использования в скриптах.

Мы немного затронули тему хранилища проектов GitHub. Разобрались в минимуме необходимом для создания игры — заготовки проектов. Нами были подробно изучены наиболее часто применяемые в играх объекты и физические законы, которым они следуют. Тема физики стоит краеугольным камнем в разработке игр, и мы подробно изучили ее с точки зрения Torque 2D.

В течение книги мы разработали 4 игры. Первой из них был ремейк легендарной классической игры Asteroids. При ее разработке мы применили теоретические знания, полученные в предыдущих главах об объектах, включенных в движок, их физических свойствах. Рассмотрели способы распространения игр, разработанных на движке Torque 2D. Затем речь пошла о поддерживаемых платформах, в том числе десктопных и мобильных. Далее мы подготовили билд для устройства под управлением операционной системы Android.

Второй игрой стал простой физический эксперимент, где мы применили возможности движка для построения физической симуляции. В ней на диске размещен крепеж, к которому прикреплена цепь, на конце которой привязан камень. Вращая диск, соответственно вращается ка-

мень, поддаваясь всем физически законам. Во время разработки этой демонстрационной игры мы еще глубже окунулись в физические механизмы Torque 2D.

Третья игра стала центральной темой книги. Она посвящена разработке логической игры MagicMancala. Разрабатывая эту игру, мы на практике рассмотрели массу вопросов и их решений по созданию игры с помощью движка Torque 2D. Вдобавок мы добавили в игру возможность использования магии и воспроизведение соответствующих эффектов. Затем, подготовив билд и добавив в него рекламу, мы выложили его в Google Play для всеобщего доступа.

В последней главе мы разработали еще одну небольшую игру для реализации управления с помощью контроллера Leap Motion.

Эпилог. Планы на будущее

Движок Torque 2D активно развивается сообществом разработчиков на github.com. Вы, уважаемый читатель, тоже можете помочь этому движению в развитие движка.

Я начал писать эту книгу, когда актуальной была версия Torque 2D 2.0. Уже через некоторое время появилась версия 3.0, за ней 3.1 и 3.2. А незадолго до сдачи рукописи в печать появилась версия 3.3. И каждый раз я переделывал примеры и переписывал необходимые главы. Все, чтобы предоставить читателю самый свежий и актуальный материал. На своей веб-страничке я буду выкладывать обновленные материалы и коды примеров.

Я очень люблю семейство движков Torque. После написание этой книги у меня появилась идея написать книгу о движке Torque 3D — трехмерной версии Торка. Я с ней так же очень много работаю и часто использую для разработки своих трехмерных игровых проектов. На этом движке разработано много замечательных игр, например, из последних стоит отметить MMORPG «Life is Feudal», разработанная российской инди-командой Bitbox Ltd. Эта игра считается одной из лучших MMORPG в мире. Из этого следует, что движок Torque 3D очень интересная, популярная и мощная технология.

На этом я хочу поблагодарить вас, дорогой друг, за прочтение книги и пожелать успехов на ниве игростроя!

Связь с автором

Автор очень положительно относится к обратной связи и любит получать письма от читателей, разговаривать об играх и их разработке. Самый верный способ связаться с автором — это написать ему письмо на электронный адрес: yazevsoft@gmail.com.

Кроме того, обязательно посетите его страничку в интернете, посвященную разработке игр на движке Torque 2D, куда будут помещаться обновления и исправления для книги и примеров: http://www.t2d-dev.ru . Также там будет доступен новый материал по разработке игр.

Дополнительный сопроводительный материал

Рассмотрим содержимое дополнительного сопроводительного материала к данной книге.

В папке images находятся цветные изображения к каждой главе.

В папке samples находятся все рассматриваемые в книге приложения и их исходные коды:

— В подпапке «MyProject» находится заготовка для игр на движке Torque 2D, подготовленная нами в главе 3 — «Эксперименты».

— В подпапке «Torque2D-android-studio-Asteroids» находятся все материалы, необходимые для запуска игры Asteroids на платформе PC в ОС Windows, на Mac в OS X и на устройствах под управлением ОС Android. Разработка игры Asteroids описывается в главах: 6 — «Разработка аркадной игры Asteroids» и 7 — «Torque 2D и мобильные платформы».

— В подпапке «ProjectManager» находится менеджер проектов (вместе с исходным кодом), предназначенный для автоматического создания заготовок проектов на движке Torque 2D. Мы занимались его разработкой на протяжении главы 8 — «Менеджер проектов для Torque 2D».

— В подпапке RopeStone находится одноименная игра — физический эксперимент, разработанный нами в главе 9 — «Простой физический эксперимент». Это приложение предназначено для запуска в ОС Windows.

— В подпапке «Torque2D-dev-MM-book» находятся исходные коды, необходимые изображения, анимации и другие материалы, необходимые для построения и запуска логической игры MagicMancala на компьютере с ОС Windows и на устройствах под управлением ОС Android. Разработке этой игры посвящены главы: 10 — «Логическая игра MagicMancala» и 11 — «MagicMancala в Google Play».

— В подпапке LeapObjects находится игра (вместе с исходными кодами) для PC, где управление реализовано посредством контроллера Leap Motion. Мы разработали эту игру в главе 12 — «Использование контроллера Leap Motion».

— Кроме того, для удобства в папке samples находится исполняемый файл менеджера проектов — ProjectManager.exe.

Версию движка Torque 2D, с которой работал автор, подготавливая материал для этой книги, можно скачать с его страницы на GitHub: https://github.com/yurambo/Torque2D/tree/development.

Дополнительный материал можно скачать с сайта автора: *http://www.t2d-dev.ru*.

Юрий Язев

Как самому написать мобильную 2D-игру

Ответственный за выпуск: **В. Митин**
Верстка и обложка: **СОЛОН-Пресс**

По вопросам приобретения обращаться:
ООО «СОЛОН-Пресс»
123001, г. Москва, а/я 82
Телефоны: (495) 617-39-64, (495) 617-39-65
E-mail: **kniga@solon-press.ru**, **www.solon-press.ru**

По вопросам подписки на журнал «Ремонт & Сервис» обращаться:
ООО «СОЛОН-Пресс»
тел.: (495) 617-39-64, www.remserv.ru

ООО «СОЛОН-Пресс»
115487, г. Москва,
пр-кт Андропова, дом 38, помещение № 8, комната № 2.
Формат 60×88/16. Объем 29,75 п. л. Тираж 100 экз.

Заказ №