Дэвид Вольф

OpenGL 4. Язык шейдеров. Книга рецептов

David Wolff

OpenGL 4 Shading Language Cookbook

[PACKT] PUBLISHING
open source
community experience distilled

Дэвид Вольф

OpenGL 4. Язык шейдеров. Книга рецептов

ДМК Пресс
издательство
Москва, 2015

УДК 004.92;004.42GLSL
ББК 32.973
В72

Вольф Д.
В72 OpenGL 4. Язык шейдеров. Книга рецептов / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2015. – 368 с.: ил.

ISBN 978-5-97060-255-3

Язык шейдеров OpenGL (OpenGL Shading Language, GLSL) является фундаментальной основой программирования с использованием OpenGL. Его применение дает беспрецедентную гибкость и широту возможностей, позволяет использовать мощь графического процессора (GPU) для реализации улучшенных приемов отображения и даже для произвольных вычислений. Версия GLSL 4.x несет еще более широкие возможности, благодаря введению новых видов шейдеров: шейдеров тесселяции и вычислительных шейдеров.

В этой книге рассматривается весь спектр приемов программирования на GLSL, начиная с базовых видов шейдеров – вершинных и фрагментных, – и заканчивая геометрическими, вычислительными и шейдерами тесселяции. Здесь приводится множество практических примеров – от наложения текстур, воспроизведения теней и обработки изображений до применения искажений и манипуляций системами частиц. Прочтя ее, вы сможете задействовать GPU для решения самых разных задач, даже тех, что никак не связаны с формированием изображений.

Издание предназначено для программистов трехмерной графики, желающих задействовать в своих проектах всю мощь современных программных и аппаратных средств.

УДК 004.92;004.42GLSL
ББК 32.973

ISBN 978-1-78216-702-0 (анг.)
ISBN 978-5-97060-255-3 (рус.)

Содержание

Об авторе

Дэвид Вольф (David Wolff) – адъюнкт-профессор факультета «Информатика и вычислительная техника» в университете Пасифик Лютеран (Pacific Lutheran University, PLU). Имеет степень кандидата физико-математических наук и магистра информатики, полученную в университете штата Орегон. Преподает компьютерную графику студентам университета Пасифик Лютеран уже более 10 лет.

О рецензентах

Варфоломей Филипек (Bartłomiej Filipek) – опытный разработчик программного обеспечения, всеми силами старающийся передать свой опыт другим. Вот уже пять лет ведет курс программирования компьютерной графики с применением OpenGL в Ягеллонском университете (Краков, Польша). Кроме того, читает лекции и проводит семинары на тему современных приемов программирования на языке C++ в местных университетах.

Имеет огромный опыт разработки программного обеспечения, в том числе низкоуровневого кода для систем отображения, крупномасштабных программных продуктов, игровых движков, мультимедийных приложений, пользовательских интерфейсов, вычислений на GPU и даже приложений для клинического мониторинга.

Он делится своим опытом программирования в своем блоге по адресу: **http://www.bfilipek.com/**.

Томас Ле Гуэро-Древиллон (Thomas Le Guerroué-Drévillon) – инженер-программист, недавно закончивший обучение. Увлекается математикой и рисованием – ему легко удается совмещать эти два увлечения в компьютерной графике.

Родился во Франции, но приложил все усилия, чтобы воспользоваться возможностями, данными ему его университетом, чтобы учиться и работать по всему миру. Первый международный опыт получил в Эстонии, затем обучался в престижном Корейском институте перспективных научных исследований и технологий (KAIST) в Южной Корее и, наконец, сейчас живет в Канаде.

Даже при том, что математическую базу он получил в университете, опыт владения OpenGL и GLSL приобретался им самостоятельно. Томас уверен, что между документированным API и реализацией шейдеров лежит глубокая пропасть, и без колебаний принялся за рецензирование этого сборника рецептов, который он с радостью приобрел бы, когда только начинал осваивать OpenGL/GLSL.

Доктор Мухаммад Мобин Мования (Dr. Muhammad Mobeen Movania) защитил докторскую диссертацию по теме «Перспективы компьютерной графики и визуализации» в Наньянском технологическом университете, Сингапур. После защиты диссертации приступил к работе в научно-исследовательском институте A-Star (Сингапур) по теме «Дополненная реальность на основе систем виртуальной примерки и имитации движения тканей с использованием GPU и OpenGL». Прежде чем поступить в Наньянский университет, работал младшим разработчиком графики в Data Communication and Control (DCC) Pvt. Ltd. (Карачи, Пакистан). Использовал DirectX и OpenGL для создания интерактивных тактических тренажеров реального времени и динамических комплексных тренажеров. В числе его интересов можно назвать: методы объемной визуализации на основе GPU, приемы использования GPU, физика мягких тканей в режиме реального времени, динамическое наложение теней в режиме реального времени, определение столкновений в режиме реального времени и отклик на них, а также иерархические структуры геометрических данных. Является автором проекта OpenCloth (**http://code.google.com/p/opencloth**), в котором реализовал различные алгоритмы имитации тканей с использованием OpenGL. В его блоге (**http://mmmovania.blogspot.com**) можно

найти большое число полезных советов и хитростей, связанных с компьютерной графикой. В свободное время любит сочинять музыку и играть в сквош.

Доктор Мобин издал несколько статей о компьютерной графике и визуализации в режиме реального времени. Недавно закончил работу над книгой по OpenGL («OpenGL Development Cookbook», изданной в Packt Publishing в 2013 году), где описываются практические рецепты программирования графики с использованием современной версии OpenGL. Его перу принадлежит также глава в другой книге («OpenGL Insights», выпущенной издательством AK Peters/CRC Press в 2012 году).

В настоящее время доктор Мобин работает адъюнкт-профессором в университете DHA Suffa University (Карачи, Пакистан), где занимается взращиванием юных дарований, которые завтра станут выдающимися программистами и исследователями.

В первую очередь я хочу возблагодарить всемогущего Аллаха за то, что он так милостив ко мне и моей семье. Я также очень благодарен моей семье, родителям (мистеру и миссис Абдул Азизу Мования (Abdul Aziz Movania)), моей супруге Танвир Таджи (Tanveer Taji), моим сестрам (миссис Азра Салем (Azra Saleem) и миссис Саджида Шакир (Sajida Shakir)), моим братьям (мистеру Халид Мования (Khalid Movania) и мистеру Абдул Маджид Мования (Abdul Majid Movania)), всем моим племянникам/племянницам и моей дочери (Мунтаха Мования (Muntaha Movania)).

Дарио Скарпа (Dario Scarpa) занимается программированием последние 15 лет и не намерен оставлять эту работу. Переходя с одного места на другое, он побывал в шкуре и разработчика, и системного администратора, и создателя видеоигр, и экзаменатора по информатике в университете Салерно (Италия). При этом он осуществлял руководство разработкой проекта Zodiac для компании Adventure Productions, интернет-магазина, специализирующегося на распространении приключенческих игр. В то время когда Дарио занимался рецензированием этой книги, он параллельно готовился получить степень магистра со своей дипломной работой по теме компьютерной графики, где демонстрировал обширные возможности – как нетрудно догадаться – OpenGL.

Джавед Раббани Шах (Javed Rabbani Shah) имеет квалификацию инженера-электромеханика, полученную в инженерно-технологическом университете (Лахор, Пакистан) в 2004 году. Свою профессиональную карьеру он начинал в Delta Indus Systems (ныне Vision Master Inc.), где плотно занимался системами статистического контроля процессов, применяя такие технологии, как обработка изображений, трехмерное машинное зрение и динамически перепрограммируемые микросхемы. Затем в 2007 году перешел в подразделение Embedded Systems компании Mentor Graphics, где получил возможность поработать с системой реального времени Nucleus+, микропрограммами USB 2.0, WebKit и OpenGL ES 2.0. Он стал инициатором внедрения OpenGL ES 2.0 в кросс-платформенный движок пользовательского интерфейса Inflexion 3D компании Mentor.

В настоящее время Джавед работает в Saffron Digital, в центре Лондона, где занимается проблемами кросс-платформенной защиты видеофайлов от воспроизведения и технологиями DRM и UltraViolet.

В свободное время изучает смежные технологии, такие как OpenGL ES 3.0 и OpenCL.

www.PacktPub.com

Файлы поддержки, электронные книги, скидки и многое другое

Файлы поддержки для данной книги можно найти на сайте издательства **www.PacktPub.com**.

Знаете ли вы, что издательство Packt предлагает электронные версии всех выпускаемых им книг в форматах PDF и ePub? Приобрести электронные версии книг можно на сайте **www.PacktPub.com**, а при покупке печатной книги вы получите скидку на ее электронную копию. За дополнительной информацией обращайтесь по адресу: **service@packtpub.com**.

На сайте **www.PacktPub.com** можно также найти множество технических статей, подписаться на бесплатные новостные письма и получить исключительные скидки на печатные и электронные книги издательства Packt.

http://PacktLib.PacktPub.com

Вам нужна возможность быстро находить ответы на свои вопросы? К вашим услугам PacktLib – электронная библиотека издательства Packt. Здесь вы получите доступ ко множеству электронных книг.

Что дает подписка?

- Неограниченную возможность поиска по всем книгам издательства Packt.
- Копирование, печать и установку закладок.
- Быстрый доступ к содержимому с помощью веб-браузера.

Свободный доступ для обладателей учетных записей Packt

Если у вас есть учетная запись на сайте **www.PacktPub.com**, вы сможете использовать ее для доступа к PacktLib прямо сегодня и просмотреть девять книг бесплатно. Просто воспользуйтесь своей учетной записью для входа в библиотеку.

Предисловие

Язык шейдеров OpenGL (OpenGL Shading Language, GLSL) в настоящее время является фундаментальной основой и неотъемлемой частью программирования с использованием библиотеки OpenGL. Его применение дает беспрецедентную гибкость и широту возможностей за счет превращения прежде фиксированного конвейера обработки графики в программируемый. Благодаря GLSL можно пользоваться мощью графического процессора (Graphics Processing Unit, GPU) для реализации улучшенных приемов отображения и даже для произвольных вычислений. Версия GLSL 4.x дает программистам еще более широкие возможности управления GPU, чем когда-либо, благодаря введению новых видов шейдеров, таких как шейдеры тесселяции (tessellation shaders) и вычислительные шейдеры.

В этой книге мы рассмотрим весь спектр приемов программирования на GLSL. Начав с базовых видов шейдеров – вершинных и фрагментных, мы пройдем путь от простых до сложных приемов. Мы покажем множество практических примеров – от наложения текстур, воспроизведения теней и обработки изображений до применения искажений и манипуляций системами частиц – чтобы дать инструменты, которые вам понадобятся при использовании GLSL в ваших проектах. Мы также расскажем, как пользоваться геометрическими шейдерами, шейдерами тесселяции и совсем недавно появившимися в GLSL вычислительными шейдерами. С их помощью вы сможете задействовать GPU для решения самых разных задач, даже тех, что никак не связаны с формированием изображений. С помощью геометрических шейдеров и шейдеров тесселяции можно выполнять геометрические построения, а с помощью вычислительных – произвольные вычисления на GPU.

Для тех, кто впервые приступает к изучению GLSL, лучше читать эту книгу по порядку, начав с главы 1. Рецепты даются в порядке увеличения их сложности. Те же, кто уже имеет опыт использования GLSL, могут выбирать рецепты в произвольном порядке. В большинстве своем рецепты не зависят друг от друга, но есть и такие, которые ссылаются на другие рецепты. Введение к каждой главе содержит важную информацию о рассматриваемой теме, поэтому, возможно, вам также стоит прочитать и вводные разделы.

Версия GLSL 4.x превращает программирование с использованием OpenGL в настоящую забаву. Я искренне надеюсь, что эта книга пригодится вам и вы будете пользоваться ее рецептами при работе над своими проектами. Я верю, что программирование в OpenGL и GLSL понравится вам так же, как мне, и что описываемые здесь приемы вдохновят вас на создание красивой графики.

О чем рассказывается в этой книге

Глава 1 «Введение в GLSL» описывает этапы, которые нужно пройти, чтобы скомпилировать, скомпоновать и запустить шейдеры GLSL в программе OpenGL. В ней также рассказывается, как передавать данные в шейдеры с помощью переменных-атрибутов и uniform-переменных и как с помощью библиотеки GLM

осуществлять математические вычисления. Каждой современной программе на основе OpenGL требуется загрузчик функций. В этой главе мы расскажем, как пользоваться GLLoadGen, относительно новым и простым в использовании генератором загрузчиков OpenGL.

Глава 2 «Основы шейдеров GLSL» знакомит с основами программирования на GLSL на примере вершинных шейдеров. В этой главе вы увидите примеры основных приемов программирования шейдеров, таких как алгоритмы освещения ADS (ambient – фоновое, diffuse – рассеянное и specular – глянцевые блики), двустороннее окрашивание и плоское окрашивание. В ней также демонстрируются примеры основных концепций языка GLSL, таких как функции и подпрограммы.

Глава 3 «Освещение, затенение и оптимизация» знакомит с расширенными приемами. Основное внимание в ней будет уделено фрагментным шейдерам. Здесь вы познакомитесь с такими приемами, как освещение точечным источником света, пофрагментное отображение, придание изображению «мультяшного» вида, воспроизведение эффекта тумана, и др. Мы также обсудим несколько простых оптимизаций, которые помогут вам ускорить работу ваших шейдеров.

Глава 4 «Текстуры» представляет собой обобщенное введение в приемы использования текстур в шейдерах GLSL. Текстуры могут использоваться с самыми разными целями, помимо самого простого наложения изображения на поверхность. В этой главе мы познакомим вас с основами использования одной или более двухмерных текстур, а также с различными другими приемами наложения текстур, включая альфа-наложение, наложение по нормалям, кубическое наложение, проекционное наложение и отображение в текстуру. Мы затронем семплеры, относительно новую особенность, которая помогает отделить параметры выборки из текстур от самих текстур.

Глава 5 «Обработка изображений и приемы работы с экранным пространством» описывает типичные способы постобработки созданных изображений и другие приемы работы с экранным пространством. Постобработка изображения является важнейшим элементом современных игровых движков и других средств отображения трехмерной графики. В этой главе рассказывается, как реализовать некоторые из наиболее типичных способов постобработки, таких как создание размытости на границах ярких участков, гамма-коррекция и выделение границ. Здесь также рассматриваются некоторые приемы обработки экранного пространства, такие как отложенное освещение и затенение, сглаживание множественной выборкой и порядко-независимой прозрачности.

Глава 6 «Использование геометрических шейдеров и шейдеров тесселяции» охватывает приемы использования новых видов шейдеров. В этой главе вы познакомитесь с основными функциональными возможностями этих шейдеров и получите представление о том, как ими пользоваться. Здесь рассказывается о таких приемах, как генерация точечных спрайтов геометрическим шейдером, выделение силуэтов, тесселяция с учетом глубины, поверхности Безье и многих других.

Глава 7 «Тени» знакомит с основными приемами отображения теней в реальном масштабе времени. Эта глава включает рецепты, реализующие два основных при-

ема: карты теней и теневые объемы. Мы охватим типичные приемы сглаживания карт теней, а также покажем, как использовать геометрический шейдер для воспроизведения теневых объемов.

Глава 8 «Использование шума в шейдерах» охватывает использование градиентного шума Перлина для создания разнообразных эффектов. Первый рецепт показывает, как создать широкое разнообразие текстур шума с применением GLM (мощной математической библиотеки). Затем мы перейдем к рецептам, использующим подобные текстуры для создания таких эффектов, как текстура древесины, облака, рассыпание, брызги краски и статические разряды.

Глава 9 «Системы частиц и анимация» описывает приемы создания систем частиц. Здесь мы увидим, как с помощью системы частиц воспроизвести имитацию взрыва, облака дыма и брызг воды. В этой главе мы также будем использовать одну особенность OpenGL, которая называется трансформацией с обратной связью, чтобы увеличить производительность алгоритма за счет передачи обновления положения частиц графическому процессору GPU.

Глава 10 «Вычислительные шейдеры» знакомит с несколькими приемами использования одной из новейших особенностей в OpenGL – вычислительного шейдера. Поддержка вычислительных шейдеров открывает широкие возможности для выполнения универсальных вычислений на GPU с поддержкой OpenGL. В этой главе мы обсудим, как использовать вычислительные шейдеры для моделирования частиц, облаков, выделения границ и генерации фрактальной текстуры. К концу этой главы читатель получит хорошее представление о том, как организовать произвольные математические вычисления с помощью вычислительных шейдеров.

Что потребуется для работы с книгой

Рецепты в этой книге демонстрируют использование некоторых новейших особенностей OpenGL 4.x. Поэтому для их опробования вам потребуются графическая аппаратная подсистема (графическая карта или встроенный GPU) и драйверы, поддерживающие, как минимум, версию OpenGL 4.3. Если вы не знаете, какая версия OpenGL поддерживается вашим окружением, воспользуйтесь какой-нибудь из доступных утилит, которая поможет вам получить необходимую информацию. К таким утилитам, например, относится GLview от Realtech VR, доступная по адресу: **http://www.realtech-vr.com/glview/**. Для ОС Windows или Linux всегда можно найти самые свежие драйверы. К сожалению, драйверы для MacOS X часто выходят с запаздыванием, и если вы пользуетесь этой операционной системой, вам, возможно, придется подождать. На момент написания этих строк последняя версия MacOS X (10.9 Mavericks) поддерживала только OpenGL 4.1.

Помимо современных драйверов OpenGL, вам также потребуются:

- компилятор C++. В ОС Linux часто уже имеется установленный пакет GNU Compiler Collection (gcc, g++ и т. д.), но если это не так, его нетрудно установить с помощью менеджера пакетов. В Windows с успехом можно исполь-

зовать Microsoft Visual Studio, но если у вас нет своей копии этой системы разработки, отличным вариантом для вас может стать компилятор MinGW (доступен по адресу: **http://mingw.org/**);

- библиотека GLFW версии 3.0 или выше, доступная по адресу: **http://www.glfw.org/**. Эта библиотека обеспечивает возможность создания контекстов OpenGL, поддержку окон и событий ввода от пользователя;
- библиотека GLM версии 0.9.4 или выше, доступная по адресу: **http://glm.g-truc.net/**. Эта библиотека обеспечивает поддержку математических вычислений и содержит реализации классов матриц, векторов, часто используемых преобразований, функций шума и многое другое.

Кому адресована эта книга

В центре внимания этой книги находится язык программирования шейдеров OpenGL (OpenGL Shading Language, GLSL). Соответственно, мы не будем тратить времени на обсуждение основ программирования с использованием OpenGL. Далее в этой книге я буду предполагать, что читатель уже имеет некоторый опыт программирования в OpenGL и понимает основные концепции трехмерной графики, такие как координаты модели, координаты представления, координаты отсечения, перспективные преобразования и другие виды преобразований. Однако здесь не делается никаких предположений о навыках программирования шейдеров, поэтому, если вы только начинаете осваивать GLSL, данная книга может послужить для вас неплохой отправной точкой.

Если вы имеете опыт использования OpenGL и стремитесь побольше узнать о программировании на GLSL, тогда эта книга для вас. Даже если вы имеете некоторый опыт программирования шейдеров, вы наверняка найдете здесь ценные для вас рецепты. Мы охватим широкий диапазон приемов, от простых до сложных, использующих новейшие особенности OpenGL (такие как вычислительные шейдеры и шейдеры тесселяции). И даже программисты с большим опытом использования GLSL, стремящиеся побольше узнать об этих новейших особенностях, также найдут эту книгу полезной.

Проще говоря, эта книга адресована программистам, знакомым с основами трехмерной графики в OpenGL и заинтересованным в изучении языка GLSL или желающим получить дополнительные сведения о некоторых новейших особенностях GLSL 4.x.

Соглашения

В этой книге используется несколько разных стилей оформления текста с целью обеспечить визуальное отличие информации разных типов. Ниже приводятся несколько примеров таких стилей оформления и краткое описание их назначения.

Программный код в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, пути в файловой системе, ввод пользователя и ссылки в Twitter оформляются, как показано в следующем предложении: «Число рабочих групп определяется параметром `glDispatchCompute`.».

Блоки программного кода оформляются так:

```
void main()
{
    Color = VertexColor;
    gl_Position = RotationMatrix * vec4(VertexPosition,1.0);
}
```

Когда нам потребуется привлечь ваше внимание к определенному фрагменту в блоке программного кода, мы будем выделять его жирным шрифтом:

```
void main()
{
    Color = VertexColor;
    gl_Position = RotationMatrix * vec4(VertexPosition,1.0);
}
```

Ввод и вывод в командной строке будут оформляться так:

```
Active attributes:
1    VertexColor (vec3)
0    VertexPosition (vec3)
```

Новые термины и **важные определения** будут выделяться в обычном тексте жирным.


Так будут оформляться предупреждения и важные примечания.


Так будут оформляться советы и рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезными.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** или **www.дмк.рф** в разделе Читателям – Файлы к книгам.

Также все исходные тексты для всех рецептов в этой книге доступны в GitHub по адресу: **https://github.com/daw42/glslcookbook**.

Цветные иллюстрации для этой книги

Мы бесплатно предоставляем файл PDF с цветными иллюстрациями – скриншотами/диаграммами, что приводятся в этой книге. Цветные иллюстрации помогут вам лучше понять, что происходит на экране. Скачать их можно на сайте **www.dmkpress.com** или **www.дмк.рф** в разделе Читателям – Файлы к книгам.

Ошибки и опечатки

Мы тщательно проверяем содержимое наших книг, но от ошибок никто не застрахован. Если вы найдете ошибку в любой из наших книг – в тексте или в программном коде, мы будем весьма признательны, если вы сообщите нам о ней. Тем самым вы оградите других читателей от разочарований и поможете улучшить последующие версии этой книги.

Если вы найдёте какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и «Packt» очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Вопросы

Если у вас появились какие-либо вопросы, связанные с нашими книгами, присылайте их по адресу **dmkpress@gmail.com**, а мы приложим все усилия, чтобы ответить на них.

Глава 1

Введение в GLSL

В этой главе описываются следующие рецепты:

- использование загрузчика функций для доступа к новейшим возможностям OpenGL;
- использование GLM для математических вычислений;
- определение версий GLSL и OpenGL;
- компиляция шейдера;
- компоновка шейдерной программы;
- передача данных в шейдер с использованием переменных-атрибутов и буферов;
- получение списка активных атрибутов и их индексов;
- передача данных в шейдер с использованием uniform-переменных;
- получение списка активных uniform-переменных;
- использование uniform-блоков и uniform-буферов;
- получение отладочных сообщений;
- создание класса C++, представляющего шейдерную программу.

Введение

Язык программирования шейдеров OpenGL (OpenGL Shading Language, GLSL) версии 4 дает беспрецедентные возможности и гибкость программистам, заинтересованным в создании современных, интерактивных графических программ. Он позволяет легко и просто использовать мощь современных **графических процессоров (Graphics Processing Unit, GPU)**, предоставляя простые, но мощные языковые конструкции и прикладной программный интерфейс (API). Разумеется, первым шагом к использованию GLSL является создание программы на основе последней версии OpenGL API. Программы на языке GLSL не являются полностью самостоятельными; они должны входить в состав более крупных программ на основе библиотеки OpenGL. В этой главе мы познакомимся с несколькими советами и приемами, которые помогут создать и запустить простую программу. Но перед этим рассмотрим некоторые теоретические выкладки.

Язык шейдеров OpenGL

В настоящее время язык GLSL является фундаментальной и неотъемлемой частью OpenGL API. Забегая вперед, отметим, что любая программа, написанная с привлечением OpenGL API, внутри использует одну или более программ на язы-

ке GLSL. Такие «мини-программы» часто называют **шейдерными программами (shader programs)**. Обычно шейдерная программа состоит из нескольких компонентов, называемых **шейдерами (shaders)**. Каждый шейдер выполняется в рамках отдельного этапа в общем конвейере OpenGL. Каждый шейдер выполняется на GPU и, как можно заключить из названия[1], реализует (обычно) алгоритм, так или иначе связанный с эффектами освещения и затенения в изображении. Однако шейдеры способны на большее, чем просто воспроизводить эффекты освещения и затенения. С их помощью можно также воспроизводить анимацию, выполнять тесселяцию (tessellation) и даже производить универсальные математические вычисления.

Вопросам использования GPU (часто с использованием специализированных API, таких как CUDA или OpenCL) для выполнения универсальных вычислений, например в физике жидкостей и газов, молекулярной динамике, криптографии и т. д., посвящена целая область исследований с названием **GPGPU (General Purpose Computing on Graphics Processing Units – универсальные вычисления на графических процессорах)**. Благодаря появлению вычислительных шейдеров в версии OpenGL 4.3 мы теперь можем выполнять аналогичные вычисления в рамках OpenGL.

Шейдерные программы предназначены для непосредственного выполнения на GPU и действуют параллельно. Например, фрагментный шейдер может вызываться для каждого пикселя, при этом все пиксели могут обрабатываться одновременно, в отдельных потоках выполнения на GPU. Число процессоров в графической карте определяет, сколько шейдеров может выполняться одновременно. Это обстоятельство делает шейдерные программы чрезвычайно эффективными, а программист получает в свое распоряжение простой API для реализации вычислений с высокой степенью параллелизма.

Вычислительная мощность современных графических процессоров потрясает. В табл. 1.1 приводится число шейдерных процессоров, имеющееся в некоторых моделях графических карт серии GeForce, выпускаемых компанией NVIDIA (источник: **http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units**).

Таблица 1.1. Сравнение графических карт NVIDIA

Модель	Число шейдерных процессоров
GeForce GTS 450	192
GeForce GTX 480	480
GeForce GTX 780	2304

Шейдерные программы предназначены для замены части архитектуры OpenGL, которую называют **конвейером с фиксированной функциональностью (fixed-function pipeline)**. В OpenGL до версии 2.0 алгоритмы отображения были

[1] Дословно название *shader* переводится на русский язык как «программа построения теней». – *Прим. перев.*

«жестко зашиты» в функциональный конвейер и имели весьма ограниченные возможности настройки. Этот алгоритм отображения, действующий по умолчанию, составлял основу конвейера с фиксированной функциональностью. Когда нам как программистам требовалось реализовать более сложные или более реалистичные эффекты, мы использовали различные трюки, чтобы хоть как-то повысить гибкость конвейера с фиксированной функциональностью. Появление поддержки языка GLSL дало возможность заменять «жестко зашитую» функциональность своим программным кодом, написанным на GLSL, и помогло нам получить дополнительную гибкость и возможности. Более подробно о конвейере с программируемой функциональностью рассказывается в главе 2 «Основы шейдеров GLSL».

Фактически последние (основные) версии OpenGL не только дают такую возможность, но даже требуют, чтобы шейдерные программы входили в состав любых программ, использующих OpenGL. Старый конвейер с фиксированной функциональностью был объявлен устаревшим, и предпочтение отдано новому конвейеру с программируемой функциональностью, ключевым элементом которого являются шейдеры, написанные на GLSL.

Профили – базовый и совместимости

В версии OpenGL 3.0 появилась **модель определения устаревшей функциональности (deprecation model)**, обеспечивающая возможность постепенного устранения функций из спецификации OpenGL. Как предполагается, функции или особенности, объявленные устаревшими, будут удаляться в будущих версиях OpenGL. Например, непосредственный режим отображения с использованием `glBegin/glEnd` был объявлен устаревшим в версии 3.0 и убран в версии 3.1.

В OpenGL 3.2 для поддержки обратной совместимости была добавлена концепция **профилей совместимости (compatibility profiles)**. Программист, пишущий код для какой-то определенной версии OpenGL (из которой были удалены устаревшие функции), может использовать так называемый **базовый профиль (core profile)**. Любой, кто пожелает поддерживать совместимость с устаревшей функциональностью, может использовать **профиль совместимости (compatibility profile)**.

Кому-то может показаться странным, что существует также понятие контекста **опережающей совместимости (forward compatible)**, или совместимости снизу вверх, которое несколько отличается от понятий базового профиля и профиля совместимости. Контекст опережающей совместимости, как считается, просто указывает, что вся устаревшая функциональность была удалена. Иными словами, если контекст объявлен совместимым снизу вверх, это означает, что он включает только функции базового профиля, за исключением функций, объявленных устаревшими. Некоторые оконные API предоставляют возможность выбрать статус опережающей совместимости наряду с профилем.

Шаги, которые требуется выполнить для выбора базового профиля или профиля совместимости, зависят от API оконной системы. Например, в GLFW выбрать контекст опережающей совместимости и базовый профиль 4.3 можно с помощью следующего кода:

```
glfwWindowHint( GLFW_CONTEXT_VERSION_MAJOR, 4 );
glfwWindowHint( GLFW_CONTEXT_VERSION_MINOR, 3 );
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

GLFWwindow *window = glfwCreateWindow(640, 480, "Title",
      NULL, NULL);
```

Все программы, что приводятся в данной книге, проектировались с учетом опережающей совместимости с базовым профилем OpenGL 4.3.

Использование загрузчика функций для доступа к новейшей функциональности OpenGL

В ОС Windows OpenGL **ABI (application binary interface – двоичный прикладной интерфейс)** был заморожен в версии OpenGL 1.1. К сожалению, для Windows-разработчиков это означает, что они не могут использовать функции новейших версий OpenGL непосредственно. Вместо этого они должны получать доступ к таким функциям, приобретая указатели на них во время выполнения. Получить указатели на функции несложно, но для этого требуется выполнить дополнительную рутинную работу и написать дополнительный код, захламляющий программу. Кроме того, обычно в состав Windows включается стандартный заголовочный файл OpenGL `gl.h`, который также соответствует версии OpenGL 1.1. На вики-странице OpenGL отмечается, что компания Microsoft не планирует обновлять файлы `gl.h` и `opengl32.lib`, поставляемые вместе с их компиляторами. К счастью, другие позаботились о том, чтобы предоставить обновленные заголовочные файлы, а также библиотеки, которые автоматически выполняют рутинную работу и обеспечивают прозрачный доступ к необходимым указателям на функции. Существует несколько библиотек, обеспечивающих такого рода поддержку. Одной из старейших и наиболее широко используемых является библиотека **GLEW (OpenGL Extension Wrangler)**. Однако она имеет несколько серьезных недостатков, снижающих ее ценность и сделавших ее недостаточной для моих целей, когда я писал эту книгу. Во-первых, на момент написания этих слов она некорректно поддерживала базовые профили, тогда как в этой книге я хотел сосредоточиться только на новейшей функциональности, без устаревших функций. Во-вторых, в ее состав входит огромный заголовочный файл, включающий все, что когда-либо входило в состав OpenGL. Было бы предпочтительнее иметь более короткий заголовочный файл, включающий только те функции, которые можно использовать. Наконец, библиотека GLEW распространяется в виде исходных текстов, которые нужно компилировать отдельно и компоновать с проектом. Часто бывает предпочтительнее иметь загрузчик, который можно включить в проект простым добавлением файлов с исходным кодом и компилировать его непосредственно в выполняемый файл, избежав необходимости поддерживать дополнительную зависимость.

В данном рецепте будет использоваться утилита **OpenGL Loader Generator** (GLLoadGen), доступная по адресу: **https://bitbucket.org/alfonse/glloadgen/**

wiki/Home. Это очень гибкий и эффективный инструмент, решающий все три проблемы, описанные выше. Он поддерживает базовые профили, может генерировать заголовочные файлы, включающие только то, что необходимо, и генерирует лишь пару файлов (файл с исходным кодом и заголовочный файл), которые можно включить непосредственно в проект.

Подготовка

Чтобы воспользоваться утилитой GLLoadGen, необходима поддержка **Lua**. Lua – это легковесный встраиваемый язык сценариев, доступный практически для любых платформ. Двоичные файлы можно найти по адресу: **http://luabinaries.sourceforge.net**, а полностью готовый к установке комплект для Windows (LuaForWindows) можно загрузить по адресу: **https://code.google.com/p/luaforwindows**.

Загрузите дистрибутив GLLoadGen по адресу: **https://bitbucket.org/alfonse/glloadgen/downloads**. Файл дистрибутива сжат архиватором 7zip. Этот архиватор установлен не везде, поэтому вам, возможно, потребуется установить его, загрузив утилиту 7zip по адресу: **http://7-zip.org/**. Распакуйте дистрибутив в каталог по выбору. Так как утилита GLLoadGen написана на Lua, ее не нужно компилировать – сразу после распаковки дистрибутива она готова к использованию.

Как это делается...

Первый шаг – сгенерировать заголовочный файл и файл с исходным кодом для выбранной версии OpenGL и профиля. В данном примере мы сгенерируем файлы для базового профиля OpenGL 4.3. Сгенерированные файлы можно скопировать в проект и скомпилировать их вместе с исходным кодом проекта:

1. Чтобы сгенерировать файлы, перейдите в каталог, куда был распакован дистрибутив GLLoadGen, и запустите команду `GLLoadGen` со следующими аргументами:

```
lua LoadGen.lua -style=pointer_c -spec=gl -version=4.3 \
-profile=core core_4_3
```

2. После выполнения предыдущего шага должны появиться два файла: `gl_core_4_3.c` и `gl_core_4_3.h`. Скопируйте эти файлы в свой проект и включите `gl_core_4_3.c` в сборку. В своем программном коде, где требуется получить доступ к функциям OpenGL, подключите `gl_core_4_3.h`. Однако этого недостаточно – чтобы инициализировать указатели на функции, требуется также вызвать функцию `ogl_LoadFunctions`. Где-нибудь, сразу после создания контекста GL (обычно в функции инициализации) и перед вызовами любых функций OpenGL, вставьте следующий код:

```
int loaded = ogl_LoadFunctions();
if(loaded == ogl_LOAD_FAILED) {
  // Освободить контекст и прервать выполнение
  return;
}
```

```
int num_failed = loaded - ogl_LOAD_SUCCEEDED;
printf("Number of functions that failed to load: %i.\n",
       num_failed);
```

Вот и все!

Как это работает...

Команда `lua`, что вызывается на шаге 1, сгенерирует пару файлов – заголовочный файл и файл с исходным кодом. Заголовочный файл содержит определения прототипов всех выбранных функций OpenGL и переопределяет их как указатели на функции, а также определяет все константы OpenGL. Файл с исходным кодом содержит реализацию процедуры инициализации указателей на функции и несколько вспомогательных функций. Мы можем подключить `gl_core_4_3.h` к любому файлу, где необходимы объявления прототипов функций OpenGL, благодаря чему все точки входа в функции будут доступны на этапе компиляции. На этапе выполнения `ogl_LoadFunctions()` инициализирует все имеющиеся указатели на функции. Если какие-то функции не удастся загрузить, число таких функций можно определить вычитанием, как показано в шаге 2. Если требуемая функция отсутствует в выбранной версии OpenGL, код не скомпилируется, потому что в заголовочном файле будут присутствовать определения прототипов функций только для выбранной версии OpenGL и профиля.

Полное описание всех аргументов командной строки для GLLoadGen можно найти по адресу: **https://bitbucket.org/alfonse/glloadgen/wiki/Command_Line_Options**. Предыдущий пример показывает наиболее типичные настройки, но существует еще масса других, обеспечивающих этому инструменту высокую гибкость.

Теперь, получив пару файлов, мы больше не зависим от утилиты GLLoadGen, и наша программа может быть скомпилирована без ее участия. Это большое преимущество перед другими инструментами, такими как GLEW.

И еще...

Утилита GLLoadGen имеет еще несколько интересных и весьма полезных особенностей. С ее помощью можно генерировать код, более дружественный для C++, управлять расширениями и генерировать файлы, не требующие вызова функции инициализации.

Создание загрузчика на C++

Утилита GLLoadGen способна также генерировать заголовочный файл и файл с исходным кодом на C++. Добиться этого можно, передав параметр `-style`. Например, чтобы сгенерировать файлы на языке C++, нужно передать параметр `-style=pointer_cpp`, как в следующем примере:

```
lua LoadGen.lua -style=pointer_cpp -spec=gl -version=4.3 \
-profile=core core_4_3
```

Эта команда сгенерирует файлы gl_core_4_3.cpp и gl_core_4_3.hpp. Все определения функций OpenGL в этих файлах будут помещены в пространство имен gl::, и из их имен будет убран префикс gl (или GL). Например, чтобы вызвать функцию glBufferData, вам придется использовать следующий синтаксис:

```
gl::BufferData(gl::ARRAY_BUFFER, size, data, gl::STATIC_DRAW);
```

Загрузка указателей функций также немного отличается. Возвращаемое значение на этот раз является объектом, а не простым целым числом, и функция LoadFunctions теперь находится в пространстве имен gl::sys.

```
gl::exts::LoadTest didLoad = gl::sys::LoadFunctions();

if(!didLoad) {
    // освободить ресурсы (разрушить контекст) и прервать выполнение.
    return;
}
printf("Number of functions that failed to load: %i.\n",
  didLoad.GetNumMissing());
```

Автоматическая загрузка

GLLoadGen поддерживает автоматическую инициализацию указателей на функции. Она включается передачей значения noload_c или noload_cpp в параметре -style. В этом случае отпадает необходимость вызывать функцию инициализации ogl_LoadFunctions. Указатели загружаются автоматически, при вызове первой же функции. Возможно, это удобно, но такой режим работы влечет за собой небольшие накладные расходы на инициализацию.

Использование расширений

Утилита GLLoadGen не поддерживает расширения автоматически – их необходимо передавать явно в параметрах командной строки. Например, чтобы задействовать расширения ARB_texture_view и ARB_vertex_attrib_binding, можно воспользоваться следующей командой:

```
lua LoadGen.lua -style=pointer_c -spec=gl -version=3.3 \
-profile=core core_3_3 \
-exts ARB_texture_view ARB_vertex_attrib_binding
```

В параметре -exts передается список расширений, перечисленных через пробел. GLLoadGen поддерживает также возможность загружать списки расширений из файлов (с помощью параметра -extfile), а на веб-сайте проекта можно найти несколько файлов с расширениями.

GLLoadGen может также проверять наличие расширений во время выполнения. Подробности смотрите на вики-странице GLLoadGen.

См. также

GLEW, более старый и широко используемый загрузчик расширений и менеджер расширений, доступный по адресу: **glew.sourceforge.net**.

Использование GLM для математических вычислений

В основе компьютерной графики лежит математика. В ранних версиях OpenGL предоставлялась возможность управления преобразованиями и проекциями координат с использованием стандартных матричных стеков (`GL_MODELVIEW` и `GL_PROJECTION`). Однако в последних версиях OpenGL все, что относилось к матричным стекам, было удалено из библиотеки. Поэтому теперь мы должны сами реализовать все необходимое для поддержки матриц преобразований и проекций и затем передавать их в шейдеры. Конечно, можно было бы написать собственные классы матриц и векторов, но многие предпочитают готовые, надежные решения.

Одним из таких решений является библиотека **GLM (OpenGL Mathematics)**, написанная Кристофом Риччио (Christophe Riccio). Она разработана на основе спецификации GLSL, поэтому синтаксис ее функций очень похож на поддержку математических операций в GLSL. Опытные программисты на GLSL без труда освоят библиотеку GLM. Кроме того, для этой библиотеки имеется множество расширений, включая функции, которых очень не хватает в OpenGL, такие как `glOrtho`, `glRotate` или `gluLookAt`.

Подготовка

Поскольку вся библиотека GLM определена исключительно в заголовочных файлах, она устанавливается очень просто. Загрузите дистрибутив с последней версией GLM по адресу: **http://glm.g-truc.net**. Распакуйте архив и скопируйте каталог `glm` куда-нибудь в дерево каталогов, где компилятор будет искать подключаемые заголовочные файлы.

Как это делается...

Чтобы задействовать библиотеку GLM, достаточно просто подключить основной заголовочный файл и заголовочные файлы с расширениями. В этом примере мы подключим расширение, реализующее преобразования матриц:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

После этого программе станут доступны классы GLM из пространства имен `glm`. Ниже демонстрируется, как используются некоторые из них:

```
glm::vec4 position = glm::vec4( 1.0f, 0.0f, 0.0f, 1.0f );
glm::mat4 view = glm::lookAt( glm::vec3(0.0,0.0,5.0),
       glm::vec3(0.0,0.0,0.0),
       glm::vec3(0.0,1.0,0.0) );
glm::mat4 model(1.0f); // Единичная матрица
model = glm::rotate( model, 90.0f, glm::vec3(0.0f,1.0f,0.0) );
glm::mat4 mv = view * model;
glm::vec4 transformed = mv * position;
```

Как это работает...

Библиотека GLM состоит только из заголовочных файлов. То есть вся реализация подключается к программе в виде заголовочных файлов. Ее не требуется компилировать отдельно и затем компоновать с программой – достаточно расположить файлы библиотеки где-нибудь в пути поиска заголовочных файлов, и все!

В примере выше сначала создается объект типа `vec4` (вектор с четырьмя координатами), представляющий позицию в пространстве. Затем вызовом функции `glm::lookAt` создается матрица вида, имеющая размер 4×4. Эта функция действует подобно устаревшей функции `gluLookAt`. Здесь камера устанавливается в точку с координатами (0, 0, 5) и направляется в начало координат, а направление «вверх» совпадает с направлением оси Y. Далее создается матрица модели: сначала переменная `model` инициализируется как единичная матрица (передачей единственного аргумента конструктору) и затем умножается на матрицу поворота вызовом функции `glm::rotate`. Умножение здесь выполняется неявно. Функция `glm::rotate` умножает (справа) свой первый параметр на матрицу поворота, генерируемую функцией. Во втором параметре передается угол поворота (в градусах), а в третьем – ось, вокруг которой осуществляется поворот. Так как перед этой инструкцией переменная `model` хранила единичную матрицу, в конечном итоге она превращается в матрицу поворота на 90° вокруг оси Y.

В заключение умножением переменных `model` и `view` создается матрица модели вида (`mv`), после чего полученная комбинированная матрица используется для преобразования позиции `position`. Обратите внимание, что оператор умножения перегружен, благодаря чему достигается требуемый результат.

И еще...

Не рекомендуется импортировать все пространство имен GLM, как показано ниже:

```
using namespace glm;
```

Это наверняка вызовет множество конфликтов имен. Предпочтительнее импортировать имена по одному, по мере их необходимости. Например:

```
#include <glm/glm.hpp>
using glm::vec3;
using glm::mat4;
```

Использование типов GLM для передачи данных в OpenGL

Библиотека GLM поддерживает непосредственную передачу типов GLM в OpenGL с использованием векторных функций OpenGL (имена которых оканчиваются на `v`). Например, ниже показано, как передать переменную `proj` типа `mat4`:

```
glm::mat4 proj = glm::perspective( viewAngle, aspect, nearDist,
    farDist );
glUniformMatrix4fv(location, 1, GL_FALSE, &proj[0][0]);
```

См. также

- Пакет Qt SDK включает множество классов, поддерживающих операции с векторами/матрицами, и их можно считать отличной альтернативой, если вы уже имеете опыт использования Qt.
- Веб-сайт GLM (**http://glm.g-truc.net**), где можно найти дополнительную документацию и примеры.

Определение версий GLSL и OpenGL

При необходимости поддерживать широкий спектр систем очень важно уметь определять версии OpenGL и GLSL, поддерживаемые текущим драйвером. Сделать это совсем несложно – достаточно воспользоваться всего двумя функциями: `glGetString` и `glGetIntegerv`.

Как это делается...

Следующий фрагмент выведет информацию о версии в `stdout`:

```
const GLubyte *renderer = glGetString( GL_RENDERER );
const GLubyte *vendor = glGetString( GL_VENDOR );
const GLubyte *version = glGetString( GL_VERSION );
const GLubyte *glslVersion =
      glGetString( GL_SHADING_LANGUAGE_VERSION );

GLint major, minor;
glGetIntegerv(GL_MAJOR_VERSION, &major);
glGetIntegerv(GL_MINOR_VERSION, &minor);

printf("GL Vendor            : %s\n", vendor);
printf("GL Renderer          : %s\n", renderer);
printf("GL Version (string)  : %s\n", version);
printf("GL Version (integer) : %d.%d\n", major, minor);
printf("GLSL Version         : %s\n", glslVersion);
```

Как это работает...

Обратите внимание, что существуют два разных способа получить версию OpenGL: с помощью `glGetString` и с помощью `glGetIntegerv`. Первая функция может пригодиться, когда желательно получить информацию в удобочитаемом виде, но ее неудобно использовать для анализа версии в программе, потому что для этого придется проанализировать строку. Строка, возвращаемая вызовом `glGetString(GL_VERSION)`, всегда должна начинаться со старшего и младшего номеров версии, разделенных точкой, при этом младший номер может сопровождаться дополнительным числом – номером сборки, разным для разных производителей. Кроме того, остальная часть строки может содержать дополнительную инфор-

мацию о производителе и включать сведения о выбранном профиле (см. раздел «Введение» в этой главе). Важно отметить, что функцию glGetIntegerv можно использовать только в версии OpenGL 3.0 или выше.

При вызове с аргументами GL_VENDOR и GL_RENDERER функция возвращает дополнительную информацию о драйвере OpenGL. Вызов glGetString(GL_VENDOR) вернет название компании-производителя текущей реализации OpenGL. Вызов glGetString(GL_RENDERER) вернет название аппаратной платформы (например, ATI Radeon HD 5600 Series). Имейте в виду, что оба аргумента действуют одинаково в разных версиях, поэтому их можно использовать для определения текущей платформы.

Но самое важное для нас в контексте этой книги заключается в том, что вызов glGetString(GL_SHADING_LANGUAGE_VERSION) возвращает номер поддерживаемой версии GLSL. Возвращаемая строка должна начинаться со старшего и младшего номеров, разделенных точкой, и, так же как строка, полученная при вызове с аргументом GL_VERSION, может включать дополнительную информацию о производителе.

И еще...

Часто бывает полезно узнать, какие расширения поддерживаются текущей реализацией OpenGL. В версиях ниже OpenGL 3.0 можно было получить полный список имен расширений, разделенных пробелами, как показано ниже:

```
GLubyte *extensions = glGetString(GL_EXTENSIONS);
```

Возвращаемая строка могла оказаться очень длинной, и ее анализ мог приводить к ошибкам при невнимательном отношении.

В OpenGL 3.0 стал поддерживаться новый способ, а прежняя функциональность была объявлена устаревшей (и полностью удалена в версии 3.1). Теперь имена расширений индексируются, и их можно запрашивать по индексам. Мы используем версию glGetStringi для этого. Например, получить имя расширения с индексом i можно так: glGetStringi(GL_EXTENSIONS, i). Ниже демонстрируется фрагмент, который выведет список всех расширений:

```
GLint nExtensions;
glGetIntegerv(GL_NUM_EXTENSIONS, &nExtensions);

for( int i = 0; i < nExtensions; i++ )
      printf("%s\n", glGetStringi( GL_EXTENSIONS, i ) );
```

См. также

- Инструмент GLLoadGen обладает дополнительной поддержкой получения информации о версии и расширениях. Обращайтесь к рецепту «Использование загрузчика функций для доступа к новейшей функциональности OpenGL» и веб-сайту GLLoadGen.

Компиляция шейдера

Сначала нам нужно узнать, как компилируются шейдеры GLSL. Компилятор GLSL встроен непосредственно в библиотеку OpenGL, и шейдеры могут компилироваться только в контексте выполняющейся программы. В настоящее время не существует внешних инструментов для предварительной компиляции шейдеров GLSL и/или шейдерных программ.

Совсем недавно, в версии OpenGL 4.1, была добавлена возможность сохранять скомпилированные шейдеры в файлы, что позволяет программам избежать затрат на их компиляцию за счет загрузки предварительно скомпилированного кода.

Компиляция шейдера заключается в создании объекта шейдера, сохранении в нем исходного кода (в виде строки или множества строк) и вызове метода компиляции. Эта процедура схематически изображена на рис. 1.1.

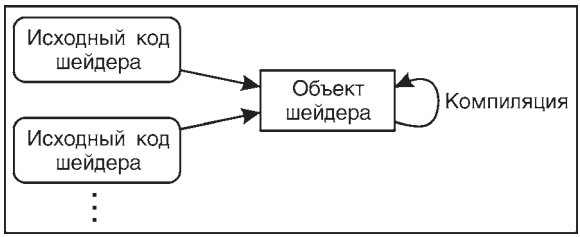

Рис. 1.1 ❖ Процесс компиляции шейдера

Подготовка

Чтобы исследовать процедуру компиляции шейдера, нам нужен простой пример исходного кода. Начнем со следующего простого вершинного шейдера. Сохраните его в файле `basic.vert`.

```
#version 430

in vec3 VertexPosition;
in vec3 VertexColor;

out vec3 Color;

void main()
{
   Color = VertexColor;
   gl_Position = vec4( VertexPosition, 1.0 );
}
```

Для тех, кому интересно знать, что делает этот код: он действует как «мостик» – принимает атрибуты `VertexPosition` и `VertexColor` и передает их фрагментному шейдеру через переменные `gl_Position` и `Color`.

Далее нужно создать каркас программы OpenGL, используя инструментарий, поддерживающий OpenGL. Примерами таких инструментариев могут служить GLFW, GLUT, FLTK, Qt и wxWidgets. В оставшейся части книги я буду предполагать, что вы в состоянии самостоятельно создать простую программу на основе OpenGL с применением своего любимого инструментария. Фактически все инструменты имеют точки для регистрации функций инициализации, обработчиков изменения размеров окна и обработчиков, осуществляющих рисование (которые вызываются при каждом обновлении окна). Для целей данного рецепта нам нужна программа, создающая и инициализирующая контекст OpenGL, она ничего не должна делать, кроме отображения пустого окна OpenGL. Имейте в виду, что вам также потребуется загрузить указатели на функции OpenGL (см. рецепт «Использование загрузчика функций для доступа к новейшей функциональности OpenGL»).

Наконец, добавьте в программу загрузку исходного кода шейдера в массив символов с именем `shaderCode` и не забудьте добавить нулевой символ в конец! Данный пример предполагает, что переменная `shaderCode` указывает на массив типа `GLchar`, завершающийся нулевым символом.

Как это делается...

Чтобы скомпилировать шейдер, требуется выполнить следующие шаги:

1. Создать объект шейдера:

```
GLuint vertShader = glCreateShader( GL_VERTEX_SHADER );
if( 0 == vertShader )
{
  fprintf(stderr, "Error creating vertex shader.\n");
  exit(EXIT_FAILURE);
}
```

2. Скопировать исходный код (возможно, из нескольких мест) в объект шейдера:

```
const GLchar * shaderCode = loadShaderAsString("basic.vert");
const GLchar* codeArray[] = {shaderCode};
glShaderSource( vertShader, 1, codeArray, NULL );
```

3. Скомпилировать шейдер:

```
glCompileShader( vertShader );
```

4. Проверить результат компиляции:

```
GLint result;
glGetShaderiv( vertShader, GL_COMPILE_STATUS, &result );
if( GL_FALSE == result )
{
  fprintf(stderr, "Vertex shader compilation failed!\n");

  GLint logLen;
  glGetShaderiv(vertShader, GL_INFO_LOG_LENGTH, &logLen);
```

```
    if( logLen > 0 )
    {
      char * log = new char[logLen];

      GLsizei written;
      glGetShaderInfoLog(vertShader, logLen, &written, log);

      fprintf(stderr, "Shader log:\n%s", log);
      delete [] log;
    }
  }
```

Как это работает...

Первый шаг – создать объект шейдера вызовом функции `glCreateShader`. Ее аргумент определяет тип шейдера и может иметь одно из следующих значений: `GL_VERTEX_SHADER`, `GL_FRAGMENT_SHADER`, `GL_GEOMETRY_SHADER`, `GL_TESS_EVALUATION_SHADER`, `GL_TESS_CONTROL_SHADER` или (начиная с версии 4.3) `GL_COMPUTE_SHADER`. В данном случае, так как компилируется вершинный шейдер, функции передается аргумент `GL_VERTEX_SHADER`. Функция возвращает значение, которое можно использовать для ссылки на объект вершинного шейдера, иногда это значение называют «дескриптором», или «описателем». В примере это значение сохраняется в переменной `vertShader`. Если во время создания объекта шейдера возникнет ошибка, функция вернет `0`. В этом случае мы выводим соответствующее сообщение и завершаем программу.

Вслед за созданием объекта шейдера выполняется загрузка исходного кода в него вызовом функции `glShaderSource`. Эта функция принимает массив строк, благодаря чему поддерживается возможность компилировать код сразу из множества источников (файлов, строк). Поэтому перед вызовом `glShaderSource` указатель на строку с исходным кодом помещается в массив с именем `sourceArray`. Первый аргумент `glShaderSource` – дескриптор объекта шейдера. Второй – число строк с исходным кодом в массиве. Третий аргумент – указатель на массив строк с исходным кодом. И последний аргумент – массив значений типа `GLint` с длинами строк в предыдущем аргументе-массиве. В данном примере в четвертом аргументе передается значение `NULL`, указывающее, что все строки с исходным кодом оканчиваются нулевым символом. Если бы строки с исходным кодом не оканчивались нулевым символом, тогда мы должны были бы передать в этом аргументе массив значений типа `GLint`. Обратите внимание, что эта функция копирует исходный код во внутреннюю память OpenGL, поэтому память программы, занятую исходным кодом, можно освободить.

Следующий шаг – компиляция исходного кода шейдера. Делается это простым вызовом функции `glCompileShader` и передачей ей дескриптора шейдера, который требуется скомпилировать. Конечно, при наличии ошибок в исходном коде компиляция может потерпеть неудачу, поэтому далее проверяется успех компиляции.

Проверить успех компиляции можно вызовом `glGetShaderiv` – функции, используемой для получения атрибутов объектов шейдеров. В данном случае тре-

буется получить код завершения компиляции, поэтому во втором аргументе передается GL_COMPILE_STATUS. В первом аргументе передается, конечно же, дескриптор объекта шейдера, а в третьем – указатель на целочисленную переменную, куда следует сохранить код. Функция может записать в эту переменную одно из двух значений – GL_TRUE или GL_FALSE, сообщающих об успехе или неуспехе компиляции, соответственно.

Если функция вернет код завершения компиляции GL_FALSE, из журнала шейдера можно извлечь дополнительную информацию с описанием ошибки. С этой целью мы сначала определяем длину записи в журнале вызовом glGetShaderiv с аргументом GL_INFO_LOG_LENGTH и сохраняем ее в переменной logLen. Обратите внимание, что возвращаемое значение длины уже учитывает заключительный нулевой символ. Далее выделяется память для строки и вызовом glGetShaderInfoLog извлекается запись из журнала. Первый параметр этой функции – дескриптор объекта шейдера, второй – размер буфера символов для сохранения записи из журнала, третий – указатель на целочисленную переменную, куда будет записано фактическое число символов (без учета заключительного нулевого символа), скопированных в буфер, и четвертый аргумент – указатель на буфер символов для самой записи из журнала. После извлечения дополнительной информации она выводится в stderr, после чего занятая память освобождается.

И еще...

В предыдущем примере демонстрировалась компиляция только вершинного шейдера, однако существуют шейдеры других типов, в том числе фрагментные шейдеры, геометрические шейдеры и шейдеры тесселяции. Процедуры компиляции разных типов шейдеров почти не отличаются. Единственным важным отличием является аргумент в вызове функции glCreateShader.

Важно также отметить, что компиляция шейдера – это лишь первый шаг. Чтобы создать действующую шейдерную программу, часто требуется скомпилировать хотя бы два шейдера и затем скомпоновать их в объект шейдерной программы. Шаги, связанные с компоновкой, описываются в следующем рецепте.

Удаление объекта шейдера

Объекты шейдеров можно удалять после того, как они станут не нужны, вызовом glDeleteShader. Эта функция освободит память, занятую шейдером, и сделает недействительным дескриптор, ссылающийся на него. Обратите внимание, что если объект шейдера уже подключен к объекту программы (см. рецепт «Компоновка шейдерной программы»), удаление произойдет только после отключения от объекта программы.

См. также

- Рецепт «Компоновка шейдерной программы».

Компоновка шейдерной программы

После компиляции шейдеров и перед тем, как они фактически будут добавлены в конвейер OpenGL, необходимо скомпоновать их в шейдерную программу. Помимо всего прочего, на этапе компоновки производится стыковка входных переменных одного шейдера с выходными переменными другого, а также стыковка входных/выходных переменных шейдеров с соответствующими областями памяти в окружении OpenGL.

Шаги, выполняемые на этапе компоновки (см. рис. 1.2), делают его похожим на этап компиляции: каждый объект шейдера подключается к новому объекту шейдерной программы, и затем этому объекту дается команда выполнить компоновку (перед компоновкой все шейдеры обязательно должны быть скомпилированы).

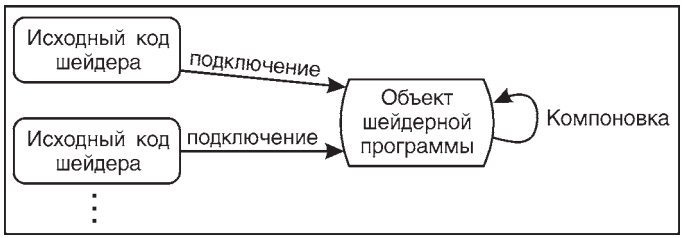

Рис. 1.2 ❖ Процесс компоновки шейдерной программы

Подготовка

В этом рецепте предполагается, что вы уже скомпилировали два объекта шейдеров, дескрипторы которых хранятся в переменных `vertShader` и `fragShader`.

Здесь и в некоторых других рецептах в этой главе будет использоваться следующий фрагментный шейдер:

```
#version 430

in vec3 Color;

out vec4 FragColor;

void main() {
  FragColor = vec4(Color, 1.0);
}
```

Исходный код вершинного шейдера был представлен в предыдущем рецепте «Компиляция шейдера».

Как это делается...

В функции инициализации OpenGL, после компиляции объектов шейдеров, дескрипторы которых хранятся в переменных `vertShader` и `fragShader`, выполняются следующие шаги:

1. Создать объект программы:

```
GLuint programHandle = glCreateProgram();
if( 0 == programHandle )
{
  fprintf(stderr, "Error creating program object.\n");
  exit(1);
}
```

2. Подключить шейдеры к объекту программы:

```
glAttachShader( programHandle, vertShader );
glAttachShader( programHandle, fragShader );
```

3. Скомпоновать программу:

```
glLinkProgram( programHandle );
```

4. Проверить успех компоновки:

```
GLint status;
glGetProgramiv( programHandle, GL_LINK_STATUS, &status );
if( GL_FALSE == status ) {

  fprintf( stderr, "Failed to link shader program!\n" );

  GLint logLen;
  glGetProgramiv(programHandle, GL_INFO_LOG_LENGTH,
         &logLen);
  if( logLen > 0 )
  {
    char * log = new char[logLen];
    GLsizei written;
    glGetProgramInfoLog(programHandle, logLen, &written, log);
    fprintf(stderr, "Program log: \n%s", log);
    delete [] log;
  }
}
```

5. В случае успеха компоновки добавить программу в конвейер OpenGL:

```
else
{
  glUseProgram( programHandle );
}
```

Как это работает...

Сначала вызывается функция `glCreateProgram`, чтобы создать пустой объект программы. Она возвращает дескриптор объекта программы, который мы сохраняем в переменной `programHandle`. Если во время создания объекта программы возникнет ошибка, функция вернет `0`. В этом случае мы выводим соответствующее сообщение и завершаем программу.

Далее выполняется подключение каждого шейдера к объекту программы вызовом функции `glAttachShader`. В первом аргументе ей передается дескриптор объекта программы, а во втором – дескриптор подключаемого объекта шейдера.

Затем производится компоновка программы вызовом функции glLinkProgram, которой в качестве единственного аргумента передается дескриптор объекта программы. Как и в процедуре компиляции, далее проверяется успех процедуры компоновки.

Проверка выполняется вызовом glGetProgramiv. Подобно glGetShaderiv, функция glGetProgramiv позволяет получать значения разных атрибутов шейдерной программы. В данном случае передачей во втором аргументе значения GL_LINK_STATUS запрашивается код завершения компоновки. Код возвращается в переменной, на которую ссылается указатель в третьем аргументе, то есть в переменной status.

Код компоновки может иметь одно из двух значений – GL_TRUE или GL_FALSE, сообщающих об успехе или неуспехе компоновки соответственно. Если возвращается код завершения GL_FALSE, мы извлекаем и отображаем информацию из журнала с описанием ошибки. Извлечение информации осуществляется вызовом функции glGetProgramInfoLog. В первом аргументе ей передается дескриптор объекта программы, во втором – размер буфера символов для сохранения записи из журнала, третий – указатель на переменную типа GLsizei , куда будет записано фактическое число символов (без учета заключительного нулевого символа), скопированных в буфер, и четвертый аргумент – указатель на буфер символов для самой записи из журнала. Необходимый объем памяти для буфера можно определить вызовом glGetProgramiv с аргументом GL_INFO_LOG_LENGTH. Строка, возвращаемая в переменной log, уже будет оканчиваться нулевым символом.

Наконец, если компоновка прошла успешно, программа добавляется в конвейер OpenGL вызовом функции glUseProgram, которой в качестве аргумента передается дескриптор объекта программы.

После компиляции фрагментного шейдера из этого рецепта и вершинного шейдера из предыдущего рецепта, компоновки и добавления программы в конвейер OpenGL мы получаем законченный конвейер OpenGL и готовы приступить к отображению. Если попытаться нарисовать треугольник с разными значениями в атрибуте Color, мы получим изображение многоцветного треугольника с вершинами красного, зеленого и синего цветов и с плавными переходами цветов внутри, как показано на рис. 1.3.

И еще...

В одной программе можно создать множество шейдерных программ. Их можно менять в конвейере OpenGL вызовом функции glUseProgram, переключаясь на нужную программу.

Удаление шейдерной программы

Если программа стала не нужна, ее можно удалить из памяти OpenGL вызовом функции glDeleteProgram, передав ей дескриптор объекта программы в единственном аргументе. Это сделает дескриптор программы недействительным и освободит память, занятую программой. Обратите внимание, что если в настоящий момент программа используется, она будет удалена, только когда перестанет использоваться.

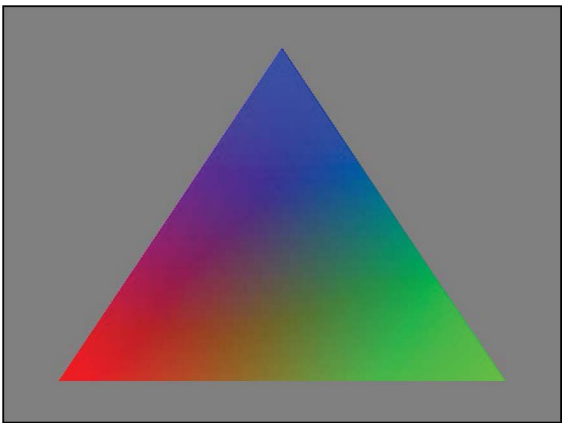

Рис. 1.3 ❖ Многоцветный треугольник с плавными переходами цветов

Удаление программы приведет также к отключению объектов шейдеров, не не удалит их, если прежде не была выполнена операция удаления шейдеров вызовом функции `glDeleteShader`.

См. также

- Рецепт «Компиляция шейдера».

Передача данных в шейдер с использованием вершинных атрибутов и вершинных буферных объектов

Вершинный шейдер вызывается один раз для каждой вершины. Его главная задача – обработать данные, ассоциированные с вершиной, и передать их (и, возможно, другую информацию) на следующий этап конвейера. Чтобы передать вершинному шейдеру что-то для обработки, необходим некоторый механизм передачи исходных данных (для каждой вершины) в шейдер. Обычно такие исходные данные включают (помимо всего прочего) координаты вершины, вектор нормали и координаты текстуры. В прежних версиях OpenGL (до версии 3.0) для передачи каждого элемента информации о вершине имелись отдельные «каналы» – различные функции, такие как `glVertex`, `glTexCoord` и `glNormal` (или внутри клиентских массивов вершин, с использованием `glVertexPointer`, `glTexCoordPointer` или `glNormalPointer`). Шейдер мог обращаться к значениям, переданным таким способом, посредством встроенных переменных, таких как `gl_Vertex` и `gl_Normal`. Но в версии OpenGL 3.0 эта функциональность была объявлена устаревшей и позднее удалена из библиотеки. Теперь информация о вершинах должна передаваться

через *универсальные вершинные атрибуты*, обычно в комплексе с (вершинными) *буферными объектами*. Программист сейчас может определить произвольный набор атрибутов для каждой вершины с целью передачи информации в вершинный шейдер. Например, чтобы реализовать рельефное текстурирование методом наложения нормалей (normal mapping), программист может решить передавать для каждой вершины не только ее координаты, но также вектор нормали и вектор касательной. В OpenGL 4 это легко реализовать, определив комплект входных атрибутов. Такой подход определения информации для каждой вершины не только обеспечивает большую гибкость, но и может потребовать времени на освоение для тех, кто привык пользоваться устаревшими приемами.

В вершинном шейдере определение входных атрибутов выполняется с использованием квалификатора `in`. Например, определить входной атрибут с трехкомпонентным вектором `VertexColor` можно следующим образом:

```
in vec3 VertexColor;
```

Разумеется, данные для этого атрибута должны поставляться программой OpenGL. Для этого используются вершинные буферные объекты (vertex buffer objects). Буферный объект содержит значения для входного атрибута. Основная программа связывает буферный объект с входным атрибутом и определяет, как «шагать» по данным. Затем во время отображения OpenGL будет извлекать из буфера данные для атрибута перед каждым вызовом вершинного шейдера.

В данном рецепте рисуется единственный треугольник. В число необходимых нам атрибутов входят координаты вершины и цвет. Мы будем использовать фрагментный шейдер для интерполяции цветов вершин, чтобы получился треугольник, как показано на рис. 1.4. Вершины треугольника имеют красный, зеленый и синий цвета, а тело раскрашивается путем смешивания этих цветов. Возможно, на бумаге цвета не видны (в черно-белой книге), но по градациям серого цвета можно догадаться о смешивании цветов.

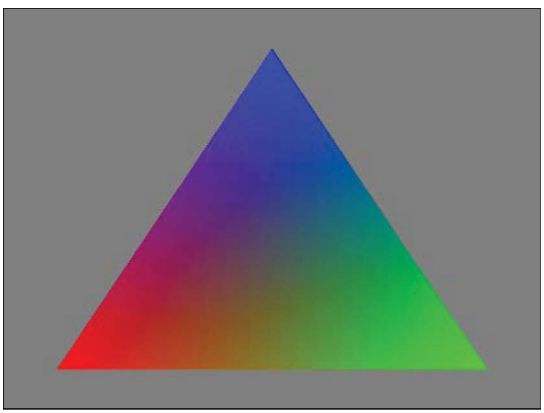

Рис. 1.4 ❖ Результат работы шейдеров в данном рецепте

Подготовка

Начнем с создания пустой программы OpenGL и следующих ниже шейдеров.

Вершинный шейдер (`basic.vert`):

```
#version 430

layout (location=0) in vec3 VertexPosition;
layout (location=1) in vec3 VertexColor;

out vec3 Color;

void main()
{
  Color = VertexColor;

  gl_Position = vec4(VertexPosition,1.0);
}
```

Атрибуты – это входные переменные для вершинного шейдера. В предыдущем фрагменте имеются два входных атрибута: `VertexPosition` и `VertexColor`. Они объявляются с использованием ключевого слова `in`. Пусть вас пока не смущает префикс `layout` – мы подробно обсудим его ниже. Основная программа должна организовать передачу данных в эти два атрибута для каждой вершины. Мы реализуем это путем отображения массива данных на эти переменные.

В шейдере имеется также одна выходная переменная `Color`, значение которой будет передаваться во фрагментный шейдер. В данном случае через переменную `Color` просто передается копия `VertexColor`. Отметьте также, что атрибут `VertexPosition` просто дополняется еще одним постоянным компонентом и передается во встроенную выходную переменную `gl_Position` для дальнейшей обработки.

Фрагментный шейдер (`basic.frag`):

```
#version 430

in vec3 Color;

out vec4 FragColor;

void main() {
  FragColor = vec4(Color, 1.0);
}
```

В этом шейдере имеется лишь одна входная переменная `Color`. Она связана с соответствующей выходной переменной в вершинном шейдере и будет содержать значение, полученное в результате интерполяции цветов трех вершин треугольника. Мы просто дополняем копию цвета еще одним компонентом и передаем ее в переменную `FragColor` (подробнее о выходных переменных фрагментного шейдера рассказывается в последующих рецептах).

Напишите код, осуществляющий компиляцию и компоновку шейдеров в шейдерную программу (см. рецепты «Компиляция шейдера» и «Компоновка шейдер-

ной программы»). В программном коде, следующем ниже, предполагается, что дескриптор объекта шейдерной программы хранится в переменной `programHandle`.

Как это делается...

Настройка буферных объектов и отображение треугольника выполняются в несколько этапов:

1. Создать глобальную (или приватную для экземпляра) переменную для хранения дескриптора объекта массива вершин:

```
GLuint vaoHandle;
```

2. Внутри функции инициализации создать и заполнить вершинные буферные объекты, по одному для каждого атрибута:

```
float positionData[] = {
      -0.8f, -0.8f, 0.0f,
      0.8f, -0.8f, 0.0f,
      0.0f, 0.8f, 0.0f };
float colorData[] = {
      1.0f, 0.0f, 0.0f,
      0.0f, 1.0f, 0.0f,
      0.0f, 0.0f, 1.0f };

// Создать и заполнить буферные объекты
GLuint vboHandles[2];
glGenBuffers(2, vboHandles);
GLuint positionBufferHandle = vboHandles[0];
GLuint colorBufferHandle = vboHandles[1];

// Заполнить буфер координат
glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float),
      positionData, GL_STATIC_DRAW);

// Заполнить буфер цветов
glBindBuffer(GL_ARRAY_BUFFER, colorBufferHandle);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float), colorData,
      GL_STATIC_DRAW);
```

3. Создать объект массива вершин, определяющий отношения между буферами и входными атрибутами. (См. раздел «И еще...» ниже, где описывается альтернативный способ, поддерживаемый в версиях OpenGL 4.3 и выше.)

```
// Создать объект массива вершин
glGenVertexArrays( 1, &vaoHandle );
glBindVertexArray(vaoHandle);

// Активировать массивы вершинных атрибутов
glEnableVertexAttribArray(0); // Координаты вершины
glEnableVertexAttribArray(1); // Цвет вершины

// Закрепить индекс 0 за буфером с координатами
glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
```

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);

// Закрепить индекс 1 за буфером с цветом
glBindBuffer(GL_ARRAY_BUFFER, colorBufferHandle);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

4. В функции отображения следует связать объект массива вершин и вызвать `glDrawArrays`, чтобы запустить процедуру рисования:

```
glBindVertexArray(vaoHandle);
glDrawArrays(GL_TRIANGLES, 0, 3 );
```

Как это работает...

Атрибуты вершин – это входные переменные для вершинного шейдера. В данном вершинном шейдере имеются два атрибута: `VertexPosition` и `VertexColor`. Основная программа ссылается на атрибуты вершин путем связывания каждой (активной) входной переменной с индексом атрибута. Роль индексов играют простые целые числа в диапазоне от 0 до `GL_MAX_VERTEX_ATTRIBS - 1`. Соответствие между этими индексами и атрибутами можно установить с помощью квалификатора `layout`. Например, в нашем вершинном шейдере с помощью квалификатора `layout` атрибуту `VertexPosition` присваивается индекс `0`, а атрибуту `VertexColor` – индекс `1`.

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexColor;
```

Ссылка на атрибуты вершин в основной программе производится с применением соответствующих индексов.

Вообще говоря, можно и не назначать соответствия между переменными-атрибутами и индексами, потому что OpenGL автоматически отобразит активные атрибуты вершин в индексы во время компоновки программы. В этой ситуации мы могли бы получить карту индексов и по ней определить соответствия между индексами и входными переменными. Однако явное назначение индексов выглядит более очевидно.

На первом шаге осуществляется подготовка пары буферных объектов для хранения информации о координатах и цвете. Как и в случае с большинством объектов OpenGL, сначала, вызовом функции `glGenBuffers`, создаются сразу два буферных объекта. Затем дескрипторы объектов сохраняются в отдельных переменных, чтобы сделать следующий далее код более ясным.

Каждый буферный объект сначала связывается с точкой привязки `GL_ARRAY_BUFFER` вызовом `glBindBuffer`. В первом аргументе функции `glBindBuffer` передается целевая точка привязки. В данном случае, так как данные представляют собой самые обычные массивы, используется точка привязки `GL_ARRAY_BUFFER`. Примеры других точек привязки (таких как `GL_UNIFORM_BUFFER` или `GL_ELEMENT_ARRAY_BUFFER`) вы увидите в последующих рецептах. После привязки буферного объекта его можно заполнить соответствующими данными с помощью `glBufferData`. Во втором и третьем аргументах этой функции передаются размер массива и указатель на массив с данными. Но давайте сосредоточимся на первом и последнем аргументах. В первом аргументе указывается целевая точка привязки буферного объекта. Дан-

ные, переданные в третьем аргументе, копируются в буфер, связанный с данной точкой привязки. Последний аргумент подсказывает библиотеке OpenGL, как будут использоваться данные, чтобы она могла решить, как лучше организовать управление внутренним буфером. Более подробное описание этого аргумента можно найти в документации по OpenGL. В нашем случае данные определяются один раз, не будут изменяться в дальнейшем и будут использоваться многократно в операциях рисования. Такому шаблону использования лучше соответствует значение `GL_STATIC_DRAW`.

Теперь, после настройки буферных объектов, их нужно объединить в **объект массива вершин (Vertex Array Object, VAO)**. VAO содержит информацию о связях между данными в буферах и входными атрибутами вершин. Создается объект VAO с помощью функции `glGenVertexArrays`. Она возвращает дескриптор нового объекта, который мы сохраняем в (глобальной) переменной `vaoHandle`. Затем мы активируем атрибуты вершин с индексами 0 и 1 вызовом `glEnableVertexAttribArray`. Тем самым мы указываем, что значения атрибутов будут доступны и использованы при отображении.

На следующем шаге устанавливается связь между буферными объектами и индексами атрибутов.

```
// Закрепить индекс 0 за буфером с координатами
glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, 0, NULL );
```

Сначала мы связываем буферный объект с целевой точкой привязки `GL_ARRAY_BUFFER`, а затем вызываем функцию `glVertexAttribPointer`, которая сообщит OpenGL индекс, соответствующий данным, формат данных, хранящихся в буферном объекте, и где хранятся эти данные в буферном объекте, связанном с точкой привязки `GL_ARRAY_BUFFER`. Первый аргумент – это индекс атрибута. Второй – число компонентов на каждую вершину (1, 2, 3 или 4). В данном случае передаются трехмерные данные, поэтому на каждую вершину приходится три компонента. Третий аргумент определяет тип данных каждого компонента в буфере. Четвертый – логическое значение, определяющее необходимость нормализации данных (то есть необходимость отображения целочисленных значений со знаком в диапазон `[-1, 1]` и целочисленных значений без знака – в диапазон `[0, 1]`). Пятый аргумент – это шаг, смещение в байтах между соседними атрибутами. Так как наши данные плотно упакованы, мы указываем в этом аргументе нуль. Последний аргумент – указатель, который интерпретируется не как указатель! Значение этого аргумента интерпретируется как смещение в байтах первого атрибута в буфере относительно начала этого буфера. В данном случае перед первым элементом в буфере нет никаких дополнительных данных, поэтому в примере указано значение `0`.

Функция `glVertexAttribPointer` сохранит указатель на буфер (в VAO), в настоящее время связанный с точкой привязки `GL_ARRAY_BUFFER`. Если с данной точкой привязки связать другой буфер, это не изменит значения указателя.

Объект массива вершин (VAO) хранит всю информацию, связанную с отношениями между буферными объектами и атрибутами вершин, а также информацию

о формате данных в буферных объектах. Это позволяет быстро получить все необходимые сведения во время отображения. Объект VAO играет чрезвычайно важную роль, но понять ее очень непросто. Важно запомнить, что информация в VAO в первую очередь имеет отношение к активным атрибутам и их связям с буферными объектами. Это не означает, что объект VAO хранит информацию о привязках буферов. Например, связь с точкой привязки `GL_ARRAY_BUFFER` не запоминается. Мы выполняем связывание с этой точкой привязки, только чтобы сохранить указатели вызовом `glVertexAttribPointer`.

После настройки объекта VAO (это однократная операция) можно выполнять команды рисования для вывода изображения. В функции вывода выполняется очистка буфера цвета вызовом `glClear`, устанавливается связь с объектом массива вершин и вызывается функция `glDrawArrays`, которая рисует треугольник. Функция `glDrawArrays` инициализирует вывод примитивов, выполняя обход всех активных атрибутов в буферах и передавая данные далее по конвейеру в вершинный шейдер. Первый аргумент определяет режим отображения (в данном случае мы рисуем треугольники), во втором аргументе указывается первый активный индекс в массивах атрибутов, в третьем – число индексов для отображения (для единственного треугольника – 3 вершины).

Если говорить более кратко, мы выполняем следующие шаги:

1. С помощью квалификатора `layout` каждому атрибуту в вершинном шейдере назначается свой индекс.
2. Для каждого атрибута создается и заполняется данными свой буферный объект.
3. Пока соответствующий буфер связан с целевой точкой привязки, вызовом `glVertexAttribPointer` создается и определяется объект массива вершин.
4. Перед отображением выполняется связывание объекта массива вершин и вызывается функция `glDrawArrays` или другая подобная функция рисования (например, `glDrawElements`).

И еще...

Далее мы обсудим некоторые детали, расширения и альтернативы предыдущих приемов.

Отдельное определение форматов атрибутов

В OpenGL 4.3 появился альтернативный (необязательно лучший) способ определения состояния объекта массива вершин (формат атрибутов, активные атрибуты и буферы). В предыдущем примере вызов функции `glVertexAttribPointer` решает две важные задачи. Во-первых, она косвенно определяет, какой буфер содержит данные для атрибута, какой буфер в настоящий момент (на момент вызова) связан с `GL_ARRAY_BUFFER`. Во-вторых, она определяет формат данных (тип, смещение, шаг и другие характеристики). Возможно, код будет выглядеть яснее, если каждую из задач решать вызовом отдельной функции. Именно это решение и было реализовано в OpenGL 4.3. Например, ниже показано, как можно реализовать шаг 3 из предыдущего раздела «Как это делается...»:

```
glGenVertexArrays(1, &vaoHandle);
glBindVertexArray(vaoHandle);
glEnableVertexArray(0);
glEnableVertexArray(1);

glBindVertexBuffer(0, positionBufferHandle, 0, sizeof(GLfloat)*3);
glBindVertexBuffer(1, colorBufferHandle, 0, sizeof(GLfloat)*3);

glVertexAttribFormat(0, 3, GL_FLOAT, GL_FALSE, 0);
glVertexAttribBinding(0, 0);
glVertexAttribFormat(1, 3, GL_FLOAT, GL_FALSE, 0);
glVertexAttribBinding(1, 1);
```

Первые четыре строки в предыдущем фрагменте кода в точности повторяют код из первого примера. Он создает и связывает VAO, затем активирует атрибуты 0 и 1. Далее, с помощью `glBindVertexBuffer`, два буфера связываются с двумя разными индексами внутри точки привязки вершинных буферов (vertex buffer binding point). Обратите внимание, что здесь больше не используется точка привязки `GL_ARRAY_BUFFER`. Вместо нее у нас теперь имеется новая точка привязки, предназначенная для вершинных буферов. Эта точка привязки имеет несколько индексов (обычно от 0 до 15), что дает возможность привязывать к ней несколько буферов. В первом аргументе функции `glBindVertexBuffer` передается индекс в точке привязки вершинных буферов. В данном примере буферу с координатами назначается индекс `0`, а буферу со значениями цвета – индекс `1`.

Имейте в виду, что индексы внутри точки привязки вершинных буферов необязательно должны совпадать с индексами атрибутов.

Во втором аргументе функции `glBindVertexBuffer` передается сам буфер, в третьем – смещение от начала буфера до начала данных, в четвертом – шаг, то есть расстояние в байтах между началами соседних элементов в буфере. В отличие от `glVertexAttribPointer`, здесь нельзя передать `0` в четвертом аргументе, потому что OpenGL не в состоянии определить размер данных без дополнительной информации, поэтому мы вынуждены указывать величину шага явно.

Далее вызывается функция `glVertexAttribFormat`, с помощью которой задается формат данных для атрибута. Обратите внимание, что на этот раз не указывается буфер, хранящий данные, – мы просто указываем формат данных для этого атрибута. Аргументы имеют ту же семантику, что и первые четыре аргумента функции `glVertexAttribPointer`.

Функция `glVertexAttribBinding` определяет отношения между буферами, связанными с точкой привязки вершинных буферов, и атрибутами. В первом аргументе передается индекс атрибута, а во втором – индекс внутри точки привязки вершинных буферов. В данном примере они совпадают, но их совпадение не является обязательным условием.

Отметьте также, что привязки буферов к точке привязки вершинных буферов (задаются с помощью `glBindVertexBuffer`) являются частью состояния VAO, в отличие от привязки к точке `GL_ARRAY_BUFFER`.

Кому-то такой вариант покажется более ясным и более простым для понимания. Он устраняет аспекты, связанные с «невидимыми» указателями в VAO, и де-

лает отношения между атрибутами и буферами намного более очевидными. Кроме того, он разделяет задачи, которые в действительности не связаны между собой.

Выходные переменные фрагментного шейдера

Возможно, кто-то заметил, что я ничего не сказал о выходной переменной `FragColor` во фрагментном шейдере. В эту переменную записывается окончательное значение цвета для каждого фрагмента (пикселя). По аналогии с входными переменными в вершинном шейдере этой переменной также должен быть назначен соответствующий индекс. Обычно эту переменную принято связывать с фоновым буфером цвета, по умолчанию (в системах с поддержкой двойной буферизации) имеющим нулевой «индекс». (Соответствия между индексами и буферами отображения можно изменить с помощью `glDrawBuffers`.) В данной программе мы опираемся на тот факт, что компоновщик автоматически присвоит нулевой индекс единственной выходной переменной фрагментного шейдера. Чтобы сделать это явно, мы могли бы (и, наверное, должны) добавить в объявление переменной квалификатор `layout`:

```
layout (location = 0) out vec4 FragColor;
```

Благодаря такому подходу мы свободно можем определить во фрагментном шейдере множество выходных переменных для отображения множества выходных буферов. Это может пригодиться для реализации специализированных алгоритмов, таких как отложенное отображение (см. главу 5 «Обработка изображений и приемы работы с экранным пространством»).

Назначение индексов без использования квалификатора layout

Если у вас нет желания загромождать исходный код вершинного шейдера квалификаторами `layout` (или используемая версия OpenGL не поддерживает их), есть возможность назначить индексы атрибутам внутри программы OpenGL. Сделать это можно вызовом функции `glBindAttribLocation` непосредственно перед компоновкой шейдерной программы. Например, в нашу программу можно было бы добавить следующий код непосредственно перед операцией компоновки:

```
glBindAttribLocation(programHandle, 0, "VertexPosition");
glBindAttribLocation(programHandle, 1, "VertexColor");
```

Эти вызовы функций подсказали бы компоновщику, что атрибуту `VertexPosition` должен соответствовать индекс `0`, а атрибуту `VertexColor` – индекс `1`.

Аналогично можно назначить индекс выходной переменной фрагментного шейдера, не прибегая к квалификатору `layout`. Для этого можно вызвать `glBindFragDataLocation` непосредственно перед операцией компоновки шейдерной программы:

```
glBindFragDataLocation(programHandle, 0, "FragColor");
```

Этот вызов подсказал бы компоновщику, что выходной переменной `FragColor` должен соответствовать индекс `0`.

Массивы элементов

Часто бывает желательно совершить обход массива вершин не по порядку. То есть может потребоваться «перепрыгивать взад и вперед», вместо того чтобы перемещаться по ним последовательно, как реализовано в данном примере. Например, такой прием может пригодиться для рисования куба по восьми вершинам (углам куба). В общем случае, чтобы изобразить куб, нужно нарисовать 12 треугольников (по два на каждую грань), каждый из которых определяется тремя вершинами. Все необходимые данные для этого можно получить из координат восьми вершин, но, чтобы нарисовать все треугольники, нужно информацию о каждой вершине использовать при рисовании, по меньшей мере, трех разных треугольников.

Обеспечить такие «прыжки» по массиву вершин можно с помощью массива элементов. Массив элементов – это еще один буфер, определяющий последовательность обхода индексов в массиве вершин. Более подробную информацию об использовании массивов элементов можно найти в документации OpenGL, в описании функции `glDrawElements` (**http://www.opengl.org/sdk/docs/man**).

Перемежение разнородных данных в массивах

В данном примере использовались два буфера (один – для передачи цвета и один – для передачи координат). Вместо двух буферов можно было бы использовать один и объединить в нем сразу все данные. Вообще, в единственном буфере можно объединить данные для любого числа атрибутов. В этом случае разнородные данные будут перемежаться в массиве и группироваться по вершинам. Однако тогда придется с особым вниманием отнестись к использованию аргумента `stride` функций `glVertexAttribPointer` и `glBindVertexBuffer`. За более подробной информацией обращайтесь к документации (**http://www.opengl.org/sdk/docs/man**).

Принятие решения о том, когда использовать объединенные массивы, а когда отдельные, во многом зависит от конкретной ситуации. Применение объединенных массивов может способствовать увеличению производительности, благодаря тому что все данные с информацией о каждой вершине хранятся в памяти рядом друг с другом (известный принцип близости ссылок).

Получение списка активных атрибутов и их индексов

Как уже говорилось в предыдущем рецепте, входным переменным вершинного шейдера на этапе компоновки программы назначаются индексы. При необходимости вручную определить отношения между переменными и индексами можно воспользоваться квалификатором `layout` внутри шейдера или вызвать `glBindAttribLocation` перед компоновкой.

Однако иногда бывает предпочтительнее позволить компоновщику автоматически назначить индексы переменным и затем, после компоновки, запрашивать их. В этом рецепте представлен простой пример, который выводит все активные атрибуты и их индексы.

Подготовка

Для демонстрации нам потребуется программа, компилирующая и компонующая пару шейдеров. В качестве основы можно использовать шейдеры из предыдущего рецепта.

Как и в предыдущих рецептах, мы будем предполагать, что дескриптор шейдерной программы хранится в переменной `programHandle`.

Как это делается...

После компоновки и активации шейдерной программы добавьте следующий код, отображающий список активных атрибутов:

1. Запросить число активных атрибутов:

```
GLint numAttribs;
glGetProgramInterfaceiv(programHandle, GL_PROGRAM_INPUT,
       GL_ACTIVE_RESOURCES, &numAttribs);
```

2. Обойти в цикле все атрибуты и для каждого запросить длину имени, тип и индекс, вывести результаты в `stdout`:

```
GLenum properties[] = {GL_NAME_LENGTH, GL_TYPE,
      GL_LOCATION};

printf("Active attributes:\n");
for( int i = 0; i < numAttribs; ++i ) {
  GLint results[3];
  glGetProgramResourceiv(programHhandle, GL_PROGRAM_INPUT,
       i, 3, properties, 3, NULL, results);

  GLint nameBufSize = results[0] + 1;
  char * name = new char[nameBufSize];
  glGetProgramResourceName(programHandle,
      GL_PROGRAM_INPUT, i, nameBufSize, NULL, name);
  printf("%-5d %s (%s)\n", results[2],
      name, getTypeString(results[1]));
  delete [] name;
}
```

Как это работает...

На первом шаге вызовом `glGetProgramInterfaceiv` запрашивается число активных атрибутов. В первом аргументе этой функции передается дескриптор объекта программы, а во втором (`GL_PROGRAM_INPUT`) – признак, что запрашивается информация о входных переменных (атрибутах вершин). Третий аргумент (`GL_ACTIVE_RESOURCES`) указывает, что нас интересует число активных ресурсов. В последнем аргументе передается адрес переменной `numAttribs`, куда следует сохранить результат.

После получения числа атрибутов можно приступать к извлечению информации об отдельных атрибутах. Порядковые номера атрибутов изменяются в диапазоне от `0` до `numAttribs-1`. Программа выполняет цикл по этим порядковым номерам и для каждого вызывает функцию `glGetProgramResourceiv`, чтобы получить длину

имени атрибута, его тип и индекс. Мы указываем, какую информацию хотим получить, передавая массив properties значений типа GLenum. В первом аргументе функции передается дескриптор объекта программы, во втором – тип запрашиваемого ресурса (GL_PROGRAM_INPUT), в третьем – порядковый номер атрибута, в четвертом – число запрашиваемых свойств, в пятом – массив, перечисляющий интересующие нас свойства. Массив properties содержит значения типа GLenum, определяющие конкретные свойства атрибута. В данном примере массив содержит значения GL_NAME_LENGTH, GL_TYPE и GL_LOCATION, указывающие, что запрашиваются длина имени атрибута, тип данных атрибута и его индекс соответственно. В шестом аргументе передается размер буфера для приема результатов; в седьмом – адрес целочисленной переменой, куда следует сохранить число принятых результатов. Если в этом аргументе передать NULL, эта информация возвращаться не будет. Наконец, в последнем аргументе передается указатель на массив GLint, куда должны быть сохранены результаты. Каждому элементу в массиве properties соответствует элемент в массиве results с тем же индексом.

Далее извлекается имя атрибута, для чего сначала выделяется память под буфер с именем, который затем будет заполнен вызовом функции glGetProgramResourceName. Длина имени содержится в первом элементе массива results, соответственно мы выделяем буфер данного размера плюс дополнительный символ в качестве меры предосторожности. В документации по OpenGL утверждается, что glGetProgramResourceiv возвращает размер имени уже с учетом завершающего пустого символа, но программе совершенно не повредит выделить чуть больше памяти. Кстати, при тестировании последних драйверов компании NVIDIA я обнаружил, что эта мера предосторожности совсем не лишняя.

Наконец вызывается функция glGetProgramResourceName, после чего на экран выводится полученная информация. Сначала выводится индекс атрибута, затем его имя и тип. Индекс атрибута хранится в третьем элементе массива results, а тип – во втором. Обратите внимание на вызов функции getTypeString. Это наша простая функция, возвращающая строку, представляющую тип данных. Тип данных в массиве results представлен одним из значений: GL_FLOAT, GL_FLOAT_VEC2, GL_FLOAT_VEC3 и т. д. Функция getTypeString состоит из одной большой инструкции switch, возвращающей строку с названием типа, соответствующим значению параметра (см. исходный код в файле glslprogram.cpp, который можно найти в загружаемом пакете примеров для книги).

Ниже показан вывод описанного фрагмента, когда он выполняется в программе с шейдерами из предыдущего рецепта:

```
Active attributes:
1    VertexColor (vec3)
0    VertexPosition (vec3)
```

И еще...

Следует отметить следующее: чтобы входная переменная вершинного шейдера считалась активной, она должна использоваться внутри шейдера. Иными слова-

ми, переменная считается активной, если компоновщик GLSL обнаружит, что она будет применяться во время выполнения программы. Если переменная объявлена в шейдере, но не используется, предыдущий код не выведет информацию о ней, потому что OpenGL посчитает ее неактивной и фактически не будет ее использовать.

Обратите также внимание, что предыдущий код допустим только для версии OpenGL 4.3 или выше. В более ранних версиях аналогичный результат можно получить с помощью функций `glGetProgramiv`, `glGetActiveAttrib` и `glGetAttribLocation`.

См. также

- Рецепт «Компиляция шейдера».
- Рецепт «Компоновка шейдерной программы».
- Рецепт «Передача данных в шейдер с использованием вершинных атрибутов и вершинных буферных объектов».

Передача данных в шейдер с использованием uniform-переменных

Переменные-атрибуты – лишь один из способов передачи входных данных в шейдеры; второй способ заключается в использовании uniform-переменных. Переменные этого вида применяются для передачи данных, которые не изменяются от вершины к вершине, подобно переменным-атрибутам. Более того, uniform-переменные нельзя даже изменить при переходе от одной вершины к другой. Эти переменные отлично подходят, например, для передачи матриц модели, вида и проекции.

Внутри шейдера uniform-переменные доступны только для чтения. Их значения могут изменяться лишь за пределами шейдера, посредством функций OpenGL. Однако их можно инициализировать внутри шейдера, присваивая им константные значения в объявлениях.

Uniform-переменные могут присутствовать в любых шейдерах внутри шейдерной программы, и они всегда используются как входные переменные. Они могут объявляться в одном или в нескольких шейдерах, но если переменная с одним и тем же именем объявлена более чем в одном шейдере, все одноименные переменные должны иметь один и тот же тип во всех шейдерах. Иными словами, uniform-переменные разделяют общее uniform-пространство имен, простирающееся по всей шейдерной программе.

В данном рецепте мы нарисуем тот же треугольник, что и в предыдущих рецептах, но на этот раз мы повернем его с применением матрицы в uniform-переменной, как показано на рис. 1.5.

Подготовка

В этом рецепте используется следующий вершинный шейдер:

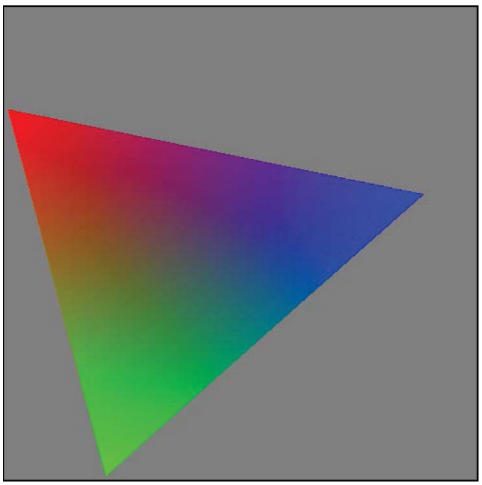

Рис. 1.5 ❖ Треугольник после поворота

```
#version 430

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexColor;

out vec3 Color;

uniform mat4 RotationMatrix;

void main()
{
  Color = VertexColor;
  gl_Position = RotationMatrix * vec4(VertexPosition,1.0);
}
```

Обратите внимание, что переменная `RotationMatrix` объявлена с квалификатором `uniform`. Запись данных в эту переменную будет осуществляться в основной программе. Переменная `RotationMatrix` используется для преобразования координат в `VertexPosition` перед передачей их в стандартную выходную переменную `gl_Position`.

Фрагментный шейдер взят из предыдущих рецептов без изменений:

```
#version 430

in vec3 Color;

layout (location = 0) out vec4 FragColor;

void main() {
  FragColor = vec4(Color, 1.0);
}
```

Основная программа определяет матрицу поворота и передает ее в шейдер через uniform-переменную. Создание матрицы поворота реализовано с применением библиотеки GLM (см. рецепт «Использование GLM для математических вычислений» выше). В исходный код основной программы добавьте следующие инструкции:

```
#include <glm/glm.hpp>
using glm::mat4;
using glm::vec3;

#include <glm/gtc/matrix_transform.hpp>
```

Также в рецепте будет предполагаться, что программа уже имеет код, выполняющий компиляцию и компоновку шейдеров и создающий объект массива вершин для цветного треугольника. Еще предполагается, что дескриптор объекта массива вершин хранится в переменной `vaoHandle`, а дескриптор объекта программы – в переменной `programHandle`.

Как это делается...

Добавьте следующий код в метод отображения:

```
glClear(GL_COLOR_BUFFER_BIT);

mat4 rotationMatrix = glm::rotate(mat4(1.0f), angle,
      vec3(0.0f,0.0f,1.0f));

GLuint location = glGetUniformLocation(programHandle,
      "RotationMatrix");

if( location >= 0 )
{
  glUniformMatrix4fv(location, 1, GL_FALSE,
      &rotationMatrix[0][0]);
}

glBindVertexArray(vaoHandle);
glDrawArrays(GL_TRIANGLES, 0, 3 );
```

Как это работает...

Для записи значения в uniform-переменную необходимо определить ее индекс и затем присвоить значение по этому индексу вызовом функции `glUniform`.

Фрагмент выше начинается с очистки буфера цвета. Затем он создает матрицу поворота с помощью GLM. Далее вызовом функции `glGetUniformLocation` определяется индекс uniform-переменной. Эта переменная принимает дескриптор объекта шейдерной программы и имя uniform-переменной и возвращает ее индекс. Если искомая переменная неактивна, функция вернет `-1`.

Затем вызовом `glUniformMatrix4fv` выполняется присваивание значения uniform-переменной. В первом аргументе передается индекс переменной. Во втором – число присваиваемых матриц (имейте в виду, что uniform-переменная

может быть массивом). В третьем – логическое значение, указывающее на необходимость транспонирования матрицы перед записью в uniform-переменную. Для матриц, созданных с помощью GLM, транспонирование не требуется, поэтому здесь в третьем аргументе передается значение GL_FALSE. Если бы матрица была создана вручную, как массив, в котором данные располагаются построчно, тогда в этом аргументе необходимо было бы передать GL_TRUE. Последний аргумент – указатель на данные для записи в uniform-переменную.

И еще...

Разумеется, uniform-переменные могут быть любого типа, допустимого в языке GLSL, включая составные типы, такие как массивы или структуры. Библиотека OpenGL предоставляет семейство функций glUniform, имена которых оканчиваются суффиксами, соответствующими поддерживаемым типам. Например, чтобы присвоить значение переменной типа vec3, следует воспользоваться функцией glUniform3f или glUniform3fv.

Передачу массивов можно осуществлять с помощью функций, имена которых оканчиваются на "v". Обратите внимание, что для доступа к отдельным элементам uniform-массива можно использовать оператор []. Например, ниже показано, как получить ссылку на второй элемент массива MyArray:

```
GLuint location =
  glGetUniformLocation( programHandle, "MyArray[1]" );
```

Члены структур должны инициализироваться по отдельности. Ниже показано, как можно запросить ссылку на поле структуры:

```
GLuint location =
  glGetUniformLocation( programHandle, "MyMatrices.Rotation" );
```

где MyMatrices – имя переменной, в которой хранится структура, а Rotation – имя поля в структуре.

См. также

- Рецепт «Компиляция шейдера».
- Рецепт «Компоновка шейдерной программы».
- Рецепт «Передача данных в шейдер с использованием вершинных атрибутов и вершинных буферных объектов».

Получение списка активных uniform-переменных

Получить индекс отдельной uniform-переменной несложно, однако иногда бывает полезно иметь возможность получить список всех активных uniform-переменных. Например, в основной программе можно создать множество переменных для хранения ссылок на все uniform-переменные, с помощью которых можно будет присваивать этим uniform-переменным новые значения после компоновки программы. Такой подход поможет избежать необходимости запрашивать индексы

uniform-переменных перед записью новых значений и за счет этого повысить производительность программного кода.

Процедура получения списка uniform-переменных очень похожа на получение списка атрибутов (см. рецепт «Получение списка активных атрибутов и их индексов»), поэтому за подробными разъяснениями мы рекомендуем обращаться к предыдущему рецепту.

Подготовка

Для демонстрации нам потребуется программа, компилирующая и компонующая шейдерную программу. Как и прежде, мы будем предполагать, что дескриптор программы хранится в переменной `programHandle`.

Как это делается...

После компоновки и активации шейдерной программы добавьте следующий код, отображающий список активных uniform-переменных:

1. Запросить число активных uniform-переменных:

```
GLint numUniforms = 0;
glGetProgramInterfaceiv( handle, GL_UNIFORM,
  GL_ACTIVE_RESOURCES, &numUniforms);
```

2. Обойти в цикле все uniform-переменные и для каждой запросить длину имени, тип, ссылку и индекс блока:

```
GLenum properties[] = {GL_NAME_LENGTH, GL_TYPE,
  GL_LOCATION, GL_BLOCK_INDEX};

printf("Active uniforms:\n");
for( int i = 0; i < numUniforms; ++i ) {
  GLint results[4];
  glGetProgramResourceiv(handle, GL_UNIFORM, i, 4,
      properties, 4, NULL, results);
  if( results[3] != -1 )
    continue; // Пропустить uniform-переменные в блоках
  GLint nameBufSize = results[0] + 1;
  char * name = new char[nameBufSize];
  glGetProgramResourceName(handle, GL_UNIFORM, i,
      nameBufSize, NULL, name);
  printf("%-5d %s (%s)\n", results[2], name,
      getTypeString(results[1]));
  delete [] name;
}
```

Как это работает...

Процедура очень похожа на процедуру, описанную в рецепте «Получение списка активных атрибутов и их индексов», поэтому я сосредоточусь лишь на основных отличиях.

Первое отличие, которое сразу бросается в глаза, – здесь функциям `glGetProgramResourceiv` и `glGetProgramInterfaceiv` вместо значения `GL_PROGRAM_INPUT`

передается значение GL_UNIFORM. Во-вторых, запрашивается индекс в блоке (значение GL_BLOCK_INDEX в массиве properties). Дело в том, что некоторые uniform-переменные хранятся в блоках (см. рецепт «Использование uniform-блоков и uniform-буферов»). В данном рецепте предполагается, что нужно получить сведения только об uniform-переменных, находящихся за пределами таких блоков. Переменные за пределами блоков имеют индекс в блоке, равный -1, поэтому программа пропускает все переменные, имеющие индекс в блоке, отличный от -1.

Здесь снова используется функция getTypeString, преобразующая значение типа в строку с его названием (см. пример кода).

Ниже показан вывод описанного фрагмента, когда он выполняется в программе с шейдерами из предыдущего рецепта:

```
Active uniforms:
0    RotationMatrix (mat4)
```

И еще...

Как и переменные-атрибуты, uniform-переменные считаются неактивными, если компоновщик GLSL определит, что они не используются в шейдере.

Предыдущий код допустим только для версии OpenGL 4.3 или выше. В более ранних версиях аналогичный результат можно получить с помощью функций glGetProgramiv, glGetActiveUniform, glGetUniformLocation и glGetActiveUniformName.

См. также

- Рецепт «Передача данных в шейдер с использованием uniform-переменных».

Использование uniform-блоков и uniform-буферов

Если в основной программе предусматривается использование нескольких шейдерных программ с одинаковыми uniform-переменными, в таком случае можно организовать управление этими переменными отдельно для каждой программы. Индексы uniform-переменных генерируются на этапе компоновки программы, поэтому в разных шейдерных программах они могут отличаться, и данные таким uniform-переменным может потребоваться присваивать по разным индексам.

Назначение uniform-блоков – упростить совместное использование uniform-данных разными шейдерными программами. При такой необходимости можно создать буферный объект для хранения значений всех uniform-переменных и связать буфер с uniform-блоком. При смене программы достаточно будет всего лишь повторно связать тот же самый буферный объект с соответствующим блоком в новой программе.

Uniform-блок – это всего лишь группа uniform-переменных, объявленных в пределах синтаксической структуры, известной как uniform-блок. Например, в данном рецепте используется следующий uniform-блок:

```
uniform BlobSettings {
  vec4 InnerColor;
```

```
  vec4 OuterColor;
  float RadiusInner;
  float RadiusOuter;
};
```

Этот фрагмент определяет блок с именем `BlobSettings`, содержащий четыре uniform-переменные. Переменные внутри блока все еще остаются частью глобального пространства имен, и для обращения к ним не требуется предварять их имена именем блока.

Буферные объекты, используемые как хранилища данных для uniform-переменных, часто называют *объектами uniform-буферов*. Далее вы увидите, что объект uniform-буфера – это самый обычный буферный объект, связанный с определенным индексом.

Применение uniform-буферов и uniform-блоков в данном рецепте будет демонстрироваться на простом примере. Мы нарисуем квадрат (из двух треугольников) с координатами текстуры и с помощью фрагментного шейдера нарисуем в его центре круг с размытыми краями, как показано на рис. 1.6.

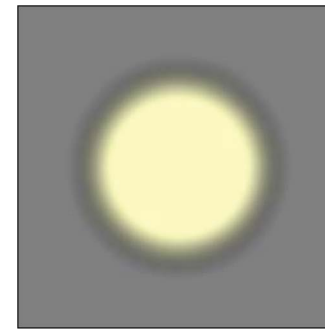

Рис. 1.6 ❖ Круг с размытыми краями в центре квадрата

Подготовка

Для начала нам потребуется программа, рисующая два треугольника, образующих квадрат. Назначьте атрибуту с координатами индекс `0`, а атрибуту с координатами текстуры (изменяющимися в диапазоне от `0` до `1` в каждом направлении) – индекс `1` (см. рецепт «Передача данных в шейдер с использованием переменных-атрибутов и буферов»).

Ниже приводится исходный код вершинного шейдера:

```
#version 430

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexTexCoord;

out vec3 TexCoord;

void main()
{
  TexCoord = VertexTexCoord;
  gl_Position = vec4(VertexPosition,1.0);
}
```

Фрагментный шейдер включает uniform-блок и реализует рисование круга с размытыми краями:

```
#version 430

in vec3 TexCoord;
layout (location = 0) out vec4 FragColor;
```

```
layout (binding = 0) uniform BlobSettings {
  vec4 InnerColor;
  vec4 OuterColor;
  float RadiusInner;
  float RadiusOuter;
};

void main() {
  float dx = TexCoord.x - 0.5;
  float dy = TexCoord.y - 0.5;
  float dist = sqrt(dx * dx + dy * dy);
  FragColor =
    mix( InnerColor, OuterColor,
      smoothstep( RadiusInner, RadiusOuter, dist ));
}
```

Обратите внимание на uniform-блок с именем `BlobSettings`. Переменные внутри этого блока определяют параметры круга. Переменная `OuterColor` хранит значение цвета за пределами круга. Переменная `InnerColor` – цвет в центре круга. Переменная `RadiusInner` определяет радиус внутренней части круга, которая будет заливаться сплошным цветом, и расстояние от центра круга до границы, откуда начнется градиентная заливка. Переменная `RadiusOuter` хранит радиус внешней границы круга, где цвет размытого края совпадает с цветом фона (то есть с цветом `OuterColor`).

Код в функции `main` вычисляет расстояние от координаты текстуры до центра квадрата в точке (0.5, 0.5). Затем это расстояние используется для вычисления цвета с помощью функции `smoothstep`. Эта функция возвращает значение, плавно изменяющееся от 0.0 до 1.0, когда значение третьего аргумента находится в диапазоне, определяемом первыми двумя аргументами. В противном случае она возвращает 0.0 или 1.0 в зависимости от того, является ли `dist` меньше первого аргумента или больше второго аргумента соответственно. Затем используется функция `mix`, осуществляющая линейную интерполяцию между `InnerColor` и `OuterColor` на основе значения, возвращаемого функцией `smoothstep`.

Как это делается...

В основной программе, после компоновки шейдерной программы, реализуйте следующие шаги, чтобы обеспечить передачу данных в uniform-блок внутри фрагментного шейдера:

1. Получить индекс uniform-блока вызовом `glGetUniformBlockIndex`.

```
GLuint blockIndex = glGetUniformBlockIndex(programHandle,
  "BlobSettings");
```

2. Выделить в памяти буфер, который должен хранить данные для uniform-блока. Получить размер буфера можно с вызовом `glGetActiveUniformBlockiv`:

```
GLint blockSize;
glGetActiveUniformBlockiv(programHandle, blockIndex,
       GL_UNIFORM_BLOCK_DATA_SIZE, &blockSize);

GLubyte * blockBuffer;
blockBuffer = (GLubyte *) malloc(blockSize);
```

3. Получить смещение каждой переменной в блоке. Для этого нужно сначала узнать индексы всех переменных в блоке:

```
const GLchar *names[] = { "InnerColor", "OuterColor",
      "RadiusInner", "RadiusOuter" };
GLuint indices[4];
glGetUniformIndices(programHandle, 4, names, indices);

GLint offset[4];
glGetActiveUniformsiv(programHandle, 4, indices,
      GL_UNIFORM_OFFSET, offset);
```

4. Записать требуемые данные в буфер по соответствующим смещениям:

```
// Записать данные в буфер по соответствующим смещениям
GLfloat outerColor[] = {0.0f, 0.0f, 0.0f, 0.0f};
GLfloat innerColor[] = {1.0f, 1.0f, 0.75f, 1.0f};
GLfloat innerRadius = 0.25f, outerRadius = 0.45f;

memcpy(blockBuffer + offset[0], innerColor,
      4 * sizeof(GLfloat));
memcpy(blockBuffer + offset[1], outerColor,
      4 * sizeof(GLfloat));
memcpy(blockBuffer + offset[2], &innerRadius,
      sizeof(GLfloat));
memcpy(blockBuffer + offset[3], &outerRadius,
      sizeof(GLfloat));
```

5. Создать буферный объект и скопировать в него данные:

```
GLuint uboHandle;
glGenBuffers( 1, &uboHandle );
glBindBuffer( GL_UNIFORM_BUFFER, uboHandle );
glBufferData( GL_UNIFORM_BUFFER, blockSize, blockBuffer,
      GL_DYNAMIC_DRAW );
```

6. Связать буферный объект с индексом в точке привязки uniform-буферов, который указан в квалификаторе `layout` внутри фрагментного шейдера (`0`):

```
glBindBufferBase(GL_UNIFORM_BUFFER, 0, uboHandle);
```

Как это работает...

Уф-ф! Похоже, что применение uniform-блоков требует немалого труда! Однако истинные их преимущества проявляются особенно ярко, когда один и тот же буферный объект можно использовать со всеми имеющимися шейдерными программами. Рассмотрим каждый шаг отдельно.

Прежде всего необходимо получить индекс uniform-блока с именем `glGetUniformBlockIndex`. Затем, с помощью функции `glGetActiveUniformBlockiv`, определить его размер. После этого выделяется память для временного буфера с именем `blockBuffer`, где будут храниться данные для блока.

Размещение данных внутри блока зависит от реализации, и разные реализации могут использовать разные отступы для выравнивания по границам слов. Поэто-

му, чтобы обеспечить точное соответствие, нужно узнать смещения всех переменных в блоке. Делается это в два этапа. Сначала вызовом `glGetUniformIndices` запрашиваются индексы всех переменных в блоке. Эта функция принимает массив имен переменных (в третьем аргументе) и возвращает массив с индексами переменных (четвертый аргумент). Затем вызовом `glGetActiveUniformsiv` индексы используются для получения смещений. Когда этой функции в четвертом аргументе передается значение `GL_UNIFORM_OFFSET`, она возвращает смещения всех переменных в массиве, на который ссылается указатель в пятом аргументе. Эту функцию можно также использовать для получения размеров и типов переменных, однако в данном случае мы решили этого не делать ради упрощения кода (хотя и в ущерб универсальности).

Следующий шаг – заполнение временного буфера `blockBuffer` данными для uniform-переменных с учетом смещений. Для этого здесь используется стандартная библиотечная функция `memcpy`.

После заполнения временного буфера с учетом всех нюансов можно создать буферный объект и скопировать в него подготовленные данные. Создание буфера выполняется вызовом функции `glGenBuffers`, которая возвращает дескриптор нового объекта. Затем вызовом `glBindBuffer` буферный объект связывается с точкой связывания `GL_UNIFORM_BUFFER`. Выделение памяти и копирование данных происходит в вызове функции `glBufferData`. В данном примере используется подсказка `GL_DYNAMIC_DRAW`, потому что uniform-данные могут изменяться в ходе отображения (разумеется, это целиком и полностью зависит от ситуации).

Наконец, вызовом `glBindBufferBase` буферный объект связывается с uniform-блоком. Эта функция осуществляет связывание с индексом внутри точки привязки буферов. Часто точки привязки называют еще «индексированными целями». Это означает, что цель фактически может быть массивом целей, а функция `glBindBufferBase` выполняет привязку к единственному индексу в массиве. В данном случае буфер привязывается к индексу, указанному в квалификаторе `layout`: `layout (binding = 0)` (см. раздел «Подготовка» в этом рецепте). Эти два индекса должны совпадать.

Многие из вас наверняка обратили внимание, что здесь используются функции `glBindBuffer` и `glBindBufferBase` с аргументом `GL_UNIFORM_BUFFER`. Разве можно использовать те же самые точки привязки в двух разных контекстах? Дело в том, что в разных функциях имя точки `GL_UNIFORM_BUFFER` имеет немного разные значения. Функция `glBindBuffer` выполняет привязку к точке, которую можно использовать для заполнения или изменения буфера, но нельзя использовать в качестве источника данных для шейдера. Функция `glBindBufferBase` выполняет привязку к индексу, который, напротив, может служить непосредственным источником данных для шейдера. Соглашусь, что все это выглядит немного запутанно.

И еще...

Если позднее данные в uniform-блоке потребуется изменить, это можно сделать с помощью функции `glBufferSubData`. Не забудьте при этом сначала связать буфер с точкой привязки `GL_UNIFORM_BUFFER`.

Использование имен экземпляров с uniform-блоками

Uniform-блоки могут иметь имена экземпляров. Например, нашему блоку `BlobSettings` можно было бы присвоить имя экземпляра `Blob`, как показано ниже:

```
uniform BlobSettings {
  vec4 InnerColor;
  vec4 OuterColor;
  float RadiusInner;
  float RadiusOuter;
} Blob;
```

В таком случае переменные, объявленные в блоке, перемещаются в пространство имен, определяемое именем экземпляра. Соответственно, чтобы обратиться к этим переменным, необходимо будет использовать имя экземпляра, например:

```
FragColor =
  mix( Blob.InnerColor, Blob.OuterColor,
    smoothstep( Blob.RadiusInner, Blob.RadiusOuter, dist )
  );
```

Кроме того, чтобы получить индексы переменных в основной программе, необходимо квалифицировать их именем блока (`BlobSettings` в данном примере):

```
const GLchar *names[] = { "BlobSettings.InnerColor",
      "BlobSettings.OuterColor", "BlobSettings. RadiusInner",
      "BlobSettings.RadiusOuter" };
GLuint indices[4];
glGetUniformIndices(programHandle, 4, names, indices);
```

Применение квалификатора к uniform-блокам

Так как размещение данных внутри объекта uniform-буфера зависит от реализации, мы были вынуждены определять смещения переменных во время выполнения. Однако этого можно избежать, потребовав от OpenGL использовать стандартную схему размещения `std140`. Сделать это можно с помощью квалификатора `layout`:

```
layout( std140 ) uniform BlobSettings {
};
```

Подробное описание схемы размещения `std140` можно найти в спецификации OpenGL (доступна на сайте **http://www.opengl.org**).

В числе других возможных схем размещения, поддерживаемых квалификатором `layout` для uniform-блоков, можно назвать `packed` и `shared`. Схема `packed` просто указывает, что реализация свободна в выборе способов оптимизации расходования памяти (исходя из частоты использования переменных и других критериев). При использовании схемы `packed` все так же требуется определять смещения переменных. Схема `shared` гарантирует, что размещение будет оставаться неизменным для разных шейдерных программ, при условии что объявления uniform-блоков в них будут совпадать. То есть если предполагается использовать один и тот же буферный объект для передачи данных в разные программы, схема `shared` будет как нельзя кстати.

Имеются еще две схемы размещения, о которых стоит упомянуть: `row_major` и `column_major`. Они определяют порядок следования данных в переменных матричных типов, внутри uniform-блоков.

В одном квалификаторе `layout` допускается указывать несколько не противоречащих друг другу схем. Например, чтобы определить блок со схемами размещения `row_major` и `shared`, можно использовать следующий синтаксис:

```
layout( row_major, shared ) uniform BlobSettings {
};
```

См. также

- Передача данных в шейдер с использованием uniform-переменных.

Получение отладочных сообщений

До недавнего времени традиционным способом получения отладочной информации был вызов функции `glGetError`. К сожалению, такой способ отладки чрезвычайно утомителен. Функция `glGetError` возвращает код ошибки, если ошибка возникла в некоторый момент после предыдущего вызова этой функции. Это означает, что в процессе поиска ошибки может потребоваться вызывать `glGetError` после каждого обращения к функциям OpenGL или прибегнуть к методике поиска методом половинного деления, когда сначала определяется блок, где возникла ошибка, а затем два вызова `glGetError` сдвигаются все ближе друг к другу, пока не будет обнаружен источник ошибки. Это так утомительно!

К счастью, в версии OpenGL 4.3 появилась поддержка более современной методики отладки. Теперь можно зарегистрировать отладочную функцию обратного вызова, которая автоматически будет вызываться при возникновении ошибки или при появлении информационного сообщения. Более того, мы сами можем посылать собственные сообщения для обработки той же функцией обратного вызова и фильтровать сообщения, используя различные критерии.

Подготовка

Создайте программу с отладочным контекстом OpenGL. Хотя приобретение отладочного контекста в программе не является строго обязательным, без него мы не сможем получать сообщения, являющиеся информационными. Чтобы получить отладочный контекст OpenGL с использованием GLFW, перед созданием окна вызовите функцию

```
glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, GL_TRUE);
```

В отладочном контексте OpenGL по умолчанию разрешено получение отладочных сообщений. Однако если потребуется включить отладочные сообщения явно, вызовите следующую функцию:

```
glEnable(GL_DEBUG_OUTPUT);
```

Как это делается...

Выполните следующие шаги:

1. Напишите функцию обратного вызова для приема отладочных сообщений. Функция должна соответствовать прототипу, описанному в документации OpenGL. В данном примере будет использоваться следующая функция:

```
void debugCallback(GLenum source, GLenum type, GLuint id,
      GLenum severity, GLsizei length,
      const GLchar * message, void * param) {

    // Преобразовать параметры типа GLenum в строки

  printf("%s:%s[%s](%d): %s\n", sourceStr, typeStr,
      severityStr, id, message);
}
```

2. Зарегистрировать функцию в OpenGL с помощью `glDebugMessageCallback`:

```
glDebugMessageCallback( debugCallback, NULL );
```

3. Включить все сообщения, все источники, все уровни и все идентификационные номера:

```
glDebugMessageControl(GL_DONT_CARE, GL_DONT_CARE,
      GL_DONT_CARE, 0, NULL, GL_TRUE);
```

Как это работает...

Функция `debugCallback` имеет несколько параметров, наиболее важным из которых является само отладочное сообщение (шестой параметр `message`). В данном примере оно просто выводится в `stdout`, но с тем же успехом его можно было бы записывать в файл журнала или отправлять в какое-то другое место.

Первые четыре параметра описывают источник (`source`), тип (`type`), идентификационный номер (`id`) и серьезность сообщения (`severity`). Идентификационный номер – это целое число без знака, определяющее сообщение. Возможные значения параметров, определяющих источник, тип и важность, описываются в следующих ниже таблицах.

Параметр `source` может иметь любое из следующих значений:

Источник	Генерируется
GL_DEBUG_SOURCE_API	Функциями OpenGL API
GL_DEBUG_SOURCE_WINDOW_SYSTEM	Функциями оконной системы
GL_DEBUG_SOURCE_THIRD_PARTY	Прикладными функциями OpenGL
GL_DEBUG_SOURCE_APPLICATION	Самим приложением
GL_DEBUG_SOURCE_OTHER	Другими источниками

Параметр type может иметь любое из следующих значений:

Тип	Описание
GL_DEBUG_TYPE_ERROR	Ошибка в OpenGL API
GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR	Вызов устаревшей функции
GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR	Неопределенная ошибка
GL_DEBUG_TYPE_PORTABILITIY	Вызов непереносимой функции
GL_DEBUG_TYPE_PERFORMANCE	Возможная проблема производительности
GL_DEBUG_TYPE_MARKER	Аннотация
GL_DEBUG_TYPE_PUSH_GROUP	Сообщения, связанные с вталкиванием отладочной группы в стек
GL_DEBUG_TYPE_POP_GROUP	Сообщения, связанные с выталкиванием отладочной группы со стека
GL_DEBUG_TYPE_OTHER	Другие сообщения

Параметр severity может иметь любое из следующих значений:

Тип	Описание
GL_DEBUG_SEVERITY_HIGH	Ошибка или опасная ситуация
GL_DEBUG_SEVERITY_MEDIUM	Важные предупреждения, связанные с производительностью, другие предупреждения или вызов устаревшей функции
GL_DEBUG_SEVERITY_LOW	Избыточное изменение состояния, неважные неопределенные ситуации
GL_DEBUG_SEVERITY_NOTIFICATION	Извещение – не ошибка и не предупреждение, касающееся производительности

В параметре length передается длина строки с сообщением, за исключением завершающего нулевого символа. Последний параметр param – это указатель, определяемый пользователем. С его помощью можно передать функции собственный объект. Например, если предполагается запись сообщения в файл, посредством этого указателя можно передать объект с информацией о доступных средствах ввода/вывода. Осуществить передачу указателя можно с помощью второго параметра функции glDebugMessageCallback (подробнее о ней рассказывается ниже).

Внутри debugCallback каждый параметр типа GLenum преобразуется в строку. Ради экономии места в книге я опустил соответствующий программный код, но вы можете увидеть его в загружаемых примерах для этой книги. Затем информация выводится в stdout.

Вызов glDebugMessageCallback регистрирует функцию обратного вызова в системе отладки OpenGL. В первом параметре ей передается указатель на функцию обратного вызова, а во втором (NULL в данном примере) – указатель на любой объект, который вы захотите передавать в функцию обратного вызова. Этот указатель будет передаваться как последний параметр в каждый вызов debugCallback.

Наконец, вызовом функции glDebugMessageControl определяются фильтры сообщений. Эту функцию можно использовать для выборочного включения или выключения фильтрации по любым комбинациям источников сообщений, их типов, идентификационных номеров или серьезности. В данном примере разрешается получение всех отладочных сообщений.

И еще...

В OpenGL имеется также поддержка стеков именованных отладочных групп. По сути, эта поддержка дает возможность запоминать все настройки фильтров в стеке и возвращаться к ним позднее, после каких-либо изменений. Это может пригодиться, например, если имеются разделы кода, при выполнении которых требуется фильтровать некоторые разновидности сообщений, и есть другие разделы кода, где требуются иные настройки фильтрации.

Для работы с этим стеком используются функции glPushDebugGroup и glPopDebugGroup. Вызов функции glPushDebugGroup генерирует отладочное сообщение с типом GL_DEBUG_TYPE_PUSH_GROUP и вталкивает на стек текущие настройки фильтров. После этого можно изменить настройки фильтрации с помощью glDebugMessageControl и позднее вернуться к первоначальным настройкам вызовом glPopDebugGroup. Функция glPopDebugGroup также генерирует отладочное сообщение, но уже с типом GL_DEBUG_TYPE_POP_GROUP.

Создание класса C++, представляющего шейдерную программу

Читателям, имеющим опыт программирования на языке C++, может показаться привлекательной идея написать классы, инкапсулирующие некоторые объекты OpenGL. Ярким примером мог бы служить класс, представляющий объект шейдерной программы. В этом рецепте мы рассмотрим конструкцию класса на языке C++, который можно использовать для управления объектом шейдерной программы.

Подготовка

В этом рецепте почти не требуется подготовительных операций – достаточно настроить окружение сборки, поддерживающее язык C++. Кроме того, я буду предполагать, что для операций с матрицами и векторами используется библиотека GLM.

Как это делается...

Прежде всего определим собственный класс исключения для ошибок, которые могут возникать в процессе компиляции или компоновки:

```
class GLSLProgramException : public std::runtime_error {
public:
```

```
  GLSLProgramException( const string & msg ) :
      std::runtime_error(msg) { }
};
```

Определим также перечисление для различных типов шейдеров:

```
namespace GLSLShader {
  enum GLSLShaderType {
        VERTEX          = GL_VERTEX_SHADER,
        FRAGMENT        = GL_FRAGMENT_SHADER,
        GEOMETRY        = GL_GEOMETRY_SHADER,
        TESS_CONTROL    = GL_TESS_CONTROL_SHADER,
        TESS_EVALUATION = GL_TESS_EVALUATION_SHADER,
        COMPUTE         = GL_COMPUTE_SHADER
  };
};
```

Класс самой программы имеет следующий интерфейс:

```
class GLSLProgram
{
private:
  int handle;
  bool linked;
  std::map<string, int> uniformLocations;

  int getUniformLocation(const char * name );

  // Несколько вспомогательных функций

public:
  GLSLProgram();
  ~GLSLProgram();

  void compileShader( const char * filename )
      throw(GLSLProgramException);
  void compileShader( const char * filename,
      GLSLShader::GLSLShaderType type )
      throw(GLSLProgramException);
  void compileShader( const string & source,
      GLSLShader::GLSLShaderType type,
      const char * filename = NULL )
      throw(GLSLProgramException);
  void link()      throw(GLSLProgramException);
  void use()       throw(GLSLProgramException);
  void validate()  throw(GLSLProgramException);

  int getHandle();
  bool isLinked();

  void bindAttribLocation( GLuint location,
      const char * name);
  void bindFragDataLocation( GLuint location,
      const char * name );
  void setUniform(const char *name, float x, float y,
```

```
      float z);
  void setUniform(const char *name, const vec3 & v);
  void setUniform(const char *name, const vec4 & v);
  void setUniform(const char *name, const mat4 & m);
  void setUniform(const char *name, const mat3 & m);
  void setUniform(const char *name, float val );
  void setUniform(const char *name, int val );
  void setUniform(const char *name, bool val );

  void printActiveUniforms();
  void printActiveAttribs();
  void printActiveUniformBlocks();
};
```

Загружаемые примеры программного кода. Вы можете загрузить файлы со всеми примерами программного кода для любой книги издательства Packt, приобретенной с использованием вашей учетной записи на сайте **http://www.packtpub.com**. Если вы приобрели эту книгу каким-то иным способом, посетите страницу **http://www.packtpub.com/support** и зарегистрируйтесь, чтобы получить файлы непосредственно на электронную почту.

Полный комплект файлов с исходным кодом примеров для всех рецептов в этой книге также доступен на сайте GitHub: **https://github.com/daw42/glslcookbook**.

Приемы, использованные при реализации данных функций, подробно описываются в предыдущих рецептах в этой главе. Ради экономии места в книге я не включил их программный код (вы найдете его в репозитории GitHub), но мы обсудим некоторые конструктивные решения в следующем разделе.

Как это работает...

Объект класса `GLSLProgram` хранит в себе следующую информацию: дескриптор объекта шейдерной программы (`handle`); логический флаг, сообщающий об успешном завершении компоновки программы (`linked`); и ассоциативный массив (`map`) с индексами uniform-переменных (`uniformLocations`).

Перегруженный метод `compileShader` возбуждает исключение `GLSLProgramException`, если компиляция завершается неудачей. Первая версия этого метода определяет тип шейдера по расширению имени файла. Вторая версия ожидает получить тип шейдера от вызывающей программы. И третья версия используется для компиляции шейдера, исходный код которого передается методу в виде строки. В третьем аргументе может быть передано имя файла, если строка была сформирована на основе файла, что позволит конструировать более полезные сообщения об ошибках.

Сообщение, возвращаемое с исключением `GLSLProgramException`, будет содержать текст из журнала шейдера или шейдерной программы.

Приватный метод `getUniformLocation` используется семейством методов `setUniform` для поиска индексов uniform-переменных. Он сначала проверяет ассоциативный массив `uniformLocations`, и если искомая переменная не найдена, запрашивает ее индекс в OpenGL, сохраняя результат в ассоциативном массиве, прежде чем вернуть его. Метод `fileExists` используется методом `compileShaderFromFile` для проверки наличия файла.

Конструктор просто инициализирует поле linked значением false и поле handle нулевым значением. Значение дескриптора будет записано в поле handle вызовом функции glCreateProgram при попытке скомпилировать первый шейдер.

Метод link пытается скомпоновать программу вызовом glLinkProgram. Затем он проверяет успешность компоновки. В отсутствие ошибок присваивает полю linked значение true и возвращает true. В противном случае он извлекает сообщение из журнала программы (вызовом glGetProgramInfoLog), сохраняет его в экземпляре исключения GLSLProgramException и возбуждает его.

Метод use вызывает функцию glUseProgram, если программа уже была скомпонована, в противном случае ничего не делает.

Методы getHandle и isLinked – это обычные методы чтения, возвращающие дескриптор объекта программы и значение поля linked.

Методы bindAttribLocation и bindFragDataLocation служат обертками вокруг glBindAttribLocation и glBindFragDataLocation. Обратите внимание, что эти методы должны вызываться только перед компоновкой программы.

Перегруженные методы setUniform также являются всего лишь обертками вокруг соответствующих функций glUniform. Каждый из них вызывает getUniformLocation, чтобы получить индекс переменной перед вызовом функции glUniform.

Наконец, методы printActiveUniforms, printActiveUniformBlocks и printActiveAttribs используются для отладки. Они просто выводят список активных uniform-переменных и переменных-атрибутов в stdout.

Ниже приводится простой пример использования класса GLSLProgram:

```
GLSLProgram prog;

try {
  prog.compileShader("myshader.vert");
  prog.compileShader("myshader.frag");
  prog.link();
  prog.validate();
  prog.use();
} catch( GLSLProgramException &e ) {
  cerr << e.what() << endl;
  exit(EXIT_FAILURE);
}

prog.printActiveUniforms();
prog.printActiveAttribs();

prog.setUniform("ModelViewMatrix", matrix);
prog.setUniform("LightPosition", 1.0f, 1.0f, 1.0f);
```

См. также

Полный исходный код, который можно получить в репозитории этой книги на сайте GitHub: **http://github.com/daw42/glslcookbook**.

Все рецепты в этой главе!

Глава 2

Основы шейдеров GLSL

В этой главе описываются следующие рецепты:

- рассеянное отражение с единственным точечным источником света;
- фоновый, рассеянный и отраженный свет;
- использование функций в шейдерах;
- реализация двустороннего отображения;
- реализация модели плоского затенения;
- использование подпрограмм для выбора функциональности в шейдере;
- отбрасывание фрагментов для получения эффекта решетчатой поверхности.

Введение

Поддержка шейдеров впервые была добавлена в OpenGL в версии 2.0, обеспечив возможность программного управления прежде фиксированным конвейером OpenGL. Шейдеры позволяют реализовать альтернативные алгоритмы отображения, обеспечивая высокую гибкость. С помощью шейдеров можно выполнять собственный код прямо на графическом процессоре и тем самым использовать непревзойденные возможности параллелизма, которыми обладают современные GPU.

Шейдеры реализуются с использованием **языка шейдеров (OpenGL Shading Language, GLSL)**. Синтаксис языка GLSL очень похож на синтаксис C, что должно упростить его освоение программистами с опытом использования OpenGL. Учитывая общую направленность данной книги, я не буду давать подробное введение в язык GLSL. Если прежде вам не приходилось использовать язык GLSL, чтение рецептов поможет вам освоить его на примерах. Если вы хорошо знаете GLSL, но не имеете опыта работы с версией 4.x, вы увидите, как реализовать различные приемы с использованием новейшего API. Но, прежде чем перейти к программированию на GLSL, я предлагаю посмотреть, какое место в конвейере OpenGL занимают вершинные и фрагментные шейдеры.

Вершинные и фрагментные шейдеры

В OpenGL 4.3 поддерживаются шесть типов шейдеров: вершинные, геометрические, управления тесселяцией, вычисления тесселяции, фрагментные и вычислительные. В этой главе рассматриваются только вершинные и фрагментные шейдеры. В главе 6 «Использование геометрических шейдеров и шейдеров тесселяции» я представлю несколько рецептов для работы с геометрическими шейдерами и

шейдерами тесселяции, а в главе 10 «Вычислительные шейдеры» расскажу о вычислительных шейдерах.

Шейдеры являются заменяемыми элементами конвейера OpenGL. Если говорить точнее, они делают эти элементы программируемыми. На рис. 2.1 изображена упрощенная схема конвейера OpenGL, состоящего только из вершинного и фрагментного шейдеров.

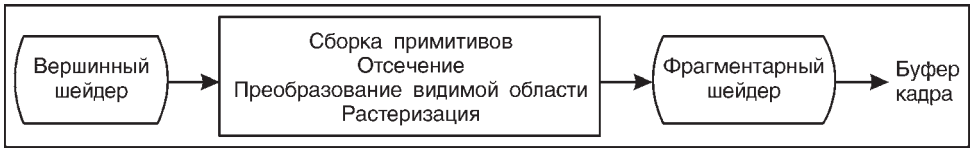

Рис. 2.1 ❖ Упрощенная схема конвейера OpenGL

Информация о вершинах передается вниз по конвейеру и попадает в вершинный шейдер через его входные переменные. Входные переменные вершинного шейдера соответствуют атрибутам вершин (см. рецепт «Передача данных в шейдер с использованием переменных-атрибутов и буферов» в главе 1 «Введение в GLSL»). В общем случае шейдеры получают исходные данные через входные переменные, определяемые программно, а данные в эти переменные записываются основной программой OpenGL или предыдущими шейдерами. Например, входные переменные фрагментного шейдера могут получать данные из выходных переменных вершинного шейдера. Данные могут также передаваться любым шейдерам через uniform-переменные (см. рецепт «Передача данных в шейдер с использованием uniform-переменных» в главе 1 «Введение в GLSL»). Эти переменные применяются для передачи информации, которая изменяется гораздо реже, чем атрибуты вершин (например, для передачи матриц, координат источника света и других параметров сцены). На рис. 2.2 представлена диаграмма, изображающая отношения между входными и выходными переменными при наличии двух активных шейдеров (вершинного и фрагментного).

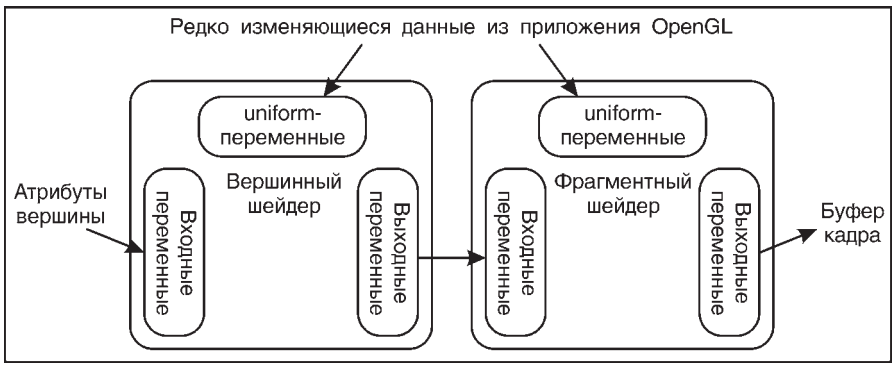

Рис. 2.2 ❖ Отношения между входными и выходными переменными при наличии двух активных шейдеров

Вершинный шейдер выполняется один раз для каждой вершины, причем, как правило, вершины обрабатываются параллельно. Координаты, соответствующие вершине, должны быть преобразованы в усеченные координаты (clip coordinates) и записаны в выходную переменную `gl_Position` до того, как вершинный шейдер закончит выполнение. Вершинный шейдер может также передавать вниз по конвейеру (через выходные переменные) другую информацию. Например, в вершинном шейдере может вычисляться цвет вершины. Этот цвет можно передать дальше через соответствующую выходную переменную.

Между вершинным и фрагментным шейдером выполняются сборка вершин в примитивы, усечение видимой области и ее преобразование (наряду с другими операциями). После этого выполняются процедура растеризации и заливка полигона (если необходимо). Фрагментный шейдер выполняется один раз для каждого фрагмента (пикселя) отображаемого полигона (как правило, фрагменты обрабатываются параллельно). Данные, переданные из вершинного шейдера (по умолчанию), интерполируются и передаются фрагментному шейдеру через его входные переменные. Фрагментный шейдер определяет цвет пикселя и передает его в буфер кадра (frame buffer) через выходные переменные. Информация о глубине обрабатывается автоматически.

Имитация старой, фиксированной функциональности

Программируемые шейдеры дают непревзойденную гибкость и широту возможностей. Однако иногда бывает желательно вернуться к прежней, фиксированной функциональности, используемой по умолчанию, или использовать ее как основу для применения других приемов отображения. Изучение простейшего алгоритма отображения в манере старого конвейера с фиксированной функциональностью также может служить отличной отправной точкой для освоения науки программирования шейдеров.

В этой главе мы рассмотрим основные приемы реализации отображения в манере старого конвейера с фиксированной функциональностью. Мы познакомимся со стандартным алгоритмом вычисления освещенности в заданной точке, реализацией двустороннего отображения (two-sided rendering) и приемом плоского затенения. Попутно мы также увидим несколько примеров использования других особенностей GLSL, таких как функции, подпрограммы и ключевое слово `discard`.

Алгоритмы, представленные в этой главе, далеки от оптимальных. Сделано это преднамеренно, чтобы не ввергать в прострацию тех, кто только начинает изучать описываемые приемы. Некоторые приемы оптимизации будут описываться в конце отдельных рецептов, и еще больше приемов оптимизации вы найдете в следующей главе.

Рассеянное отражение с единственным точечным источником света

Один из простейших приемов затенения заключается в использовании предположения, что поверхность дает только рассеянное отражение. То есть поверхность

объекта выглядит так, будто она рассеивает падающий на нее свет во всех направлениях равномерно. Падающий на поверхность свет частично поглощается поверхностью, прежде чем отразиться от нее, то есть волны одной длины полностью или частично поглощаются, а волны другой длины отражаются полностью. Типичным примером поверхности, дающей рассеянное отражение, может служить поверхность, окрашенная матовой краской. Она не дает глянцевых бликов.

На рис. 2.3 представлен скриншот с изображением тора, дающим рассеянное отражение.

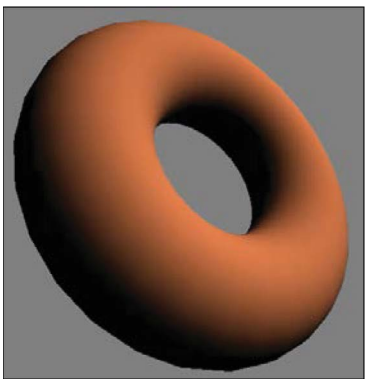

Рис. 2.3 ❖ Изображение тора, дающего рассеянное отражение

В основе математической модели рассеянного отражения лежат два вектора: вектор, направленный от точки поверхности к источнику света (**s**), и вектор нормали в точке на поверхности (**n**). Эти векторы изображены на рис. 2.4.

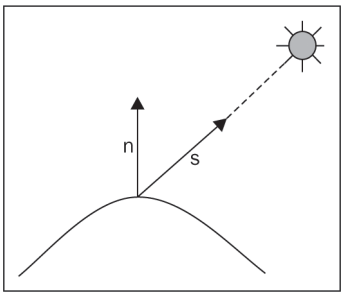

Рис. 2.4 ❖ Два вектора, лежащие в основе математической модели рассеянного отражения

Количество падающего света, достигающего поверхности, отчасти зависит от ориентации поверхности относительно источника света. Физические законы утверждают, что количество света, достигающего точки поверхности, максималь-

но, когда направление на источник из этой точки совпадает с вектором нормали в этой же точке, и равно нулю, когда лучи света распространяются перпендикулярно вектору нормали. В промежуточных положениях оно пропорционально косинусу угла между направлением на источник света и вектором нормали. То есть, так как скалярное произведение (dot product) векторов пропорционально косинусу угла между ними, количество света, падающего на поверхность, можно выразить как произведение интенсивности света на скалярное произведение векторов **s** и **n**:

$$L_d(\mathbf{s} \cdot \mathbf{n}),$$

где L_d – интенсивность света, а векторы **s** и **n** нормализованы.

Скалярное произведение двух векторов единичной длины (то есть нормализованных) равно косинусу угла между ними.

Как отмечалось выше, какая-то часть света может поглощаться поверхностью. Этот эффект можно смоделировать с помощью коэффициента отражения (K_d), представляющего долю падающего света, которая будет отражена. Иногда этот коэффициент называют **коэффициентом рассеивания** или **коэффициентом рассеянного отражения**. В уравнении вычисления количества отраженного света коэффициент рассеянного отражения превращается в масштабирующий множитель:

$$L = K_d L_d(\mathbf{s} \cdot \mathbf{n}).$$

Так как эта модель зависит только от направления на источник света и нормали к поверхности, но не от местоположения наблюдателя, мы получаем модель равномерного (во все стороны) рассеивания.

В данном рецепте это уравнение будет решаться в вершинном шейдере для каждой вершины, а полученный цвет – интерполироваться по всей поверхности.

В этом и в следующем рецептах интенсивность света и коэффициент рассеивания материала поверхности представлены трехкомпонентными (RGB) векторами. Соответственно, уравнение следует интерпретировать как операцию, применяемую к каждому из трех компонентов отдельно. К счастью, в языке GLSL выполнение таких операций реализовано достаточно прозрачно, потому что потребность манипулирования переменными-векторами возникает очень часто.

Подготовка

Для начала нам потребуется приложение OpenGL, вершинный шейдер в котором принимает координаты вершин через атрибут с индексом `0` и вектор нормали через атрибут с индексом `1` (см. рецепт «Передача данных в шейдер с использованием переменных-атрибутов и буферов» в главе 1 «Введение в GLSL»). Приложение также должно передавать матрицы преобразований (проекции, вида модели и нормальные матрицы) через uniform-переменные.

Приложение еще должно передавать через uniform-переменные местоположение источника света (в видимых координатах), `Kd` и `Ld`. Обратите внимание, что переменные `Kd` и `Ld` должны иметь тип `vec3`. Тип `vec3` можно использовать для хранения значений цвета в формате RGB, а также координат точек и векторов.

Как это делается...

Для создания пары шейдеров, реализующих модель рассеянного отражения, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 LightIntensity;

uniform vec4 LightPosition;// Позиция источника света в видимых координатах
uniform vec3 Kd;           // Коэффициент рассеивания
uniform vec3 Ld;           // Интенсивность источника света

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;          // ProjectionMatrix * ModelViewMatrix

void main()
{
    // Преобразовать нормаль и позицию в видимые координаты
    vec3 tnorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyeCoords = ModelViewMatrix *
                     vec4(VertexPosition,1.0));
    vec3 s = normalize(vec3(LightPosition - eyeCoords));

    // Решить уравнение рассеянного отражения
    LightIntensity = Ld * Kd * max( dot( s, tnorm ), 0.0 );

    // Преобразовать позицию в усеченные координаты и передать дальше
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить фрагментный шейдер:

```
in vec3 LightIntensity;

layout( location = 0 ) out vec4 FragColor;

void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```

3. Скомпилировать оба шейдера и скомпоновать шейдерную программу. Добавить шейдерную программу в конвейер OpenGL перед отображением. Подробности, касающиеся компиляции, компоновки и добавления в конвейер OpenGL шейдерных программ, см. в главе 1 «Введение в GLSL».

Как это работает...

В этом примере всю работу выполняет вершинный шейдер. Вычисление интенсивности отраженного света выполняется в видимых координатах за счет первого

преобразования вектора нормали с использованием нормальной матрицы и процедуры нормализации, результат которых сохраняется в `tnorm`. Обратите внимание, что нормализация может оказаться ненужной, если векторы нормалей уже нормализованы и нормальная матрица не предполагает никакого масштабирования.

Нормальной обычно называют транспонированную обратную матрицу, полученную на основе подматрицы 3×3 из верхнего левого угла матрицы вида модели. Обращение и транспонирование необходимы потому, что векторы нормалей преобразуются иначе, чем координаты вершин. За более подробными разъяснениями, касающимися нормальных матриц и их использования, обращайтесь к вводным книгам по компьютерной графике (отличным выбором может стать книга «Computer Graphics with OpenGL», написанная Херном (Hearn) и Бейкером (Baker)[1]). Если матрица вида модели не предполагает какого-либо неоднородного масштабирования, тогда для преобразования вектора нормали можно использовать подматрицу 3×3 из верхнего левого угла матрицы вида модели вместо нормальной матрицы. Но если матрица вида модели включает неоднородное масштабирование, вам придется (повторно) нормализовать векторы нормалей после их преобразования.

Следующий шаг, преобразование позиции вершины в видимую систему координат (то есть в систему координат камеры), выполняется с помощью матрицы вида модели. Затем путем вычитания координат вершины из координат источника света вычисляется направление на источник света, и результат сохраняется в `s`.

Далее решается уравнение интенсивности рассеянного отраженного света, описанное выше, и результат сохраняется в выходной переменной `LightIntensity`. Обратите внимание, что здесь используется функция `max`. Когда результат скалярного произведения оказывается меньше нуля, это говорит о том, что угол между вектором нормали и направлением на источник света больше 90°, то есть свет падает на объект с другой стороны. Поскольку такая ситуация физически невозможна (для закрытого объема), в качестве результата используется значение `0.0`. Однако если вы решите, что вам требуется вычислять освещенность с обеих сторон поверхности, тогда в подобных ситуациях придется обращать вектор нормали (см. рецепт «Реализация двустороннего отображения» далее в этой главе).

В заключение выполняется преобразование позиции вершины обратно в усеченную систему координат путем умножения матрицы модели вида проекции на координаты вершины (то есть: матрица_проекции * матрица_вида * матрица_модели * координаты_вершины), с сохранением результата во встроенной выходной переменной `gl_Position`.

```
gl_Position = MVP * vec4(VertexPosition,1.0);
```

Следующая стадия в конвейере OpenGL ожидает получить координаты вершины в усеченной системе координат, через выходную переменную `gl_Position`. Эта переменная не связана напрямую с какой-либо входной переменной во фрагментном шейдере, но используется конвейером OpenGL на этапах сборки примитивов, отсечения и растеризации, следующих за вершинным шейдером. Очень важно всегда сохранять в этой переменной правильное значение.

---

[1] Херн Д., Бейкер М. П. Компьютерная графика и стандарт OpenGL. – Вильямс, 2005, ISBN: 5-8459-0772-1, 0-13-015390-7. – *Прим. перев.*

Так как `LightIntensity` является выходной переменной вершинного шейдера, ее значение интерполируется для остальных точек поверхности и передается фрагментному шейдеру. Фрагментный шейдер, в свою очередь, просто присваивает полученное значение своей выходной переменной.

И еще...

Модель рассеянного отражения света может применяться к весьма ограниченному множеству поверхностей. Ее лучше применять для отображения «матовых» поверхностей. Кроме того, при использовании описанного приема темные области могут выглядеть слишком темными. Фактически такие области могут вообще не излучать свет и быть полностью черными. В реальной жизни на такие области все же падает некоторый свет, отраженный, например, от стен комнаты. В следующих рецептах мы рассмотрим приемы моделирования других типов поверхностей, а также увидим, как немного подсветить темные области.

См. также

- Рецепт «Передача данных в шейдер с использованием переменных-атрибутов и буферов» в главе 1 «Введение в GLSL».
- Рецепт «Компиляция шейдера» в главе 1 «Введение в GLSL».
- Рецепт «Компоновка шейдерной программы» в главе 1 «Введение в GLSL».
- Рецепт «Передача данных в шейдер с использованием uniform-переменных» в главе 1 «Введение в GLSL».

Фоновый, рассеянный и отраженный свет

Конвейер OpenGL с фиксированной функциональностью, реализующий алгоритмы отображения по умолчанию, действует очень похоже на то, что было представлено выше. Он моделирует освещенность поверхностей как комбинацию из трех компонентов: фонового (ambient), рассеянного (diffuse) и отраженного (specular) света. **Фоновый (ambient)** компонент моделирует свет, отраженный многократно и оттого распространяющийся равномерно во всех направлениях. **Рассеянный (diffuse)** компонент, обсуждавшийся в предыдущем рецепте, представляет свет, отражаемый во всех направлениях. **Отраженный (specular), или зеркальный,** компонент моделирует зеркальные блики на поверхности, когда свет отражается преимущественно в каком-то одном направлении. Объединение всех трех компонентов позволяет моделировать более обширное (хотя также ограниченное) множество видов поверхностей. Иногда такую модель освещенности называют моделью освещенности по Фонгу (Phong reflection model), или моделью затенения по Фонгу, по имени ее разработчика Фонга Буи Туонга (Phong Bui Tuong).

На рис. 2.5 изображен пример реализации освещения тора с применением модели фонового, рассеянного и отраженного освещений (Ambient Diffuse Specular, ADS).

Модель ADS реализуется как сумма трех компонентов: фонового, рассеянного и отраженного света. Фоновый компонент представляет свет, испускаемый

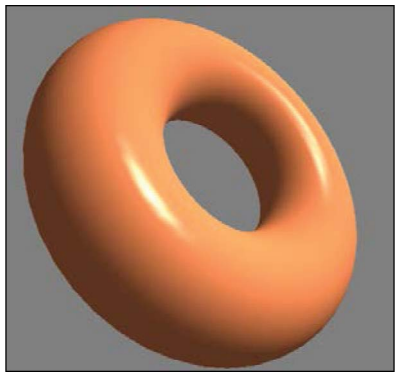

Рис. 2.5 ❖ Реализация освещения тора с применением модели фонового, рассеянного и отраженного света

всеми поверхностями во всех направлениях с равной интенсивностью. Он часто используется для организации подсветки темных областей в сцене. Так как сила фонового света не зависит ни от направления на источник света, ни от направления вектора нормали, его можно смоделировать простым умножением исходной интенсивности этого света (L_a) на коэффициент рассеивания поверхности (K_a):

$I_a = L_a K_a$.

Рассеянный компонент моделирует освещенность шероховатой поверхности, отражающей свет во всех направлениях (см. рецепт «Рассеянное отражение с единственным точечным источником света» выше). Сила отраженного света в этом случае зависит от угла между вектором нормали к поверхности и направлением на источник света:

$I_d = L_d K_d (\mathbf{s} \cdot \mathbf{n})$.

Отраженный (зеркальный) компонент используется для моделирования глянцевых поверхностей. Глянцевые поверхности отражают свет подобно зеркалу. Отраженный свет распространяется преимущественно в одном направлении. Законы физики утверждают, что в этом случае угол падения света равен углу его отражения и что векторы являются компланарными (то есть лежат в одной плоскости) с вектором нормали, как показано на рис. 2.6.

Здесь вектор **r** представляет направление преимущественного распространения отраженного света, вектор **–s** – направление распространения света от источника и вектор **n** – вектор нормали к поверхности. Вычислить вектор **r** можно с помощью следующего уравнения:

$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \mathbf{n})\mathbf{n}$.

Чтобы смоделировать зеркальное отражение, необходимо получить следующие (нормализованные) векторы: вектор направления на источник света (**s**), вектор

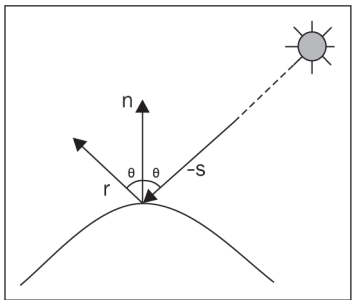

Рис. 2.6 ❖ Модель зеркального отражения

направления преимущественного распространения отраженного света (**r**), вектор направления на наблюдателя (**v**) и вектор нормали к поверхности (**n**). Эти векторы изображены на рис. 2.7.

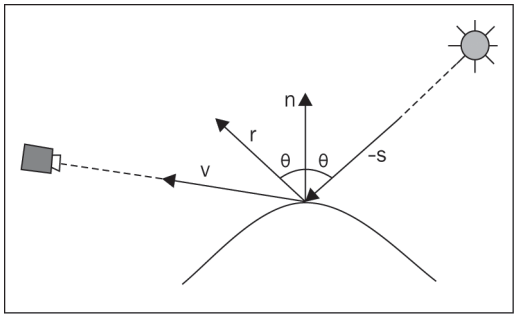

Рис. 2.7 ❖ Векторы, необходимые для моделирования зеркального отражения

Сила отраженного света должна быть максимальной, когда направление отражения совпадает с направлением на наблюдателя, и быстро падать при увеличении угла между векторами **r** и **v**. Реализовать это можно, используя косинус угла между **v** и **r**, возведенный в некоторую степень (f):

$$I_s = L_s K_s (\mathbf{r} \cdot \mathbf{v})^f.$$

(Напомню, что скалярное произведение пропорционально косинусу угла между векторами.) Чем больше степень, тем быстрее результат приближается к нулю с увеличением угла между **v** и **r**. И снова, по аналогии с другими компонентами, мы ввели коэффициент интенсивность зеркального света (L_s) и коэффициент рассеивания (K_s).

Отраженный (зеркальный) компонент создает **блик света** (яркое пятно), типичный для глянцевых поверхностей. Чем больше показатель степени f в уравнении, тем меньше размеры блика и тем ярче он выглядит. Обычно значение для f выбирается в диапазоне от 1 до 200.

Объединив все уравнения, представленные выше, получим следующее уравнение, описывающее освещенность:

$$I = I_a + I_d + I_s = L_a K_a + L_d K_d(\mathbf{s} \cdot \mathbf{n}) + L_s K_s(\mathbf{r} \cdot \mathbf{v})^f.$$

За дополнительной информацией о том, как реализована модель освещения в конвейере с фиксированной функциональностью, обращайтесь к главе 5 «Обработка изображений и приемы работы с экранным пространством».

Это уравнение решается в следующем далее коде, в вершинном шейдере, и затем цвет интерполируется по всему полигону.

Подготовка

Определите в приложении атрибут с индексом `0` для передачи координат вершин и атрибут с индексом `1` для передачи нормалей вершин. Координаты источника света и коэффициенты уравнения будут передаваться из приложения в вершинный шейдер посредством uniform-переменных.

Как это делается...

Для создания пары шейдеров, реализующих модель ADS, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 LightIntensity;

struct LightInfo {
    vec4 Position; // Позиция источника света в видимых координатах
    vec3 La;       // Интенсивность фонового света
    vec3 Ld;       // Интенсивность рассеянного света
    vec3 Ls;       // Интенсивность отраженного света
};

uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;         // Коэффициент отражения фонового света
    vec3 Kd;         // Коэффициент отражения рассеянного света
    vec3 Ks;         // Коэффициент зеркального отражения
    float Shininess; // Показатель степени зеркального отражения
};

uniform MaterialInfo Material;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
```

```
    vec3 tnorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyeCoords = ModelViewMatrix *
                     vec4(VertexPosition,1.0);
    vec3 s = normalize(vec3(Light.Position - eyeCoords));
    vec3 v = normalize(-eyeCoords.xyz);
    vec3 r = reflect( -s, tnorm );
    vec3 ambient = Light.La * Material.Ka;
    float sDotN = max( dot(s,tnorm), 0.0 );
    vec3 diffuse = Light.Ld * Material.Kd * sDotN;
    vec3 spec = vec3(0.0);
    if( sDotN > 0.0 )
        spec = Light.Ls * Material.Ks *
            pow(max( dot(r,v), 0.0 ), Material.Shininess);

    LightIntensity = ambient + diffuse + spec;
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить фрагментный шейдер:

```
in vec3 LightIntensity;

layout( location = 0 ) out vec4 FragColor;

void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```

3. Скомпилировать оба шейдера и скомпоновать шейдерную программу. Добавить шейдерную программу в конвейер OpenGL перед отображением.

Как это работает...

Решение уравнения производится в вершинном шейдере, в видимых координатах. Сначала шейдер преобразует вектор нормали для вершины в систему видимых координат и нормализует его, затем сохраняет результат в `tnorm`. Далее выполняется преобразование позиции вершины в систему видимых координат, и результат сохраняется в `eyeCoords`.

Потом вычисляется нормализованный вектор направления на источник света (`s`). Делается это путем вычитания координат вершины в системе видимых координат из координат источника света и нормализации результата.

Направление на наблюдателя (`v`) определяется как отрицание позиции (нормализованной) вершины, потому что в видимых координатах наблюдатель находится в начале координат.

Направление отражения вычисляется встроенной в GLSL функцией `reflect`, которая «отражает» первый аргумент относительно второго. Результат не требует нормализации, потому что оба вектора, участвующих в вызове, уже нормализованы.

Вычисленный фоновый компонент сохраняется в переменной `ambient`. Далее вычисляется скалярное произведение `s` и `n`. Как и в предыдущем рецепте, здесь используется встроенная функция `max`, чтобы ограничить значения результата

диапазоном от нуля до единицы. Результат сохраняется в переменной `sDotN` и затем используется для вычисления рассеянного компонента. Вычисленное значение рассеянного компонента сохраняется в переменной `diffuse`. Перед вычислением зеркального компонента проверяется значение `sDotN`. Если оно равно нулю, это означает, что свет не попадает на поверхность и нет необходимости вычислять зеркальный компонент, так как его значение все равно получится равным нулю. В противном случае, если `sDotN` больше нуля, зеркальный компонент вычисляется по формуле, представленной выше. И снова здесь используется встроенная функция `max`, чтобы ограничить значения скалярного произведения диапазоном от нуля до единицы. Затем с помощью функции `pow` скалярное произведение возводится в степень `Shininess` (соответствует показателю степени *f* в уравнении освещенности).

Если не проверить значение `sDotN` перед вычислением зеркального компонента, блики могут появиться на поверхностях, повернутых в направлении, противоположном направлению на источник света. Понятно, что в действительности такого быть не может. Некоторые решают эту проблему, умножая зеркальный компонент на компонент рассеянного света, что может привести к значительному уменьшению зеркального компонента и изменению его цвета. Решение, представленное здесь, позволяет избежать такого эффекта за счет использования инструкции ветвления (инструкции `if`). (Инструкции ветвления могут отрицательно сказываться на производительности.)

Сумма трех компонентов затем сохраняется в выходной переменной `LightIntensity`. Это значение будет ассоциировано с вершиной и передано дальше по конвейеру. Прежде чем это значение достигнет фрагментного шейдера, оно подвергнется интерполяции по всей поверхности полигона.

В заключение вершинный шейдер преобразует позицию в усеченные координаты и сохраняет результат во встроенную выходную переменную `gl_Position` (см. рецепт «Рассеянное отражение с единственным точечным источником света» выше).

Фрагментный шейдер, в свою очередь, просто переписывает интерполированное значение `LightIntensity` для текущего фрагмента в свою выходную переменную `FragColor`.

И еще...

Эту версию реализации модели фонового, рассеянного и отраженного света (Ambient Diffuse Specular, ADS) ни в коем случае нельзя считать оптимальной. В нее можно было бы внести несколько улучшений. Например, вычисления вектора зеркального отражения можно избежать, использовав аппроксимацию «вектора полупути» («halfway vector»). Этот прием обсуждается в рецепте «Использование вектора полупути для повышения производительности» в главе 3 «Освещение, затенение и оптимизация».

Аппроксимация «удаленный наблюдатель»

Можно также избежать дополнительной операции нормализации, необходимой для вычисления вектора, направленного на наблюдателя (`v`), с использованием аппроксимации «удаленный наблюдатель» («non-local viewer»). Согласно ей,

вместо вычисления направления на начало координат можно просто использовать постоянный вектор (0, 0, 1) для всех вершин. В результате получится эффект, как если бы наблюдатель находился бесконечно далеко на оси Z. Конечно, точность отображения снизится, но визуально результаты будут очень близки и на глаз почти неотличимы, и это поможет сэкономить на операции нормализации.

В прежнем фиксированном конвейере аппроксимация «удаленный наблюдатель» использовалась по умолчанию, и ее можно было выключить или включить вызовом функции `glLightModel`.

Повершинно или пофрагментно

Поскольку решение уравнения происходит внутри вершинного шейдера, освещенность вычисляется для каждой вершины, или **повершинно**. Один из недостатков такого подхода состоит в том, что зеркальные блики могут оказаться искажены или потеряны из-за того, что вычисления производятся не для каждой точки поверхности. Например, зеркальный блик, который должен быть в середине полигона, может вообще не появиться, потому что вычисления выполняются только для вершин, где зеркальный компонент может оказаться близким к нулю. В рецепте «Пофрагментное вычисление освещенности для повышения реализма» в главе 3 «Освещение, затенение и оптимизация» мы увидим, как перенос вычислений во фрагментный шейдер помогает сделать результаты более реалистичными.

Направленное освещение

Мы можем также избежать необходимости вычислять направление на источник света (`s`) для каждой вершины, если предположить, что сцена освещается направленным светом. Принято считать, что **источник направленного света** находится бесконечно далеко от сцены, в заранее известном направлении. Вместо того чтобы вычислять направление на источник света для каждой вершины, можно использовать постоянный вектор, представляющий направление на удаленный источник света. Мы увидим пример использования такого подхода в рецепте «Освещение источником направленного света» в главе 3 «Освещение, затенение и оптимизация».

Ослабление силы света с расстоянием

Многие могли бы подумать, что в данной модели вычисления освещенности отсутствует один важный компонент – она не учитывает эффекта ослабления силы света с увеличением расстояния от источника. Как мы знаем, сила света уменьшается обратно пропорционально квадрату расстояния от источника. Так почему бы не включить этот компонент в модель?

Как вы понимаете, сделать это не сложно, но дело в том, что визуальный эффект от этого не оправдывает ожиданий. Часто эффект влияния расстояния оказывается преувеличенным, из-за чего получаются изображения, выглядящие нереалистично. Напомню, что наше уравнение является всего лишь аппроксимацией, а не по-настоящему реалистичной моделью, поэтому неудивительно, что добавление фактора, основанного на строгом физическом законе, приводит к ухудшению реалистичности.

В прежнем фиксированном конвейере OpenGL имелась возможность включить эффект ослабления силы света с расстоянием с помощью функции `glLight`. При желании мы тоже могли бы добавить несколько uniform-переменных в наш шейдер, чтобы воспроизвести тот же эффект.

См. также

- Рецепт «Освещение источником направленного света» в главе 3 «Освещение, затенение и оптимизация».
- Рецепт «Пофрагментное вычисление освещенности для повышения реализма» в главе 3 «Освещение, затенение и оптимизация».
- Рецепт «Использование вектора полупути для повышения производительности» в главе 3 «Освещение, затенение и оптимизация».

Использование функций в шейдерах

В языке GLSL поддерживается возможность определения функций, синтаксически похожих на функции в языке C. Однако соглашения об их вызове немного отличаются. В следующем примере мы перепишем шейдер, реализующий модель ADS, с применением функций, чтобы абстрагировать основные шаги.

Подготовка

Как и в предыдущих примерах, определите в приложении атрибут с индексом `0` для передачи координат вершин и атрибут с индексом `1` для передачи нормалей вершин. Координаты источника света и все коэффициенты уравнения должны передаваться из основного приложения посредством uniform-переменных.

Как это делается...

Для реализации модели ADS с применением функций нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 LightIntensity;

struct LightInfo {
    vec4 Position; // Позиция источника света в видимых координатах
    vec3 La;       // Интенсивность фонового света
    vec3 Ld;       // Интенсивность рассеянного света
    vec3 Ls;       // Интенсивность отраженного света
};
uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;         // Коэффициент отражения фонового света
    vec3 Kd;         // Коэффициент отражения рассеянного света
```

```
    vec3 Ks;          // Коэффициент зеркального отражения
    float Shininess; // Показатель степени зеркального отражения
};
uniform MaterialInfo Material;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void getEyeSpace( out vec3 norm, out vec4 position )
{
    norm = normalize( NormalMatrix * VertexNormal);
    position = ModelViewMatrix * vec4(VertexPosition,1.0);
}

vec3 phongModel( vec4 position, vec3 norm )
{
    vec3 s = normalize(vec3(Light.Position - position));
    vec3 v = normalize(-position.xyz);
    vec3 r = reflect( -s, norm );
    vec3 ambient = Light.La * Material.Ka;
    float sDotN = max( dot(s,norm), 0.0 );
    vec3 diffuse = Light.Ld * Material.Kd * sDotN;
    vec3 spec = vec3(0.0);

    if( sDotN > 0.0 )
        spec = Light.Ls * Material.Ks *
            pow( max( dot(r,v), 0.0 ), Material.Shininess );

    return ambient + diffuse + spec;
}

void main()
{
    vec3 eyeNorm;
    vec4 eyePosition;

    // Преобразовать позицию и нормаль в видимые координаты
    getEyeSpace(eyeNorm, eyePosition);

    // Решить уравнение
    LightIntensity = phongModel( eyePosition, eyeNorm );

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить фрагментный шейдер:

```
in vec3 LightIntensity;

layout( location = 0 ) out vec4 FragColor;

void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```

3. Скомпилировать оба шейдера и скомпоновать шейдерную программу. Добавить шейдерную программу в конвейер OpenGL перед отображением.

Как это работает...

В функциях GLSL используется стратегия вызова «вызов по значению при возврате» («call by value-return»), которую иногда называют «вызов по копированию-восстановлению» («call by copy-restore») или «вызов по значению в результате» («call by value-result»). Параметры функций можно объявлять с квалификаторами `in`, `out` и `inout`. Аргументы, соответствующие входным параметрам (объявленным с квалификатором `in` или `inout`), копируются в параметры-переменные в момент вызова, а выходные параметры (объявленные с квалификатором `out` или `inout`) копируются обратно в соответствующие аргументы перед выходом из функции. Если параметр объявлен без квалификатора, по умолчанию подразумевается квалификатор `in`.

Здесь в вершинном шейдере объявлены две функции. Первая, с именем `getEyeSpace`, преобразует позицию вершины и вектор нормали в систему видимых координат и возвращает их в выходных параметрах. В функции `main` создаются две неинициализированные переменные (`eyeNorm` и `eyePosition`) для сохранения результатов, затем вызывается функция `getEyeSpace` с этими переменными в качестве аргументов. Вызываемая функция сохранит результаты в параметрах-переменных (`norm` и `position`), которые будут скопированы в аргументы (`eyeNorm` и `eyePosition`) перед возвратом из функции.

Вторая функция `phongModel` использует только входные параметры. Она принимает позицию и вектор нормали в системе видимых координат и решает уравнение модели ADS. Результаты, возвращаемые функцией, сохраняются в выходной переменной шейдера `LightIntensity`.

И еще...

Так как выходные параметры не предназначены для чтения, внутри функций их следует использовать только для записи. Значения, возвращаемые при попытке их чтения, не определены.

Присваивание нового значения входному параметру (объявленному с квалификатором `in`) внутри функций допустимо. При этом изменится только копия аргумента внутри функции, сам аргумент сохранит прежнее значение.

Квалификатор const

К исключительно входным параметрам (но не к параметрам с квалификатором `out` или `inout`) можно применять дополнительный квалификатор `const`. Он сделает входной параметр доступным только для чтения, то есть ему нельзя будет присвоить другое значение внутри функции.

Перегрузка функций

В GLSL поддерживается также возможность перегрузки функций путем объявления нескольких одноименных функций с разным числом и/или типами па-

раметров. Как и во многих других языках, две перегруженные функции могут не отличаться только типом возвращаемого значения.

Передача массивов или структур в функции

Следует особо отметить, что массивы и структуры передаются в функции по значению. Если передается большой массив или структура, это может привести к значительному увеличению объемов копирования, что нежелательно. В подобных случаях лучше использовать переменные в глобальной области видимости.

См. также

- Рецепт «Рассеянное отражение с единственным точечным источником света» выше.

Реализация двустороннего отображения

Когда отображаются полностью закрытые объекты, обратные стороны полигонов оказываются скрыты. Однако, если в объекте имеются отверстия, он может оказаться повернутым к наблюдателю так, что ему станут видны обратные стороны стенок. В такой ситуации полигоны могут отображаться неправильно из-за того, что фактический вектор нормали направлен не в ту сторону. Чтобы правильно отобразить обратные стороны, достаточно лишь инвертировать вектор нормали и решить уравнение освещенности для нового, инвертированного вектора нормали.

На рис. 2.8 показан скриншот программы, отображающей чайник со снятой крышкой. Для получения изображения слева использовалась модель освещения ADS. Для изображения справа использовалась та же модель ADS, но дополненная приемом двустороннего отображения, обсуждаемым в данном рецепте.

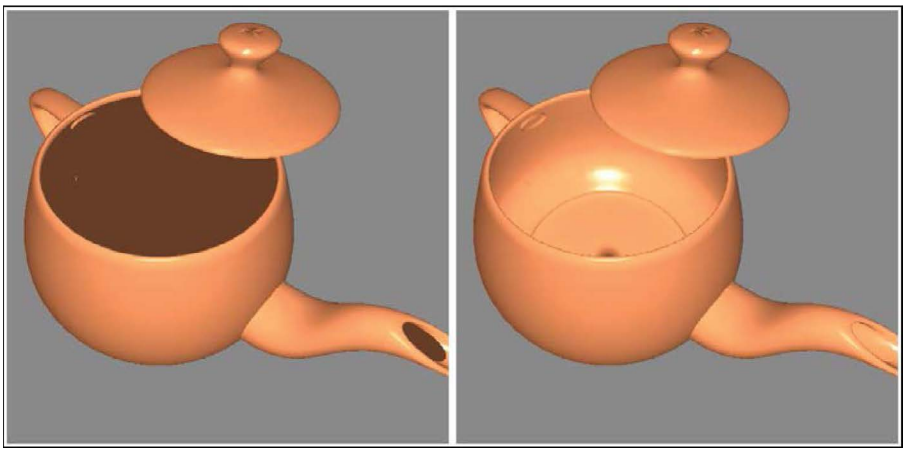

Рис. 2.8 ❖ Слева использована простая модель освещения ADS, справа – дополненная приемом двустороннего отображения

В этом рецепте мы рассмотрим пример использования модели ADS, описанной в предыдущих рецептах, дополненной возможностью правильного отображения обратных поверхностей.

Подготовка

Координаты вершины должны передаваться через атрибут с индексом 0, а вектор нормали – через атрибут с индексом 1. Как и в предыдущих рецептах, параметры источника света должны передаваться в шейдер через uniform-переменные.

Как это делается...

Для реализации пары шейдеров, использующих модель ADS с применением приема двустороннего отображения, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 FrontColor;
out vec3 BackColor;

struct LightInfo {
    vec4 Position; // Позиция источника света в видимых координатах
    vec3 La;       // Интенсивность фонового света
    vec3 Ld;       // Интенсивность рассеянного света
    vec3 Ls;       // Интенсивность отраженного света
};
uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;         // Коэффициент отражения фонового света
    vec3 Kd;         // Коэффициент отражения рассеянного света
    vec3 Ks;         // Коэффициент зеркального отражения
    float Shininess; // Показатель степени зеркального отражения
};
uniform MaterialInfo Material;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

vec3 phongModel( vec4 position, vec3 normal ) {
    // Здесь выполняются вычисления в модели ADS (см. рецепт
    // "фоновый, рассеянный и отраженный свет")
    ...
}

void main()
{
    vec3 tnorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyeCoords = ModelViewMatrix *
```

```
                    vec4(VertexPosition,1.0);
    FrontColor = phongModel( eyeCoords, tnorm );
    BackColor = phongModel( eyeCoords, -tnorm );

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить фрагментный шейдер:

```
in vec3 FrontColor;
in vec3 BackColor;

layout( location = 0 ) out vec4 FragColor;

void main() {
    if( gl_FrontFacing ) {
        FragColor = vec4(FrontColor, 1.0);
    } else {
        FragColor = vec4(BackColor, 1.0);
    }
}
```

3. Скомпилировать оба шейдера и скомпоновать шейдерную программу. Добавить шейдерную программу в конвейер OpenGL перед отображением.

Как это работает...

Вершинный шейдер решает уравнение освещения, используя обе нормали к вершине, прямую и обратную, и передает оба результата во фрагментный шейдер. Фрагментный шейдер выбирает и применяет соответствующий цвет в зависимости от ориентации грани.

В данном примере используется немного измененная версия вершинного шейдера из рецепта «Фоновый, рассеянный и отраженный свет», представленного выше в этой главе. Вычисления по формулам модели помещены в функцию с именем `phongModel`. Эта функция вызывается дважды, первый раз для выполнения вычислений с вектором нормали в прямом направлении (преобразованном в систему видимых координат), второй раз – с инвертированным вектором нормали. Полученные результаты сохраняются в переменных `FrontColor` и `BackColor` соответственно.

Обратите внимание, что в модели имеется ряд аспектов, не зависящих от ориентации вектора нормали (такие как фоновый компонент). Мы могли бы оптимизировать этот код, переписав его так, чтобы вычисление подобных аспектов выполнялось только один раз. В этом рецепте мы выполняем весь комплекс вычислений дважды, чтобы сделать код более простым и читабельным.

Во фрагментном шейдере определяется, какой цвет применить, исходя из значения встроенной переменной `gl_FrontFacing`. Это – логическое значение, определяющее принадлежность отображаемого фрагмента к лицевой или обратной стороне полигона. Обратите внимание, что принадлежность фрагмента к той или иной стороне определяется на основе определения **направления обхода вершин**

полигона (winding of the polygon), а не по вектору нормали. (Считается, что при рассмотрении полигона с лицевой стороны его вершины следуют в порядке обхода против часовой стрелки.) По умолчанию, если вершины отображаются на экране в направлении против часовой стрелки, это указывает, что отображается лицевая сторона полигона, однако такое поведение по умолчанию можно изменить вызовом функции `glFrontFace` из основной программы.

И еще...

В вершинном шейдере мы определяем сторону полигона по направлению вектора нормали, а во фрагментном шейдере сторона полигона определяется по направлению обхода вершин. Чтобы все работало как надо, вектор нормали также должен определяться с учетом значения `glFrontFace`.

Использование двустороннего отображения для отладки

Иногда может быть полезно визуально отделить лицевую и обратную стороны. Например, при работе с произвольными объектами вершины полигона необязательно могут определяться в правильном порядке. Как еще один пример: при последовательной разработке визуальных объектов иногда полезно убедиться, что стороны ориентированы в правильных направлениях. Мы легко можем решить эту задачу, подправив фрагментный шейдер. Например, можно изменить ветку `else` во фрагментном шейдере, как показано ниже:

```
FragColor = mix( vec4(BackColor,1.0),
                 vec4(1.0,0.0,0.0,1.0), 0.7 );
```

В результате все обратные поверхности будут подкрашены красным цветом, что поможет легко отличить их на изображении, как показано на рис. 2.9. На этом скриншоте обратные поверхности подкрашены красным цветом на 70%, как показано в предыдущем фрагменте.

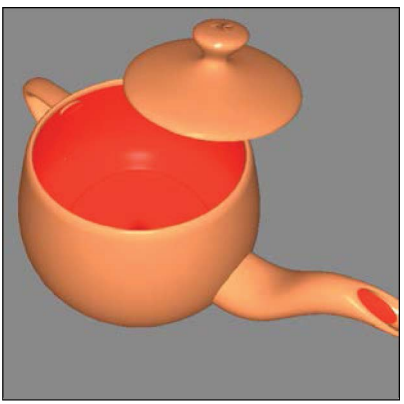

Рис. 2.9 ❖ Обратные поверхности подкрашены красным цветом

См. также

- Рецепт «Рассеянное отражение с единственным точечным источником света» выше.

Реализация модели плоского затенения

Реализация модели повершинной освещенности предполагает вычисление освещенности для каждой вершины и связывание полученного результата (цвета) с этой вершиной. Для получения более гладкого эффекта цвет интерполируется по всей поверхности полигона. Этот прием также называют **затенением по Гуро (Gouraud shading)**. В первых версиях OpenGL такой прием вычисления освещенности для вершин с интерполяцией цвета использовался по умолчанию.

Но иногда желательно, чтобы для всего полигона использовался единый цвет, без интерполяции по поверхности полигона, чтобы обеспечить ему плоский вид. Это может пригодиться в ситуациях, когда сама форма объекта предполагает подобный способ затенения, или чтобы помочь увидеть отдельные полигоны, из которых составлены сложные поверхности. Прием использования единого цвета для всего полигона часто называют приемом **плоского затенения (flat shading)**.

На рис. 2.10 представлен скриншот с изображением, полученным с применением модели ADS вычисления освещенности. Левое изображение получено с применением метода затенения по Гуро, а справа – с применением метода плоского затенения.

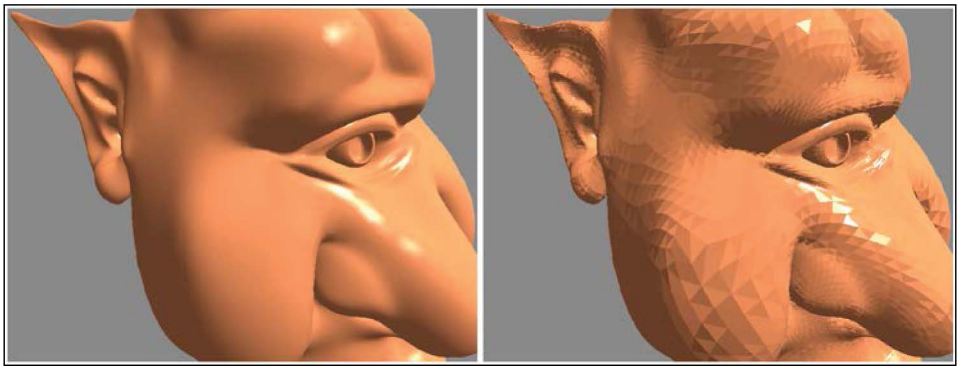

Рис. 2.10 ❖ Слева: метод затенения по Гуро, справа: метод плоского затенения

В первых версиях OpenGL режим плоского затенения можно было включить вызовом функции `glShadeModel` с аргументом `GL_FLAT`. В этом случае в качестве цвета полигона использовался цвет последней его вершины.

В OpenGL 4 необходимость использования модели плоского затенения определяется применением квалификаторов интерполяции к входным/выходным переменным.

Как это делается...

Чтобы реализовать прием плоского затенения в модели ADS вычисления освещенности, нужно выполнить следующие шаги:

1. Оставить вершинный шейдер из прежних примеров реализации модели ADS. Изменить объявление выходной переменной:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

flat out vec3 LightIntensity;

// остальной код остался без изменений...
```

2. Добавить фрагментный шейдер:

```
flat in vec3 LightIntensity;

layout( location = 0 ) out vec4 FragColor;

void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```

3. Скомпилировать оба шейдера и скомпоновать шейдерную программу. Добавить шейдерную программу в конвейер OpenGL перед отображением.

Как это работает...

Режим плоского затенения включается путем добавления квалификатора `flat` в объявление выходной переменной в вершинном шейдере (и соответствующей входной переменной во фрагментном шейдере). Этот квалификатор указывает, что данное значение не подлежит интерполяции перед передачей его фрагментному шейдеру. То есть во фрагментный шейдер попадет значение, соответствующее результату вычислений в вершинном шейдере, либо для первой, либо для последней вершины полигона. Такая вершина называется **преобладающей вершиной (provoking vertex)**, и ее можно назначить с помощью функции `glProvokingVertex`. Например, после вызова

```
glProvokingVertex(GL_FIRST_VERTEX_CONVENTION);
```

в режиме плоского затенения будет использоваться цвет первой вершины. Если передать аргумент `GL_LAST_VERTEX_CONVENTION`, будет использоваться последняя вершина.

См. также

- Рецепт «Рассеянное отражение с единственным точечным источником света» выше.

Использование подпрограмм для выбора функциональности в шейдере

Под подпрограммами в языке GLSL подразумевается механизм связывания вызова функции с одним или несколькими определениями, исходя из значения переменной. Своим действием этот механизм чем-то напоминает указатели на функции в C. Роль указателя, используемого для вызова функции, в данном случае играет uniform-переменная. Значение этой переменной может быть установлено со стороны OpenGL и тем самым связано с одним или несколькими возможными определениями. Определение подпрограммы необязательно должно иметь то же имя, но число и типы параметров, а также тип возвращаемого значения должны совпадать.

То есть подпрограммы дают возможность менять реализацию во время выполнения без замены шейдерной программы и/или перекомпиляции и использования инструкций `if`. Например, в единственном шейдере может быть реализовано несколько алгоритмов вычисления освещенности (затенения) для применения их с разными объектами в рамках одной сцены. Отображая сложные сцены, вместо переключения между шейдерными программами (или использования условных инструкций) можно просто изменять uniform-переменные, соответствующие подпрограммам, и тем самым выбирать алгоритмы отображения, соответствующие конкретным объектам.

Так как производительность является весьма важной для шейдерных программ, возможность уйти от использования условных инструкций или переключения между шейдерными программами может оказаться весьма ценной. С помощью подпрограмм можно реализовать ту же функциональность, но без высоких вычислительных затрат.

В этом рецепте демонстрируется использование подпрограмм посредством двукратного отображения чайника. Первый чайник отображается с использованием полной модели ADS вычисления освещенности, описанной выше, а второй – с использованием модели, реализующей только рассеянное отражение. Выбор между этими двумя способами будет осуществляться с помощью подпрограммы.

На рис. 2.11 представлен скриншот с изображением, созданным с помощью подпрограммы. Чайник слева отображается с использованием полной модели ADS вычисления освещенности, а чайник справа – с использованием модели, реализующей только рассеянное отражение. Переключение между этими двумя способами реализовано с применением подпрограммы.

Подготовка

Как и в предыдущих примерах, определите в приложении атрибут с индексом `0` для передачи координат вершин и атрибут с индексом `1` для передачи нормалей вершин. Координаты источника света и все коэффициенты уравнения, а также все

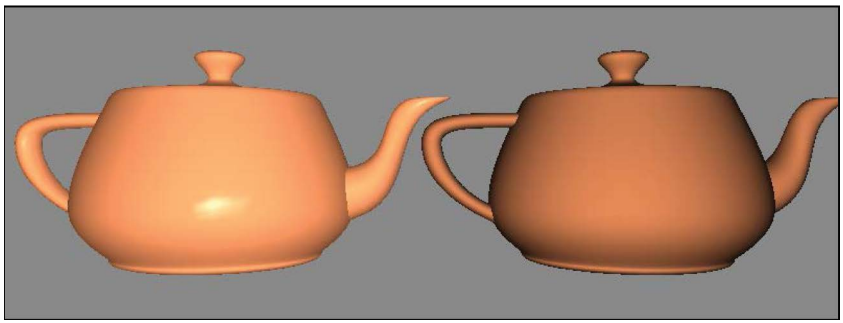

Рис. 2.11 ❖ Изображение слева получено с использованием полной модели ADS вычисления освещенности, а справа – с использованием модели, реализующей только рассеянное отражение

матрицы должны передаваться из основного приложения посредством uniform-переменных.

Далее будет предполагаться, что дескриптор объекта шейдерной программы содержится в переменной `programHandle`.

Как это делается...

Чтобы создать шейдерную программу, функциональность которой переключается с помощью подпрограммы, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
subroutine vec3 shadeModelType( vec4 position, vec3 normal);
subroutine uniform shadeModelType shadeModel;

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 LightIntensity;

struct LightInfo {
    vec4 Position; // Позиция источника света в видимых координатах
    vec3 La;       // Интенсивность фонового света
    vec3 Ld;       // Интенсивность рассеянного света
    vec3 Ls;       // Интенсивность отраженного света
};
uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;         // Коэффициент отражения фонового света
    vec3 Kd;         // Коэффициент отражения рассеянного света
    vec3 Ks;         // Коэффициент зеркального отражения
    float Shininess; // Показатель степени зеркального отражения
};
uniform MaterialInfo Material;
```

```
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void getEyeSpace( out vec3 norm, out vec4 position )
{
    norm = normalize( NormalMatrix * VertexNormal);
    position = ModelViewMatrix * vec4(VertexPosition,1.0);
}

subroutine( shadeModelType )
vec3 phongModel( vec4 position, vec3 norm )
{
    // Здесь выполняются вычисления в модели ADS (см. рецепты
    // "Использование функций в шейдерах" и
    // "Фоновый, рассеянный и отраженный свет")
    ...
}

subroutine( shadeModelType )
vec3 diffuseOnly( vec4 position, vec3 norm )
{
    vec3 s = normalize( vec3(Light.Position - position) );
    return
        Light.Ld * Material.Kd * max( dot(s, norm), 0.0 );
}

void main()
{
    vec3 eyeNorm;
    vec4 eyePosition;

    // Преобразовать позицию и нормаль в видимые координаты
    getEyeSpace(eyeNorm, eyePosition);

    // Решить уравнение освещенности вызовом одной из
    // функций: diffuseOnly или phongModel.
    LightIntensity = shadeModel(eyePosition, eyeNorm);

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить фрагментный шейдер:

```
in vec3 LightIntensity;

layout( location = 0 ) out vec4 FragColor;

void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```

3. Скомпилировать оба шейдера и скомпоновать шейдерную программу. Добавить шейдерную программу в конвейер OpenGL.

4. Использовать следующий код в функции отображения основного приложения:

```
GLuint adsIndex =
    glGetSubroutineIndex(programHandle,
                         GL_VERTEX_SHADER,"phongModel");
GLuint diffuseIndex =
    glGetSubroutineIndex(programHandle,
                         GL_VERTEX_SHADER, "diffuseOnly");

glUniformSubroutinesuiv( GL_VERTEX_SHADER, 1, &adsIndex);
... // Вывести чайник слева

glUniformSubroutinesuiv( GL_VERTEX_SHADER, 1, &diffuseIndex);
... // Вывести чайник справа
```

Как это работает...

В этом примере в вершинном шейдере определена подпрограмма. Определение начинается с объявления типа подпрограммы:

```
subroutine vec3 shadeModelType( vec4 position, vec3 normal);
```

Эта строка объявляет новый тип подпрограммы `shadeModelType`. Синтаксис очень похож на определение прототипа функции: он определяет имя, список параметров и тип возвращаемого значения. Как и в определениях прототипов функций, имена параметров можно опустить.

После определения нового типа подпрограммы объявляется uniform-переменная этого типа с именем `shadeModel`:

```
subroutine uniform shadeModelType shadeModel;
```

Эта переменная играет роль указателя на функцию, и ей во время выполнения будет присваиваться одна из двух возможных функций.

Две функции, являющиеся частью подпрограммы, предваряются квалификатором `subroutine`:

```
subroutine ( shadeModelType )
```

Такое объявление указывает, что функция соответствует типу подпрограммы и ее заголовок должен соответствовать определению типа подпрограммы. Здесь этот префикс используется в определениях функций `phongModel` и `diffuseOnly`. Функция `diffuseOnly` вычисляет освещенность объекта с учетом только рассеянного отражения, а функция `phongModel` вычисляет полную освещенность в модели ADS.

Вызов одной из двух функций подпрограммы выполняется обращением к uniform-переменной `shadeModel` внутри функции `main`:

```
LightIntensity = shadeModel( eyePosition, eyeNorm );
```

Напомню, что в результате будет вызвана одна из двух функций, в зависимости от значения uniform-переменной `shadeModel`, которое устанавливается основной программой.

Внутри функции отображения в основном приложении присваивание значения uniform-переменной, определяющей подпрограмму, выполняется в два этапа. Сначала определяется индекс каждой функции в подпрограмме вызовом `glGetSubroutineIndex`. В первом аргументе этой функции передается дескриптор программы. Во втором – тип шейдера. В данном случае подпрограмма определена в вершинном шейдере, поэтому во втором аргументе передается значение `GL_VERTEX_SHADER`. Третий аргумент – это имя подпрограммы. Программа определяет индексы обеих функций по отдельности и сохраняет их в переменных `adsIndex` и `diffuseIndex`.

Затем выбирается требуемая функция подпрограммы. Для этого нужно присвоить значение uniform-переменной `shadeModel` вызовом `glUniformSubroutinesuiv`. Эта функция способна одновременно присваивать значения сразу нескольким uniform-переменным подпрограмм. В данном случае требуется установить только одну переменную. В первом аргументе передается тип шейдера (`GL_VERTEX_SHADER`), во втором – число uniform-переменных, в третьем – указатель на массив индексов функций подпрограмм. Так как в этой реализации требуется установить единственную uniform-переменную, мы передаем адрес переменной типа `GLuint` с требуемым индексом, а не полноценный массив. Конечно, если бы нам нужно было установить сразу несколько uniform-переменных, мы использовали бы массив. В общем случае в третьем аргументе передается массив с индексами функций, который заполняется так: в *i*-й элемент массива записывается значение для uniform-переменной с индексом *i*. Так как у нас имеется единственная подпрограмма, мы передаем индекс функции для записи в uniform-переменную с индексом `0`.

Возможно, у кого-то из вас родился вопрос: «Откуда известно, что uniform-переменная подпрограммы имеет индекс 0? Мы нигде не определяли ее индекс перед вызовом `glUniformSubroutinesuiv`!» Все просто: мы положились на тот факт, что OpenGL всегда начинает нумерацию подпрограмм с нуля. Если бы у нас имелось несколько подпрограмм, тогда нам пришлось бы определить их индексы с помощью `glGetSubroutineUniformLocation` и собирать массив с индексами в соответствующем порядке.

Функция `glUniformSubroutinesuiv` требует устанавливать сразу все uniform-переменные в одном вызове, чтобы они могли быть проверены средой выполнения OpenGL за один присест.

И еще...

К сожалению, привязка подпрограмм сбрасывается при исключении шейдерной программы из конвейера вызовом `glUseProgram` или каким-то другим способом. Из-за этого требуется вызывать `glUniformSubroutinsuiv` каждый раз, когда программа активируется.

Функция подпрограммы, объявленная в шейдере, может соответствовать более чем одному типу подпрограммы. Квалификатору `subroutine` можно передать список типов подпрограмм, разделенных запятыми. Например, если функция со-

ответствует типам подпрограмм `type1` и `type2`, ее определение может включать следующий квалификатор:

```
subroutine( type1, type2 )
```

Благодаря этому одну и ту же функцию можно использовать в разных подпрограммах.

См. также

- Рецепт «Фоновый, рассеянный и отраженный свет» выше.
- Рецепт «Рассеянное отражение с единственным точечным источником света» выше.

Отбрасывание фрагментов для получения эффекта решетчатой поверхности

Внутри фрагментных шейдеров есть возможность использовать ключевое слово `discard`, чтобы «отбрасывать» ненужные фрагменты. Использование этого ключевого слова вызывает остановку выполнения фрагментного шейдера, без вывода чего бы то ни было в выходной буфер. Таким способом можно создавать «дырки» в полигонах без использования специальных приемов. Фактически, так как фрагмент полностью уничтожается, отсутствует всякая зависимость от того, в каком порядке рисуются объекты, и устраняется необходимость следить за их глубиной, что могло бы потребоваться при использовании приемов наложения.

В этом рецепте мы нарисуем чайник и с помощью ключевого слова `discard` выборочно удалим фрагменты, опираясь на координаты текстуры. Результат показан на рис. 2.12.

Рис. 2.12 ❖ Результат удаления фрагментов с помощью ключевого слова discard

Подготовка

Основное приложение должно передавать в вершинный шейдер позиции вершин, нормали и координаты текстуры. Позиция вершины должна передаваться через атрибут с индексом 0, вектор нормали – через атрибут с индексом 1, координаты текстуры – через атрибут с индексом 2. Как и в предыдущих рецептах, параметры источника света должны передаваться в шейдер через uniform-переменные.

Как это делается...

Чтобы создать шейдерную программу, отбрасывающую фрагменты по квадратной решетке (как показано на рис. 2.12), нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;

out vec3 FrontColor;
out vec3 BackColor;
out vec2 TexCoord;

struct LightInfo {
    vec4 Position; // Позиция источника света в видимых координатах
    vec3 La;       // Интенсивность фонового света
    vec3 Ld;       // Интенсивность рассеянного света
    vec3 Ls;       // Интенсивность отраженного света
};
uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;         // Коэффициент отражения фонового света
    vec3 Kd;         // Коэффициент отражения рассеянного света
    vec3 Ks;         // Коэффициент зеркального отражения
    float Shininess; // Показатель степени зеркального отражения
};
uniform MaterialInfo Material;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void getEyeSpace( out vec3 norm, out vec4 position )
{
    norm = normalize( NormalMatrix * VertexNormal);
    position = ModelViewMatrix * vec4(VertexPosition,1.0);
}

vec3 phongModel( vec4 position, vec3 norm )
{
    // Здесь выполняются вычисления в модели ADS (см. рецепт
    // "Фоновый, рассеянный и отраженный свет")
```

```
    ...
}

void main()
{
    vec3 eyeNorm;
    vec4 eyePosition;

    TexCoord = VertexTexCoord;

    // Преобразовать позицию и нормаль в видимые координаты
    getEyeSpace(eyeNorm, eyePosition);

    FrontColor = phongModel( eyePosition, eyeNorm );
    BackColor = phongModel( eyePosition, -eyeNorm );

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить фрагментный шейдер:

```
in vec3 FrontColor;
in vec3 BackColor;
in vec2 TexCoord;

layout( location = 0 ) out vec4 FragColor;

void main() {
    const float scale = 15.0;

    bvec2 toDiscard = greaterThan( fract(TexCoord * scale),
                                   vec2(0.2,0.2) );

    if( all(toDiscard) )
        discard;

    if( gl_FrontFacing )
        FragColor = vec4(FrontColor, 1.0);
    else
        FragColor = vec4(BackColor, 1.0);
}
```

3. Скомпилировать оба шейдера и скомпоновать шейдерную программу. Добавить шейдерную программу в конвейер OpenGL перед отображением.

Как это работает...

Так как в этом примере будут отбрасываться некоторые фрагменты изображения чайника, через получающиеся отверстия будет видна его внутренняя поверхность. Поэтому нужно предусмотреть вычисление освещенности с обеих сторон – внешней и внутренней. Для этого будет использоваться прием двустороннего отображения, рассматривавшийся выше.

Вершинный шейдер почти без изменений взят из рецепта реализации двустороннего отображения. Основное отличие заключается в появлении дополнитель-

ных координат текстуры. Отличия от предыдущего листинга выделены жирным. Для управления координатами текстуры была добавлена дополнительная входная переменная `VertexTexCoord` как атрибут с индексом `2`. Значение этой входной переменной передается непосредственно во фрагментный шейдер без изменений, через выходную переменную `TexCoord`. Вычисление освещенности по модели ADS выполняется дважды: один раз – для заданного вектора нормали, с сохранением результата в переменной `FrontColor`, и второй раз – для инвертированного вектора нормали, с сохранением результата в `BackColor`.

Внутри фрагментного шейдера необходимость отбрасывания фрагмента определяется с применением простого приема, воспроизводящего эффект решетчатой поверхности, как было показано на рис. 2.12. Сначала координаты текстуры масштабируются умножением на произвольный масштабный коэффициент. Он соответствует числу прямоугольных отверстий в решетке на единицу (после масштабирования) текстуры. Затем из каждого компонента масштабированных координат текстуры извлекается дробная часть с помощью встроенной функции `fract`. Дробные части сравниваются со значением `0.2` с помощью встроенной функции `greaterThan`, и результат сохраняется в векторе `toDiscard` типа `bool`. Функция `greaterThan` сравнивает два вектора поэлементно и сохраняет результаты в соответствующих компонентах возвращаемого вектора типа `bool`.

Если оба компонента вектора `toDiscard` имеют значение `true`, следовательно, проверяемый фрагмент находится внутри квадратного отверстия, и потому его следует отбросить. Для проверки здесь используется встроенная функция `all`. Она возвращает `true`, если все компоненты переданного ей вектора имеют значение `true`. Если функция вернет `true`, вызывается инструкция `discard`.

В противном случае определяется цвет фрагмента, исходя из ориентации полигона, как это делалось в рецепте «Реализация двустороннего отображения» выше.

См. также

- Рецепт «Реализация двустороннего отображения».

Глава 3

Освещение, затенение и оптимизация

В этой главе описываются следующие рецепты:

- освещение несколькими точечными источниками света;
- освещение источником направленного света;
- пофрагментное вычисление освещенности для повышения реализма;
- использование вектора полупути для повышения производительности;
- имитация узконаправленных источников света;
- придание изображению «мультяшного» вида;
- имитация тумана;
- настройка проверки глубины.

Введение

В главе 2 «Основы шейдеров GLSL» было представлено несколько приемов реализации освещенности и затенения, которые прежде входили в состав конвейера с фиксированной функциональностью. Также были продемонстрированы некоторые основные особенности языка GLSL, такие как функции и подпрограммы. Здесь мы выйдем за рамки модели освещенности, представленной в главе 2 «Основы шейдеров GLSL», и посмотрим, как можно воспроизводить такие эффекты освещенности, как блики и туман, и как придавать изображениям «мультяшный» вид. Я расскажу, как использовать несколько источников света и как повысить реализм результатов с применением приема пофрагментного затенения.

В этой главе также будут показаны приемы повышения эффективности вычислений освещенности за счет использования аппроксимации «вектора полупути» (halfway vector) и источников направленного света.

В заключение этой главы я расскажу, как выполнить тонкую настройку механизма определения глубины для оптимизации его работы.

Освещение несколькими точечными источниками света

При наличии в сцене нескольких источников света уравнение освещенности следует вычислить для каждого из них и суммировать результаты для определения

интенсивности света, отраженного данной точкой поверхности. Часто для хранения координат источников света и других их параметров создаются uniform-массивы. В этом рецепте будет использоваться массив структур, чтобы иметь возможность хранить характеристики нескольких источников света в единственной uniform-переменной.

На рис. 3.1 изображена фигура «поросенка», освещаемая пятью источниками света разных цветов. Обратите внимание на множество бликов.

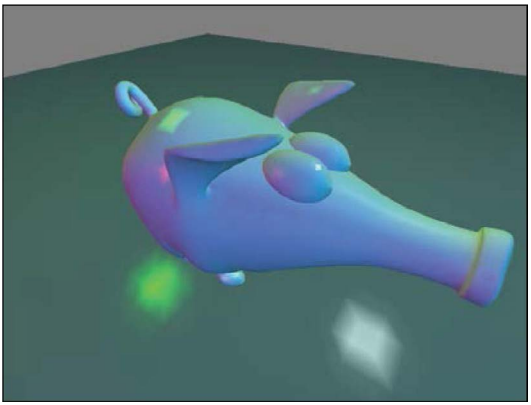

Рис. 3.1 ❖ Фигура «поросенка», освещаемая пятью источниками света разных цветов

Подготовка

Определите в приложении атрибут с индексом 0 для передачи координат вершин и атрибут с индексом 1 для передачи нормалей вершин.

Как это делается...

Для создания шейдерной программы, реализующей модель ADS (затенения по Фонгу) с несколькими источниками света, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 Color;

struct LightInfo {
  vec4 Position;  // Позиция источника в видимых координатах
  vec3 Intensity; // Интенсивность света
};
uniform LightInfo lights[5];

// Характеристики материала
```

```
uniform vec3 Kd;         // Коэффициент отражения рассеянного света
uniform vec3 Ka;         // Коэффициент отражения фонового света
uniform vec3 Ks;         // Коэффициент зеркального отражения
uniform float Shininess; // Показатель степени зеркального отражения

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;

vec3 ads( int lightIndex, vec4 position, vec3 norm )
{
  vec3 s = normalize( vec3(lights[lightIndex].Position -
                                  position) );
  vec3 v = normalize(vec3(-position));
  vec3 r = reflect( -s, norm );
  vec3 I = lights[lightIndex].Intensity;
  return
        I * ( Ka +
        Kd * max( dot(s, norm), 0.0 ) +
        Ks * pow( max( dot(r,v), 0.0 ), Shininess ) );
}

void main()
{
  vec3 eyeNorm = normalize( NormalMatrix * VertexNormal);
  vec4 eyePosition = ModelViewMatrix *
                     vec4(VertexPosition,1.0);

  // Решить уравнение освещенности для каждого источника света
  Color = vec3(0.0);
  for( int i = 0; i < 5; i++ )
    Color += ads( i, eyePosition, eyeNorm );

  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить простой фрагментный шейдер:

```
in vec3 Color;

layout( location = 0 ) out vec4 FragColor;

void main() {
  FragColor = vec4(Color, 1.0);
}
```

3. В основном приложении определить значения для компонентов массива `lights` в вершинном шейдере, как показано ниже. В этом примере используется класс шейдерной программы на C++ (`prog` – это экземпляр класса `GLSLProgram`).

```
prog.setUniform("lights[0].Intensity",
                vec3(0.0f,0.8f,0.8f) );
prog.setUniform("lights[0].Position", position );
```

Не забудьте изменить индекс массива при переходе от одного источника света к другому.

Как это работает...

Внутри вершинного шейдера параметры источников света хранятся в uniform-массиве `lights`. Каждый элемент массива – это структура типа `LightInfo`. В данном примере используются пять источников света. Интенсивность света хранится в поле `Intensity`, а местоположение в системе видимых координат – в поле `Position`.

Остальные uniform-переменные играют ту же роль, что и в реализации модели фонового, рассеянного и отраженного освещений (Ambient Diffuse Specular, ADS), представленной в главе 2 «Основы шейдеров GLSL».

Функция `ads` вычисляет освещенность от заданного источника света. В первом параметре `lightIndex` функции передается индекс источника света. Вычисления выполняются на основе значений в массиве `lights` с этим индексом.

В функции `main` вычисления освещенности от всех источников света реализованы с помощью цикла `for`. Результаты вычислений суммируются в выходной переменной `Color`.

Фрагментный шейдер просто применяет интерполированный цвет к фрагменту.

См. также

- Рецепт «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL».
- Рецепт «Освещение источником направленного света» ниже.

Освещение источником направленного света

Основным компонентом в вычислениях освещенности является вектор, направленный из точки на поверхности в сторону источника света (`s` в предыдущих примерах). Векторы, направленные на очень далекий источник света из разных точек поверхности, мало отличаются друг от друга. Фактически для очень далекого источника света можно считать, что для любой точки поверхности вектор остается одним и тем же. (Иначе говоря, лучи света от такого источника падают на поверхность практически параллельно.) Такая модель с успехом может использоваться для вычисления освещенности от очень далекого и очень мощного источника света, такого как Солнце. Подобные источники света часто называют **источниками направленного света**, потому что они не имеют определенной позиции, только направление.

Конечно, мы игнорируем тот факт, что сила света обратно пропорциональна квадрату расстояния от источника. Однако для направленных источников света это общепринятая практика.

Когда расчеты освещенности производятся для источника направленного света, направление на источник принимается одинаковым для всех точек поверхности. Соответственно, увеличивается скорость вычислений, потому что отпадает необходимость отдельно вычислять направление на источник для каждой вершины.

Разумеется, освещенности от точечного источника света и от источника направленного света выглядят по-разному. На рис. 3.2 изображен тор, освещаемый точечным источником света (слева) и источником направленного света (справа). На изображении слева источник света находится где-то неподалеку от тора. Источник направленного света освещает большую площадь поверхности тора, потому что лучи света от него распространяются параллельно.

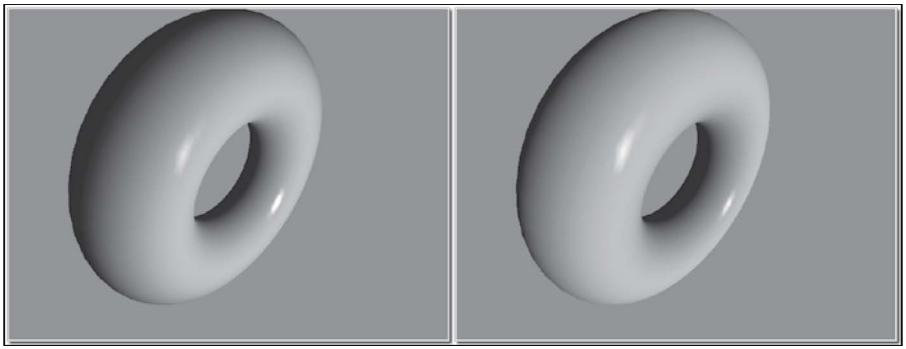

Рис. 3.2 ❖ Тор, освещаемый точечным источником света (слева) и источником направленного света (справа)

В предыдущих версиях OpenGL четвертый компонент координат источника света определял, является ли он источником направленного света. Нулевое значение в четвертом компоненте интерпретировалось как признак источника направленного света, и тогда первые три компонента трактовались как вектор направления на источник. В противном случае первые три компонента интерпретировались как фактические координаты источника света. В данном примере будет эмулироваться точно такая же функциональность.

Подготовка

Определите в приложении атрибут с индексом `0` для передачи координат вершин и атрибут с индексом `1` для передачи нормалей вершин.

Как это делается...

Для создания шейдерной программы, реализующей модель ADS с использованием источника направленного света, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 Color;

uniform vec4 LightPosition;
```

```
uniform vec3 LightIntensity;

uniform vec3 Kd;         // Коэффициент отражения рассеянного света
uniform vec3 Ka;         // Коэффициент отражения фонового света
uniform vec3 Ks;         // Коэффициент зеркального отражения
uniform float Shininess; // Показатель степени зеркального отражения

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

vec3 ads( vec4 position, vec3 norm )
{
  vec3 s;
  if( LightPosition.w == 0.0 )
    s = normalize(vec3(LightPosition));
  else
    s = normalize(vec3(LightPosition - position));

  vec3 v = normalize(vec3(-position));
  vec3 r = reflect( -s, norm );
  return
    LightIntensity * ( Ka +
                       Kd * max( dot(s, norm), 0.0 ) +
                       Ks * pow( max( dot(r,v), 0.0 ),
                         Shininess ) );
}

void main()
{
  vec3 eyeNorm = normalize( NormalMatrix * VertexNormal);
  vec4 eyePosition = ModelViewMatrix *
                     vec4(VertexPosition,1.0);

  // Решить уравнение освещенности
  Color = ads( eyePosition, eyeNorm );
  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить тот же фрагментный шейдер, что и в предыдущем рецепте:

```
in vec3 Color;

layout( location = 0 ) out vec4 FragColor;

void main() {
  FragColor = vec4(Color, 1.0);
}
```

Как это работает...

Внутри вершинного шейдера анализируется четвертая координата в uniform-переменной `LightPosition`, чтобы выяснить, как интерпретировать значение этой переменной – как местоположение точечного источника света или как направле-

ние на источник направленного освещения. Внутри функции ads, ответственной за решение уравнения освещенности, значение вектора s вычисляется, исходя из значения четвертой координаты в LightPosition. Если она равна нулю, LightPosition нормализуется и используется как направление на источник света. В противном случае LightPosition интерпретируется как местоположение точечного источника света, на основе которого вычисляется вектор направления на него, путем вычитания координат вершины из LightPosition с последующей нормализацией результата.

И еще...

Использование источника направленного света несколько ускоряет вычисления из-за отсутствия необходимости вычислять направление на источник света для каждой вершины. Это позволяет сэкономить время на операции вычитания, что не очень много, но это ускорение становится более ощутимым при наличии нескольких источников и когда вычисления освещенности выполняются пофрагментно.

См. также

- Рецепт «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL».
- Рецепт «Пофрагментное вычисление освещенности для повышения реализма» ниже.

Пофрагментное вычисление освещенности для повышения реализма

Когда вычисление освещенности производится внутри вершинного шейдера (как в предыдущем рецепте), мы получаем цвет для каждой вершины. Затем этот цвет интерполируется по поверхности, а фрагментный шейдер присваивает интерполированный цвет полученному фрагменту. Как отмечалось выше (в рецепте «Реализация модели плоского затенения» в главе 2 «Основы шейдеров GLSL»), этот прием, часто называемый **затенением по Гуро**, как и все остальные подобные приемы, является аппроксимацией и не всегда дает желаемые результаты; например, характеристики отражения света в вершинах могут несколько отличаться от тех же характеристик в середине полигона. Например, блик света может приходиться на центр полигона, а не на вершины. В результате упрощенное вычисление освещенности в вершинах может привести к пропаже блика в изображении. Также при затенении по Гуро могут проявляться другие нежелательные артефакты, такие как границы полигонов, из-за того что интерполяция дает менее точные результаты с точки зрения физики.

Чтобы повысить точность результатов, можно перенести вычисления из вершинного во фрагментный шейдер. Вместо цвета в этом случае будут интерполироваться координаты фрагмента и вектор нормали, и на их основе вычисляться

цвет каждого фрагмента. Данный прием часто называют **затенением по Фонгу**, или **интерполяцией по Фонгу**. Результат затенения по Фонгу оказывается более точным и дает более естественный результат, но все же некоторые нежелательные артефакты могут проявляться и в этом случае.

На рис. 3.3 показаны различия между результатами затенения по Гуро и затенения по Фонгу. Слева показано изображение, полученное с использованием затенения (повершинного) по Гуро, а справа – изображение, полученное с использованием затенения (пофрагментного) по Фонгу. Плоскость под чайником, имитирующая поверхность стола, нарисована как единый квадрат. Обратите внимание на различия в бликах на чайнике, а также на различия в освещенности плоскости под чайником.

Рис. 3.3 ❖ Чайник, нарисованный с использованием затенения по Гуро (слева) и с использованием затенения по Фонгу (справа)

В этом рецепте реализовано затенение по Фонгу посредством передачи координат и вектора нормали из вершинного шейдера во фрагментный шейдер и вычисления освещенности в соответствии с моделью ADS внутри фрагментного шейдера.

Подготовка

Определите в приложении атрибут с индексом `0` для передачи координат вершин и атрибут с индексом `1` для передачи нормалей вершин. Основное приложение должно также передавать значения для uniform-переменных `Ka`, `Kd`, `Ks`, `Shininess`, `LightPosition` и `LightIntensity`, из которых первые четыре – это стандартные свойства материала в модели освещенности ADS. Последние две – это местоположение источника света в видимых координатах и интенсивность испускаемого им света соответственно. Наконец, основное приложение должно также передавать значения для uniform-переменных `ModelViewMatrix`, `NormalMatrix`, `ProjectionMatrix` и `MVP`.

Как это делается...

Для создания шейдерной программы, реализующей пофрагментное затенение по Фонгу, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 Position;
out vec3 Normal;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
    Normal = normalize( NormalMatrix * VertexNormal);
    Position = vec3( ModelViewMatrix *
                     vec4(VertexPosition,1.0) );
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить фрагментный шейдер:

```
in vec3 Position;
in vec3 Normal;

uniform vec4 LightPosition;
uniform vec3 LightIntensity;
uniform vec3 Kd;          // Коэффициент отражения рассеянного света
uniform vec3 Ka;          // Коэффициент отражения фонового света
uniform vec3 Ks;          // Коэффициент зеркального отражения
uniform float Shininess; // Показатель степени зеркального отражения

layout( location = 0 ) out vec4 FragColor;

vec3 ads( )
{
  vec3 n = normalize( Normal );
  vec3 s = normalize( vec3(LightPosition) - Position );
  vec3 v = normalize(vec3(-Position));
  vec3 r = reflect( -s, n );
  return
    LightIntensity *
      ( Ka +
        Kd * max( dot(s, n), 0.0 ) +
        Ks * pow( max( dot(r,v), 0.0 ), Shininess ) );
}

void main() {
  FragColor = vec4(ads(), 1.0);
}
```

Как это работает...

Вершинный шейдер имеет две выходные переменные: `Position` и `Normal`. В функции `main` вектор нормали преобразуется в систему видимых координат путем умножения нормальной матрицы на вектор и затем сохраняется в переменной `Normal`. Координаты вершины также преобразуются в систему видимых координат путем умножения матрицы модели вида и сохраняются в переменной `Position`.

Значения `Position` и `Normal` автоматически будут интерполироваться и передаваться фрагментному шейдеру через соответствующие входные переменные. Фрагментный шейдер вычисляет освещенность фрагмента по стандартной формуле ADS, используя полученные значения. Результат сохраняется в выходной переменной `FragColor`.

И еще...

Перенос вычислений во фрагментный шейдер позволяет получить более реалистичное изображение. Однако за это приходится платить снижением производительности, так как теперь вычисления выполняются уже не для вершин, а для каждого пикселя. Но все не так плохо, потому что современные графические карты обладают достаточно большой вычислительной мощностью, чтобы обсчитать все фрагменты полигона параллельно. Фактически производительность вершинных и пофрагментных вычислений оказывается почти равной.

См. также

- Рецепт «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL».

Использование вектора полупути для повышения производительности

Как рассказывалось в рецепте «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL», для вычисления зеркального компонента в модели ADS требуется найти скалярное произведение векторов отражения (**r**) и направления на наблюдателя (**v**).

$I_s = L_s K_s (\mathbf{r} \cdot \mathbf{v})^f$.

Чтобы решить это уравнение, нужно найти вектор отражения (**r**), который является отражением вектора направления на источник света (**s**) относительно вектора нормали (**n**):

$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \mathbf{n})\mathbf{n}$.

Эта формула реализована в виде функции GLSL: `reflect`.

В вычислениях выше требуется найти скалярное произведение, выполнить операцию сложения и пару операций умножения. Мы можем немного повысить эф-

фективность вычисления зеркального компонента за счет использования следующего наблюдения. Когда угол между векторами **v** и **r** равен нулю, вектор нормали **(n)** оказывается точно посередине между **v** и **s**.

Определим вектор полупути **(h)** как вектор, находящийся посередине между **v** и **s**, где вектор **h** является нормализованной суммой:

h = **v** + **s**.

На рис. 3.4 показано положение вектора полупути относительно других векторов:

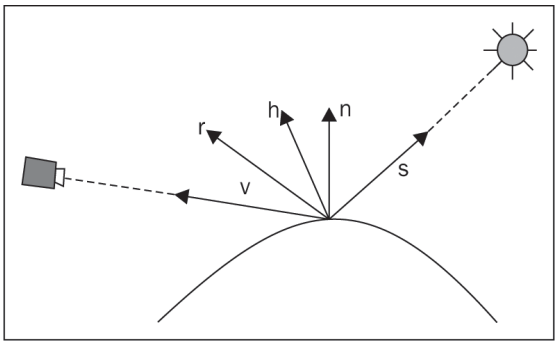

Рис. 3.4 ❖ Положение вектора полупути относительно других векторов

Таким образом, скалярное произведение в формуле вычисления зеркального компонента можно заменить скалярным произведением **h** и **n**:

$I_s = L_s K_s (\mathbf{h} \cdot \mathbf{n})^f$.

Чтобы найти вектор **h**, требуется меньше операций, чем для вычисления вектора **r**, поэтому использование аппроксимации вектора полупути должно дать некоторый прирост производительности. Угол между вектором полупути и вектором нормали пропорционален углу между вектором отражения **(r)** и вектором направления на наблюдателя **(v)**, когда все векторы компланарны. Соответственно, визуальные результаты тоже должны быть схожими, хотя и не одинаковыми.

Подготовка

В этом рецепте можно использовать ту же шейдерную программу, что была представлена в рецепте «Пофрагментное вычисление освещенности для повышения реализма»; настройте основное приложение, как описывается в том рецепте.

Как это делается...

Добавив пару шейдеров из рецепта «Пофрагментное вычисление освещенности для повышения реализма», замените функцию `ads` во фрагментном шейдере следующим кодом:

```
vec3 ads( )
{
    vec3 n = normalize( Normal );
    vec3 s = normalize( vec3(LightPosition) - Position );
    vec3 v = normalize(vec3(-Position));
    vec3 h = normalize( v + s );
    return
        LightIntensity *
            (Ka +
             Kd * max( dot(s, Normal), 0.0 ) +
             Ks * pow(max(dot(h,n),0.0), Shininess ) );
}
```

Как это работает...

Вектор полупути вычисляется как нормализованная сумма векторов направления на наблюдателя (`v`) и направления на источник света (`s`). Получившийся вектор полупути сохраняется в переменной `h`.

В вычислении зеркального компонента в этой версии участвуют скалярное произведение вектора `h` на вектор нормали (`Normal`). Остальные вычисления выполняются, как и прежде.

И еще...

Аппроксимация вектора полупути обеспечивает небольшое увеличение производительности вычислений зеркального компонента и очень похожий визуальный результат. На рис. 3.5 показаны изображение чайника, полученное с использованием аппроксимации вектора полупути (справа) и с применением полной модели ADS, описанной в рецепте «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL» (слева). Метод на основе вектора полупути производит блик, имеющий больший размер, но это не особенно сильно бросается в глаза. При желании можно компенсировать размер блика, увеличив значение `Shininess`.

Рис. 3.5. Изображение чайника, полученное с использованием аппроксимации вектора полупути (справа) и полной модели ADS (слева)

См. также

- Рецепт «Пофрагментное вычисление освещенности для повышения реализма».

Имитация узконаправленных источников света

В прежнем конвейере OpenGL с фиксированной функциональностью имелась возможность определять узконаправленные источники света. Такие источники света, как следует из названия, излучают узкий конус света с вершиной в источнике. Кроме того, сила света имеет максимальное значение вдоль оси конуса и уменьшается к его границам. Это позволяло создавать источники света, дающие эффект, похожий на тот, что производят, например, настоящие прожекторы.

На рис. 3.6 изображены чайник и тор, освещаемые единственным источником узконаправленного света. Обратите внимание на уменьшение освещенности в круге света от центра к краям.

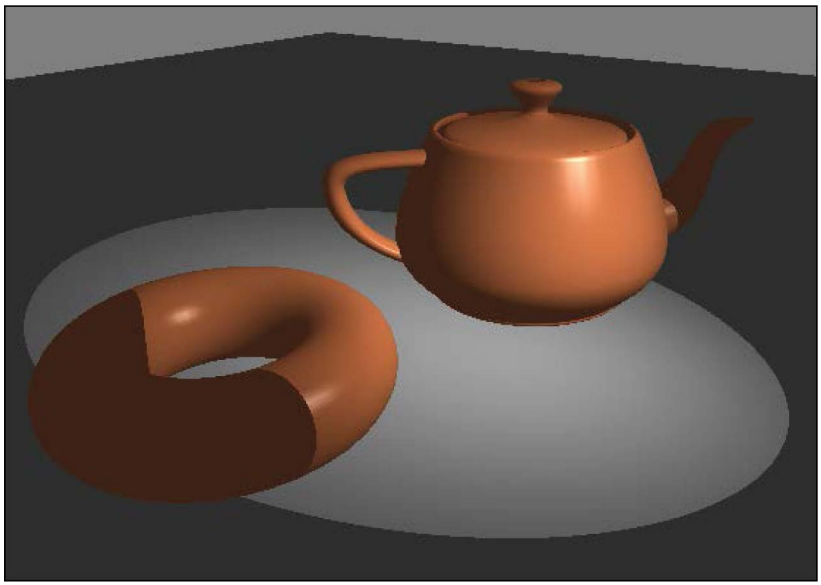

Рис. 3.6 ❖ Сцена, освещаемая единственным источником узконаправленного света

В этом рецепте будет показано, как с помощью шейдеров реализовать эффект освещения узконаправленным источником света, подобный тому, что поддерживался в прежнем конвейере OpenGL с фиксированной функциональностью.

Конус света от узконаправленного источника определяется направлением (**d** на рис. 3.7), углом отсечки (**c** на рис. 3.7) и местоположением (**P** на рис. 3.7). Сила света имеет максимальное значение вдоль оси конуса и уменьшается к его границам.

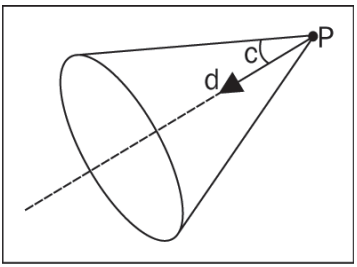

Рис. 3.7 ❖ Параметры конуса света от узконаправленного источника

Подготовка

В этом рецепте можно использовать тот же вершинный шейдер, что был представлен в рецепте «Пофрагментное вычисление освещенности для повышения реализма». Основное приложение должно устанавливать все uniform-переменные, объявленные в вершинном и фрагментном шейдерах, последний из которых представлен ниже.

Как это делается...

Для создания фрагментного шейдера, реализующего модель ADS с узконаправленным источником света, добавьте в него следующий код:

```
in vec3 Position;
in vec3 Normal;

struct SpotLightInfo {
  vec4 position;  // Позиция в видимых координатах
  vec3 intensity; // Интенсивность фонового, рассеянного и зеркального компонентов
  vec3 direction; // Нормализованный вектор направления света
  float exponent; // Экспонента углового ослабления силы света
  float cutoff;   // Угол отсечки (от 0 до 90 градусов)
};
uniform SpotLightInfo Spot;

uniform vec3 Kd;         // Коэффициент отражения рассеянного света
uniform vec3 Ka;         // Коэффициент отражения фонового света
uniform vec3 Ks;         // Коэффициент зеркального отражения
uniform float Shininess; // Показатель степени зеркального отражения

layout( location = 0 ) out vec4 FragColor;

vec3 adsWithSpotlight( )
{
  vec3 s = normalize( vec3( Spot.position) - Position );
  float angle = acos( dot(-s, Spot.direction) );
  float cutoff = radians( clamp( Spot.cutoff, 0.0, 90.0 ) );
  vec3 ambient = Spot.intensity * Ka;

  if( angle < cutoff ) {
```

```
    float spotFactor = pow( dot(-s, Spot.direction),
                            Spot.exponent );
    vec3 v = normalize(vec3(-Position));
    vec3 h = normalize( v + s );
    return
      ambient +
      spotFactor * Spot.intensity * (
        Kd * max( dot(s, Normal), 0.0 ) +
        Ks * pow(max(dot(h,Normal), 0.0),Shininess));
  } else {
    return ambient;
  }
}

void main() {
  FragColor = vec4(adsWithSpotlight(), 1.0);
}
```

Как это работает...

Все параметры источника узконаправленного света сосредоточены в структуре `SpotLightInfo`. Для хранения этих параметров объявляется единственная uniform-переменная с именем `Spot`. Поле `position` определяет местоположение источника света в системе видимых координат. Поле `intensity` определяет интенсивность трех компонентов света (фоновый, рассеянный и зеркальный). При желании эти значения можно хранить в трех разных полях. Поле `direction` содержит направление распространения света от источника, определяющее ось конуса света. Этот вектор должен задаваться в системе видимых координат. В основной программе OpenGL его следует преобразовать с помощью нормальной матрицы, как это обычно делается для нормализации векторов. Эту операцию можно было бы выполнять внутри шейдера; однако в шейдере могла бы быть задана нормальная матрица для отображаемого объекта, непригодная для преобразования вектора направления источника света.

Поле `exponent` определяет показатель степени, используемый для вычисления углового ослабления пучка света. Сила света ослабевает в направлении от оси конуса к его краям, пропорционально косинусу угла между вектором направления источника света и вектором к точке на освещаемой поверхности (отрицательное значение переменной `s`). Значение косинуса угла возводится в степень, заданную в этом поле. Чем больше значение поля `exponent`, тем быстрее уменьшается интенсивность света в конусе. Это очень похоже на ослабление освещенности в пределах светового блика.

Поле `cutoff` определяет угол между центральной осью и внешней границей конуса света. Здесь этот угол определяется в диапазоне от 0 до 90 градусов.

Функция `adsWithSpotlight` вычисляет освещенность по стандартному уравнению модели ADS, используя параметры заданного источника света. В первой строке вычисляется вектор направления от точки на поверхности к источнику света (`s`). Далее этот вектор направления нормализуется и сохраняется в переменной `spotDir`. Затем вычисляется угол между `spotDir` и обратным вектором `s` и

сохраняется в переменной `angle`. В переменную `cutoff` сохраняется значение поля `Spot.cutoff`, преобразованное из градусов в радианы. Далее вычисляется фоновый компонент и сохраняется в переменной `ambient`.

После этого значение переменной `angle` сравнивается с переменной `cutoff`. Если значение `angle` меньше, чем значение `cutoff`, это означает, что точка на освещаемой поверхности находится в пределах конуса света. Иначе данная точка освещается только фоновым светом, и функция возвращает лишь значение фонового компонента.

Если значение `angle` меньше, чем значение `cutoff`, вычисляется значение переменной `spotFactor`, возведением скалярного произведения `-s` и `spotDir` в степень `Spot.exponent`. Затем переменная `spotFactor` используется как коэффициент ослабления силы света, чтобы обеспечить максимальную интенсивность освещения в центре конуса и все более слабую по мере приближения к границам конуса. В заключение решается уравнение освещенности в модели ADS, как обычно, с той лишь разницей, что рассеянный и зеркальный компоненты масштабируются коэффициентом `spotFactor`.

См. также

- Рецепт «Пофрагментное вычисление освещенности для повышения реализма».
- Рецепт «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL».

Придание изображению «мультяшного» вида

Тун-шейдинг (toon-shading) (также иногда называют **цел-шейдингом (cel-shading)**) – прием создания нефотореалистичного отображения, результатом которого является компьютерное изображение, имитирующее изображения в мультипликационных фильмах, нарисованных вручную[1]. Существует множество способов реализации этого приема. В данном рецепте демонстрируется наиболее простой из них, основанный на немного измененной модели вычисления фонового и рассеянного компонентов освещения.

Основным результатом применения этого приема является появление больших областей одного цвета с резкими переходами между ними. Так имитируется раскрашивание художником объектов с использованием кисти или карандаша. На рис. 3.8 показан пример отображения чайника и тора с применением приема тун-шейдинга.

Приемы, описываемые здесь, основаны только на вычислении фонового и рассеянного компонентов освещения и градации косинуса рассеянного компонента.

---

[1] Cel – сокращение от *celluloid* (целлулоид – из этого материала делаются прозрачные листы, на которых рисуется традиционная анимация). Такой тип отображения часто используется для одушевления комиксов, для него характерны жёсткие контуры, ограниченное число цветов, неплавные переходы светотени. – *Прим. перев.*

Рис. 3.8 ❖ Пример отображения чайника и тора с применением приема тун-шейдинга

Иными словами, значение скалярного произведения, обычно используемого для вычисления рассеянного компонента, ограничивается фиксированным числом возможных значений. Следующая таблица иллюстрирует эту концепцию для четырех уровней:

Косинус угла между s и n	Используемое значение
Между 1.0 и 0.75	0.75
Между 0.75 и 0.5	0.5
Между 0.5 и 0.25	0.25
Между 0.25 и 0.0	0.0

В предыдущей таблице **s** – это вектор, направленный на источник света, а **n** – это вектор нормали к поверхности. За счет ограничения значений косинуса на изображении появляются резкие переходы между участками с разной степенью освещенности (см. рис. 3.8), имитирующие мазки кистью, нанесенные вручную.

Подготовка

В этом рецепте можно использовать тот же вершинный шейдер, что был представлен в рецепте «Пофрагментное вычисление освещенности для повышения реализма». Основное приложение должно устанавливать все uniform-переменные, объявленные в вершинном и фрагментном шейдерах, последний из которых представлен ниже.

Как это делается...

Для создания фрагментного шейдера, реализующего эффект тун-шейдинга, добавьте в него следующий код:

```
in vec3 Position;
in vec3 Normal;

struct LightInfo {
  vec4 position;
  vec3 intensity;
};
uniform LightInfo Light;

uniform vec3 Kd; // Коэффициент отражения рассеянного света
uniform vec3 Ka; // Коэффициент отражения фонового света

const int levels = 3;
const float scaleFactor = 1.0 / levels;
layout( location = 0 ) out vec4 FragColor;
vec3 toonShade( )
{
  vec3 s = normalize( Light.position.xyz - Position.xyz );
  float cosine = max( 0.0, dot( s, Normal ) );
  vec3 diffuse = Kd * floor( cosine * levels ) *
                 scaleFactor;

  return Light.intensity * (Ka + diffuse);
}

void main() {
  FragColor = vec4(toonShade(), 1.0);
}
```

Как это работает...

Константная переменная `levels` определяет число градаций значений, используемых при вычислении рассеянного компонента освещения. Эту переменную можно было бы объявить как uniform-переменную, чтобы дать возможность определять число уровней в основном приложении. Эта переменная будет использоваться для градации значения косинуса при вычислении рассеянного компонента освещения.

Самой важной частью фрагментного шейдера является функция `toonShade`. Она начинается с вычисления вектора `s`, определяющего направление на источник света. Далее вычисляется косинус скалярного произведения вектора `s` на вектор `Normal`. После этого вычисляется значение градации. Так как оба вектора нормализованы и негативные значения отсеиваются функцией `max`, можно быть уверенными, что значение косинуса будет находиться в диапазоне от нуля до единицы. За счет умножения этого значения на число уровней градации `levels` и округления вниз получается целочисленный результат в диапазоне от 0 до `levels - 1`. После деления результата на `levels` (за счет умножения на `scaleFactor`) опять получается вещественное значение в диапазоне от нуля до единицы. Но на этот раз результатом будет одно из `levels` возможных значений. После этого результат умножается на `Kd`, коэффициент отражения рассеянного света.

Наконец, рассеянный и фоновый компоненты складываются, и получается окончательный цвет фрагмента.

И еще...

Для градации косинуса точно так же можно было бы использовать функцию `ceil` вместо `floor`. В этом случае результаты просто сдвинулись бы на один уровень вверх, и изображение получилось бы немного ярче.

Часто для придания более явного «мультяшного» эффекта силуэты и другие границы фигур обводят черными линиями. Модель вычисления освещения, представленная в этом рецепте, не предусматривает такой возможности. Однако существуют приемы, позволяющие добиться подобного эффекта, и далее в книге я покажу один из них.

См. также

- Рецепт «Пофрагментное вычисление освещенности для повышения реализма».
- Рецепт «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL».
- Рецепт «Рисование линий силуэтов с помощью геометрического шейдера» в главе 6 «Использование геометрических шейдеров и шейдеров тесселяции».

Имитация тумана

Простой эффект тумана можно получить за счет смешивания цвета каждого фрагмента с постоянным цветом тумана. Плотность тумана определяется расстоянием объекта от камеры. При реализации эффекта можно использовать линейную или нелинейную (например, экспоненциальную) зависимость между расстоянием и плотностью тумана.

На рис. 3.9 изображены четыре чайника с эффектом тумана, произведенным за счет подмешивания цвета тумана в линейной зависимости от расстояния.

Рис. 3.9 ❖ Эффект тумана

Линейная зависимость в данном рецепте определяется следующим уравнением:

$$f = \frac{d_{\max} - |z|}{d_{\max} - d_{\min}},$$

где $d_{\min}$ – расстояние от наблюдателя до точки, где начинает проявляться эффект тумана, $d_{\max}$ – расстояние до точки, за которой туман полностью скрывает объекты. Переменная z – расстояние от наблюдателя до объекта. Значение f – это коэффициент плотности тумана. Нулевое значение этого коэффициента соответствует 100%-ной плотности тумана, а значение, равное единице, – отсутствию тумана. Так как обычно плотность тумана растет с расстоянием, коэффициент плотности тумана имеет минимальное значение, когда значение $|z|$ равно $d_{\max}$, и максимальное, когда значение $|z|$ равно $d_{\min}$.

Так как расчеты выполняются во фрагментном шейдере, эффект будет наблюдаться только на фоне видимых объектов. Туман не будет виден на «пустых» участках сцены. Чтобы добиться целостности восприятия эффекта, следует выбрать цвет фона, соответствующий цвету тумана максимальной плотности.

Подготовка

В этом рецепте можно использовать тот же вершинный шейдер, что был представлен в рецепте «Пофрагментное вычисление освещенности для повышения реализма». Основное приложение должно устанавливать все uniform-переменные, объявленные в вершинном и фрагментном шейдерах, последний из которых представлен ниже.

Как это делается...

Для создания фрагментного шейдера, реализующего эффект тумана, добавьте в него следующий код:

```
in vec3 Position;
in vec3 Normal;

struc tLightInfo {
  vec4 position;
  vec3 intensity;
};
uniform LightInfo Light;

struct FogInfo {
  float maxDist;
  float minDist;
  vec3 color;
};
uniform FogInfo Fog;

uniform vec3 Kd;          // Коэффициент отражения рассеянного света
uniform vec3 Ka;          // Коэффициент отражения фонового света
uniform vec3 Ks;          // Коэффициент зеркального отражения
uniform float Shininess; // Показатель степени зеркального отражения
```

```
layout( location = 0 ) out vec4 FragColor;

vec3 ads( )
{
  // ... Реализация вычислений в соответствии с моделью ADS
}

void main() {
  float dist = abs( Position.z );
  float fogFactor = (Fog.maxDist - dist) /
                    (Fog.maxDist - Fog.minDist);
  fogFactor = clamp( fogFactor, 0.0, 1.0 );
  vec3 shadeColor = ads();
  vec3 color = mix( Fog.color, shadeColor, fogFactor );

  FragColor = vec4(color, 1.0);
}
```

Как это работает...

В этом шейдере используется в точности такая же функция `ads`, как и в рецепте «Использование вектора полупути для повышения производительности». Все, что относится к эффекту тумана, в этом шейдере находится в функции `main`.

Uniform-переменная `Fog` содержит параметры, определяющие плотность и цвет тумана. Поле `minDist` – это расстояние от наблюдателя до точки, где начинает проявляться эффект тумана, а поле `maxDist` – расстояние до точки, за которой плотность тумана становится максимальной. Поле `color` задает цвет тумана.

Переменная `dist` используется для хранения расстояния от точки на поверхности до наблюдателя, роль которого выполняет координата Z точки. Значение переменной `fogFactor` вычисляется с использованием предыдущего уравнения. Так как точка на расстоянии `dist` может оказаться за границами `minDist` и `maxDist`, значение переменной `fogFactor` ограничивается диапазоном от нуля до единицы.

Затем вызывается функция `ads`, вычисляющая освещенность в соответствии с моделью ADS. Возвращаемый результат сохраняется в переменной `shadeColor`.

Наконец выполняется смешивание `shadeColor` и `Fog.color` с учетом значения `fogFactor`, и полученный результат принимается как цвет фрагмента.

И еще...

В этом рецепте используется линейная зависимость между плотностью тумана и расстоянием от наблюдателя. Как вариант можно было бы использовать экспоненциальную зависимость, как, к примеру, в следующей формуле:

$$f = e^{-d|z|},$$

где **d** представляет плотность тумана. Большим значениям **d** соответствует большая плотность тумана. Также можно было бы использовать квадратичную экспоненциальную зависимость, дающую немного другой эффект (более быстрое нарастание плотности тумана с расстоянием):

$$f = e^{-(dz)^2}.$$

Вычисление расстояния от наблюдателя

В примере выше в качестве оценки расстояния от наблюдателя до точки поверхности использовалась координата *Z* этой точки. Иногда это может приводить к некоторой потере реалистичности эффекта. Для более точного вычисления расстояния можно заменить строку

```
float dist = abs( Position.z );
```

на

```
float dist = length( Position.xyz );
```

Последняя версия требует вычисления квадратного корня и, соответственно, будет работать немного медленнее.

См. также

- Рецепт «Пофрагментное вычисление освещенности для повышения реализма».
- Рецепт «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL».

Настройка проверки глубины

В GLSL 4 имеется возможность настроить работу механизма проверки глубины, что позволяет получить более полный контроль над тем, как и когда будет проверяться местоположение фрагментов в буфере глубины.

Многие реализации OpenGL автоматически поддерживают оптимизацию, известную как ранняя проверка глубины, или ранняя проверка фрагмента. В соответствии с этой оптимизацией проверка глубины производится до выполнения фрагментного шейдера. Фрагменты, для которых проверка глубины вернет отрицательный результат, не отображаются на экране (или в буфере кадра), поэтому для таких фрагментов можно не вызывать фрагментный шейдер и сэкономить на этом время.

Однако спецификация OpenGL требует, чтобы проверка глубины производилась после выполнения фрагментного шейдера. Это означает, что реализация должна проявлять особую осторожность при использовании оптимизации ранней проверки глубины. Реализация должна убедиться, что внутри фрагментного шейдера не выполняется операций, которые могут оказать влияние на результат проверки глубины, в противном случае она должна запретить данную оптимизацию.

Например, фрагментный шейдер может изменить глубину фрагмента, присвоив новое значение встроенной выходной переменной `gl_FragDepth`. Если в шейдере выполняется такая операция, ранняя проверка глубины не может использоваться, потому что окончательная глубина фрагмента будет известна только после выполнения шейдера. Однако в GLSL имеются способы известить конвейер, как примерно будет изменена глубина, чтобы реализация смогла определить, когда она может использовать раннюю проверку.

В некоторых случаях фрагментный шейдер может отбрасывать фрагменты с помощью ключевого слова discard. Если есть вероятность, что фрагмент будет отброшен, некоторые реализации могут не выполнять раннюю проверку глубины.

Иногда также бывают ситуации, когда было бы желательно положиться на раннюю проверку глубины. Например, если фрагментный шейдер выполняет запись в другую область памяти, не в буфер кадра (например, в буфер хранения, в файл с изображением или в другую память, запись куда потом нельзя исправить), может быть весьма нежелательно выполнять фрагментный шейдер, если проверка глубины для фрагмента может вернуть отрицательный результат. В GLSL имеется возможность принудительно включать оптимизацию ранней проверки глубины.

Как это делается...

Потребовать от конвейера OpenGL всегда выполнять раннюю проверку глубины можно с помощью следующего квалификатора layout внутри фрагментного шейдера:

```
layout(early_fragment_tests) in;
```

Если фрагментный шейдер изменяет глубину фрагментов, но при этом все равно желательно, чтобы выполнялась ранняя проверка глубины, когда это возможно, воспользуйтесь следующим квалификатором layout в объявлении gl_FragDepth:

```
layout (depth_*) out float gl_FragDepth;
```

Здесь depth_* может принимать одно из следующих значений: depth_any, depth_greater, depth_less или depth_unchanged.

Как это работает...

Следующая инструкция требует от реализации OpenGL всегда выполнять раннюю проверку глубины:

```
layout(early_fragment_tests) in;
```

Но имейте в виду, что если при этом попытаться изменить значение переменной gl_FragDepth, новое значение будет проигнорировано.

Если глубина фрагментов должна изменяться внутри фрагментного шейдера, тогда нельзя использовать принудительную раннюю проверку. Однако есть возможность помочь конвейеру определить, когда можно задействовать раннюю проверку. Сделать это можно применением одного из квалификаторов layout к переменной gl_FragDepth, как показано выше. Он устанавливает некоторые границы изменения значения. Благодаря этому реализация OpenGL сможет определить, можно ли пропустить шейдер, – если выяснится, что изменение глубины не изменит результатов проверки, она сможет применить оптимизацию.

Применение квалификатора layout к выходной переменной gl_FragDepth сообщает реализации OpenGL, как может измениться глубина фрагмента внутри шейдера. Спецификатор depth_any указывает, что изменения возможны в любую сторону. Это значение подразумевается по умолчанию.

Остальные спецификаторы описывают, как может измениться значение относительно `gl_FragCoord.z`:

- `depth_greater`: этот фрагментный шейдер может только увеличить глубину;
- `depth_less`: этот фрагментный шейдер может только уменьшить глубину;
- `depth_unchanged`: этот фрагментный шейдер не может изменить глубину; если шейдер выполнит запись в `gl_FragDepth`, это значение наверняка будет равным `gl_FragCoord.z`.

Если указан один из данных спецификаторов, но глубина фрагмента изменяется вопреки ему, результат предсказать невозможно. Например, если переменная `gl_FragDepth` объявлена со спецификатором `depth_greater`, а глубина фрагмента уменьшится, код будет скомпилирован и выполнен, но не следует ожидать, что результаты будут соответствовать ожиданиям.

Если фрагментный шейдер выполняет запись в `gl_FragDepth`, он должен гарантировать запись во всех случаях. Иными словами, он должен присваивать переменной значение независимо от того, по какой ветке пошло выполнение.

См. также

- Рецепт «Реализация порядконезависимой прозрачности» в главе 5 «Обработка изображений и приемы работы с экранным пространством».

Глава 4

Текстуры

В этой главе описываются следующие рецепты:

- наложение двухмерной текстуры;
- наложение нескольких текстур;
- использование карт прозрачности для удаления пикселей;
- использование карт нормалей;
- имитация отражения с помощью кубической текстуры;
- имитация преломления с помощью кубической текстуры;
- наложение проецируемой текстуры;
- отображение в текстуру;
- использование объектов-семплеров.

Введение

Текстуры являются важнейшим и фундаментальным аспектом приемов отображения в реальном масштабе времени в целом и в OpenGL в частности. Поддержка текстур в шейдерах открывает широкий диапазон возможностей. Текстуры могут служить источниками информации не только о цвете, но и о глубине, параметрах освещенности, картах смещений, векторах нормалей и прочих данных, имеющих отношение к вершинам. Этот список можно продолжать практически до бесконечности. Текстуры являются наиболее широко используемым инструментом создания продвинутых эффектов в программах на основе OpenGL, и такое положение вещей едва ли изменится в обозримом будущем.

В OpenGL 4 появилась возможность выполнять операции чтения/записи с памятью через буферы текстур, буферные объекты хранилища шейдера (Shader Storage Buffer Object) и изображения текстур (операции загрузки/сохранения изображений). Это еще больше усложняет задачу точного определения термина «текстура». В общем случае название «текстура» можно интерпретировать как «буфер, в котором может храниться изображение».

В OpenGL 4.2 появилась поддержка **неизменяемых хранилищ текстур (immutable storage textures)**. Несмотря на такое название, под неизменяемыми хранилищами текстур подразумеваются вовсе не текстуры, а хранилища. То есть после выделения памяти для текстуры ее размер, формат и число слоев остаются фиксированными, но саму текстуру вполне можно изменить. Иными словами, термин «неизменяемые» относится к выделенной памяти, а не к ее содержимому. Неизменяемые хранилища текстур выглядят предпочтительнее в подавляющем большинстве случаев, потому что позволяют избавиться от многочисленных про-

верок во время выполнения и при этом дают определенную степень уверенности в «безопасности», поскольку исключают возможность изменения параметров распределения памяти по ошибке. На протяжении всей книги будут использоваться исключительно неизменяемые хранилища текстур.

Создание неизменяемых хранилищ текстур производится с помощью семейства функций `glTexStorage*`. Программисты с опытом использования текстур имеют привычку пользоваться семейством функций `glTexImage*`, которые все еще поддерживаются, но создают изменяемые хранилища текстур.

В этой главе рассматривается несколько простых и продвинутых приемов. Для начала мы посмотрим, как накладывать обычные цветные текстуры, а затем перейдем к использованию текстур в роли карт нормалей (normal maps) и карт окружений (environment maps). С помощью карт окружений можно имитировать, например, отражение и преломление. Вы увидите пример проекции текстуры на объекты в сцене, подобно тому как проецируются слайды на экран. В заключение будет представлен пример создания изображения непосредственно в текстуре (с использованием объекта буфера кадра (Framebuffer Object, FBO)), с последующим наложением этой текстуры на объект.

Наложение двухмерной текстуры

Наложение текстуры на поверхность в языке GLSL реализуется как последовательность операций с памятью, где находится текстура, извлекающих значение цвета для заданных координат текстуры и присваивающих его выходной переменной фрагментного шейдера. При присваивании цвета может выполняться его смешивание с цветом, который производит модель освещения, модель отражения или какая-то другая модель. В языке GLSL доступ к текстурам осуществляется посредством **переменных-семплеров (sampler)**. Семплер – это «дескриптор» текстурного слота (texture unit). Обычно семплеры объявляются как uniform-переменные внутри шейдера и инициализируются в основном приложении OpenGL путем связывания их с соответствующими текстурными слотами.

В данном рецепте демонстрируется простой пример наложения двухмерной текстуры на поверхность, как показано на рис. 4.1. Значения цвета в текстуре будут смешиваться с цветом, который производит модель **фонового, рассеянного и отраженного освещений (Ambient Diffuse Specular, ADS)**. На рис. 4.1 показаны результаты наложения текстуры с изображением кирпичной кладки на грани куба. Справа показана сама текстура, а слева – результат ее наложения.

Подготовка

Определите в приложении атрибут с индексом `0` для передачи координат вершин, атрибут с индексом `1` для передачи нормалей и атрибут с индексом `2` для передачи координат текстуры. Основное приложение должно также инициализировать и передавать значения параметров модели ADS в uniform-переменных. Дескриптор шейдерной программы должен храниться в переменной `programHandle`.

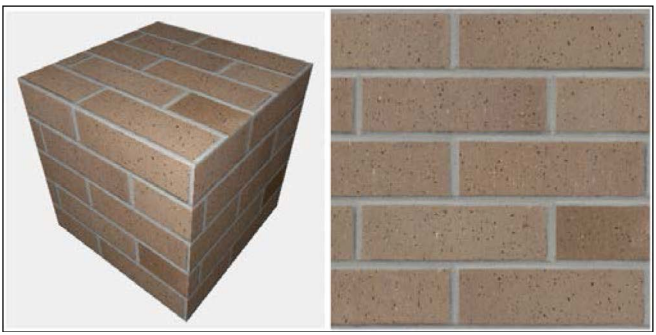

Рис. 4.1 ❖ Результаты наложения текстуры с изображением кирпичной кладки (справа) на грани куба (слева)

Как это делается...

Для отображения простой фигуры с двухмерной текстурой нужно выполнить следующие шаги:

1. Добавить в функцию инициализации приложения следующий код, выполняющий загрузку текстуры. (Здесь используется простой загрузчик изображений в формате TGA, реализация которого входит в состав загружаемых примеров к книге.)

```
GLint width, height;
GLubyte * data = TGAIO::read("brick1.tga", width, height);

// Скопировать файл в память OpenGL
glActiveTexture(GL_TEXTURE0);
GLuint tid;
glGenTextures(1, &tid);
glBindTexture(GL_TEXTURE_2D, tid);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, w, h);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height,
                GL_RGBA, GL_UNSIGNED_BYTE, data);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR);
delete [] data;

// Настроить семплер Tex1 для ссылки на текстурный слот 0
int loc = glGetUniformLocation(programHandle, "Tex1");
if( loc >= 0 )
  glUniform1i(loc, 0);
```

2. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;
```

```
out vec3 Position;
out vec3 Normal;
out vec2 TexCoord;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
  TexCoord = VertexTexCoord;
  Normal = normalize( NormalMatrix * VertexNormal);
  Position = vec3( ModelViewMatrix *
                   vec4(VertexPosition,1.0) );

  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

3. Добавить фрагментный шейдер:

```
in vec3 Position;
in vec3 Normal;
in vec2 TexCoord;

uniform sampler2D Tex1;

struct LightInfo {
  vec4 Position;  // Позиция источника света в видимых координатах
  vec3 Intensity; // Компоненты A,D,S освещенности
};
uniform LightInfo Light;

struct MaterialInfo {
  vec3 Kd;          // Коэффициент отражения рассеянного света
  vec3 Ka;          // Коэффициент отражения фонового света
  vec3 Ks;          // Коэффициент зеркального отражения
  float Shininess; // Показатель степени зеркального отражения
};
uniform MaterialInfo Material;

layout( location = 0 ) out vec4 FragColor;

void phongModel( vec3 pos, vec3 norm,
  out vec3 ambAndDiff, out vec3 spec ) {
    // Вычислить освещенность в соответствии с моделью ADS,
    // вернуть фоновый и рассеянный компоненты в ambAndDiff
    // и вернуть зеркальный компонент в spec
}

void main() {
  vec3 ambAndDiff, spec;
  vec4 texColor = texture( Tex1, TexCoord );
  phongModel(Position, Normal, ambAndDiff, spec);
  FragColor = vec4(ambAndDiff, 1.0) * texColor +
               vec4(spec, 1.0);
}
```

Как это работает...

Первый фрагмент кода реализует загрузку текстуры из файла, копирование ее в память OpenGL и инициализацию переменной-семплера для программы на GLSL. Первый шаг, загрузка файла текстуры, выполняется с помощью простого загрузчика изображений в формате TGA, входящего в состав загружаемых примеров (`TGAIO::read()`). Он читает данные из файла TGA и сохраняет их в массиве беззнаковых байтов, в формате RGBA. Ширина и высота изображения возвращаются в двух последних параметрах. Указатель на массив с данными сохраняется в переменной с именем `data`.

Формат TGA прост, понятен, не обременен никакими патентами и поддерживает полноцветные изображения с альфа-каналом. Это делает его очень удобным форматом для работы с текстурами. Если у вас есть изображения в других форматах, просто загрузите к себе копию программы ImageMagick и с ее помощью преобразуйте свои файлы в формат TGA. Однако следует признать, что формат TGA не особенно экономичен в плане расходования памяти. Если у вас появится желание загружать файлы в других форматах, сделать это можно множеством разных способов. Например, воспользуйтесь программой ResIL (**http://resil.sourceforge.net/**) или Freeimage (**http://freeimage.sourceforge.net/**).

Оставшаяся часть фрагмента кода должна быть знакома программистам, имеющим опыт использования OpenGL. Сначала вызывается функция `glActiveTexture`, назначающая активным текстурный слот `GL_TEXTURE0` (первый текстурный слот, или *текстурный канал*). Все последующие операции с текстурой будут применяться к текстурному слоту с порядковым номером `0`. Следующие две строки создают новый объект текстуры вызовом функции `glGenTextures`, которая возвращает дескриптор нового объекта текстуры, сохраняемый в переменной `tid`. Затем вызывается функция `glBindTexture`, связывающая новый объект текстуры с целью `GL_TEXTURE_2D`. После этого вызовом `glTexStorage2D` выделяется память для неизменяемого хранилища текстуры. Следом, с помощью `glTexSubImage2D`, выполняется копирование данных текстуры в объект текстуры. В последнем аргументе этой функции передается указатель на данные текстуры.

Далее с помощью `glTexParameteri` производится настройка фильтров увеличения и уменьшения для объекта текстуры. В данном случае используется фильтр `GL_LINEAR`.

Фильтры определяют порядок интерполяции цвета элемента текстуры перед его возвратом. Они могут оказывать сильное влияние на качество результатов. В данном примере используется фильтр `GL_LINEAR`, который возвращает среднее взвешенное четырех текселей, ближайших к требуемым координатам. За дополнительной информацией о других фильтрах обращайтесь к описанию функции `glTexParameteri` в документации OpenGL: **http://www.opengl.org/wiki/GLAPI/glTexParameter**.

Затем освобождается память, занятая массивом `data`. Эти данные больше не нужны, потому что были скопированы в память текстуры вызовом `glTexSubImage2D`.

В заключение производится инициализация uniform-переменной `Tex1`, которой присваивается нулевое значение. Это переменная-семплер. Обратите внимание, что она объявлена внутри фрагментного шейдера с типом `sampler2D`. Соответствен-

но, нулевое значение в ней указывает системе OpenGL, что она ссылается на текстурный слот с порядковым номером 0 (который прежде был активирован вызовом функции glActiveTexture).

Вершинный шейдер очень напоминает вершинные шейдеры, использовавшиеся в предыдущих примерах, за исключением объявления дополнительной входной переменной-атрибута VertexTexCoord с индексом 2. Значение этой переменной просто передается дальше, фрагментному шейдеру, путем присваивания выходной переменной TexCoord.

Фрагментный шейдер очень напоминает фрагментные шейдеры из рецептов в предыдущих главах. Важным отличием фрагментного шейдера в этом рецепте является переменная Tex1 – переменная-семплер типа sampler2D, которая была инициализирована в основной программе как ссылка на текстурный слот с порядковым номером 0. Эта переменная вместе с переменной TexCoord (координаты текстуры) используется в функции main для доступа к текстуре. Доступ осуществляется с помощью встроенной функции texture. В первом параметре данной функции передается переменная-семплер, определяющая текстурный слот, к которому осуществляется доступ, а во втором параметре – координаты текстуры. Возвращаемое значение типа vec4 содержит цвет точки с указанными координатами в текстуре (сохраняется в texColor), который в данном случае является результатом интерполяции по четырем ближайшим текселям.

Далее вызовом функции phongModel вычисляются компоненты освещенности в модели ADS, которые возвращаются в параметрах ambAndDiff и spec. Переменная ambAndDiff содержит только фоновый и рассеянный компоненты модели освещенности. Часто цвет текстуры должен корректироваться только цветом рассеянного освещения, без зеркальных бликов. Поэтому в данном примере цвет текстуры умножается на фоновый и рассеянный компоненты, и затем к результату добавляется зеркальный компонент. Полученная сумма сохраняется в выходной переменной FragColor.

И еще...

Существуют разные способы комбинирования цвета текстуры с другими цветами, ассоциированными с фрагментом. В данном примере было решено выполнить перемножение цветов, но кто-то предпочтет использовать только цвет текстуры или смешивать их, каким-то образом опираясь на величину прозрачности.

Как вариант цвет текстуры можно использовать в качестве коэффициента фонового и/или зеркального отражения в модели затенения по Фонгу. Выбор за вами!

Связывание семплера в коде на GLSL

В версии OpenGL 4.2 появилась возможность указывать значение по умолчанию для семплера (значение переменной-семплера) в коде на GLSL. В предыдущем примере значение uniform-переменной задавалось в основной программе, как показано ниже:

```
int loc = glGetUniformLocation(programHandle, "Tex1");
if( loc >= 0 )
  glUniform1i(loc, 0);
```

В версии OpenGL 4.2, напротив, значение по умолчанию можно определить в самом шейдере с помощью квалификатора `layout`:

```
layout (binding=0) uniform sampler2D Tex1;
```

Это поможет упростить код основного приложения и уменьшить число хлопот. В загружаемых примерах кода к этой книге таким способом определяется значение переменной `Tex1`, поэтому загляните туда, если вам нужен более исчерпывающий пример. Этот прием также будет использоваться в следующих рецептах.

См. также

- За более подробной информацией о способах передачи данных в вершинные шейдеры через переменные-атрибуты обращайтесь к рецепту «Передача данных в шейдер с использованием переменных-атрибутов и буферов» в главе 1 «Введение в GLSL».
- Рецепт «Пофрагментное вычисление освещенности для повышения реализма» в главе 3 «Освещение, затенение и оптимизация».

Наложение нескольких текстур

Возможность наложения нескольких текстур на поверхность можно использовать для создания самых разных эффектов. Текстура базового слоя может представлять «чистую» поверхность, а второго, поверхностного слоя – определять дополнительные детали, такие как тени, пятна, потертости или разрушения. Во многих играх в качестве дополнительных текстур применяются карты освещения, несущие дополнительную информацию об отражаемом свете, которые обеспечивают возможность реализации затенения без вычисления компонентов ADS. Такие текстуры иногда называют «предварительно впеченным» освещением.

В этом рецепте демонстрируется один из таких приемов, реализующий наложение двухслойной текстуры. В качестве базового слоя здесь используется полностью непрозрачное изображение кирпичной кладки, а в качестве верхнего слоя – изображение с прозрачными и непрозрачными фрагментами. Непрозрачные фрагменты будут производить эффект мха на кирпичной стене, как показано ниже.

На рис. 4.2 представлен результат наложения двух текстур. Текстуры слева накладываются на куб справа. Базовый слой образует изображение с кирпичной кладкой, а поверх него накладывается текстура с изображением мха. Сквозь прозрачные фрагменты второго изображения просвечивает изображение кирпичной кладки.

Подготовка

Определите в приложении атрибут с индексом `0` для передачи координат вершин, атрибут с индексом `1` для передачи нормалей вершин и атрибут с индексом 2 для

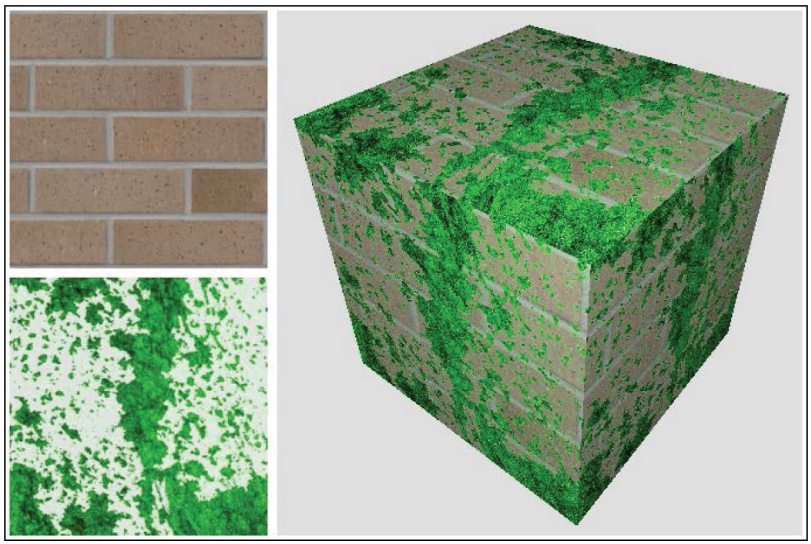

Рис. 4.2 ❖ Результат наложения двух текстур (слева) на куб (справа)

передачи координат текстуры. Параметры модели затенения по Фонгу должны быть объявлены в шейдере как uniform-переменные и инициализироваться в основном приложении.

Как это делается...

Для наложения нескольких текстур на объект нужно выполнить следующие шаги:

1. В разделе инициализации, в основной программе, следует загрузить два изображения в память текстур, как это было сделано в предыдущем рецепте «Наложение двухмерной текстуры». Текстура с изображением кирпичной кладки должна быть загружена в текстурный слот 0, а текстура с изображением мха – в текстурный слот 1, как показано ниже:

```
GLuint texIDs[2];
GLint w, h;
glGenTextures(2, texIDs);

// Загрузить файл с текстурой кирпичной кладки
GLubyte * brickImg = TGAIO::read("brick1.tga", w, h);

// Скопировать текстуру в память OpenGL
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texIDs[0]);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, w, h);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, w, h, GL_RGBA,
                GL_UNSIGNED_BYTE, brickImg);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
```

```
                GL_LINEAR);
delete [] brickImg;

// Загрузить файл с текстурой мха
GLubyte * mossImg = TGAIO::read("moss.tga", w, h);

// Скопировать текстуру в память OpenGL
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texIDs[1]);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, w, h);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, w, h, GL_RGBA,
                GL_UNSIGNED_BYTE, mossImg);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR);
delete [] mossImg;
```

2. Использовать вершинный шейдер из предыдущего рецепта «Наложение двухмерной текстуры».
3. Взяв за основу фрагментный шейдер из рецепта «Наложение двухмерной текстуры», заменить объявление переменной-семплера `Tex1` следующим кодом:

```
layout(binding=0) uniform sampler2D BrickTex;
layout(binding=1) uniform sampler2D MossTex;
```

4. Изменить функцию `main` фрагментного шейдера, как показано ниже:

```
void main() {
  vec3 ambAndDiff, spec;
  vec4 brickTexColor = texture( BrickTex, TexCoord );
  vec4 mossTexColor = texture( MossTex, TexCoord );
  phongModel(Position, Normal, ambAndDiff, spec);
  vec3 texColor = mix(brickTexColor, mossTexColor,
                      mossTexColor.a);
  FragColor = vec4(ambAndDiff, 1.0) * texColor +
              vec4(spec,1.0);
}
```

Как это работает...

Предыдущий программный код в основной программе, загружающий две текстуры, очень похож на аналогичный код из предыдущего рецепта «Наложение двухмерной текстуры». Основное отличие в том, что две текстуры загружаются в разные текстурные слоты. Перед загрузкой текстуры с кирпичной кладкой программа активирует текстурный слот с порядковым номером `0`.

```
glActiveTexture(GL_TEXTURE0);
```

А перед загрузкой второй текстуры она активирует текстурный слот с порядковым номером `1`.

```
glActiveTexture(GL_TEXTURE1);
```

На шаге 3 выполняется связывание семплеров с соответствующими текстурными слотами с помощью квалификатора `layout`.

Доступ к двум текстурам внутри фрагментного шейдера осуществляется посредством соответствующих uniform-переменных, а полученные значения цвета сохраняются в переменных `brickTexColor` и `mossTexColor`. Затем два цвета смешиваются вызовом встроенной функции `mix`. Третий параметр этой функции определяет процентное соотношение смешиваемых цветов. В данном случае используется значение альфа-канала цвета из текстуры с изображением мха. Это обеспечивает линейную интерполяцию двух цветов на основе значения альфа-канала цвета из текстуры с изображением мха. Для тех, кто знаком с функциями смешивания в OpenGL, отмечу, что данная операция дает тот же результат, что и следующий вызов:

```
glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
```

В этом случае цвет из текстуры с изображением мха будет интерпретироваться как исходный, а цвет из текстуры с изображением кирпичной кладки – как целевой цвет.

В заключение выполняется умножение результата смешивания на фоновый и рассеянный компоненты освещения в модели затенения по Фонгу, к произведению прибавляется зеркальный компонент, и полученный результат используется как цвет фрагмента.

И еще...

В этом примере выполняется смешивание цветов двух текстур с применением значения альфа-канала второй текстуры. Это лишь один из множества способов смешивания. Существуют варианты, выбор которых зависит от типов текстур и желаемого результата.

Часто для изменения процентного соотношения смешивания текстур используется дополнительный вершинный атрибут. С помощью этого дополнительного атрибута можно управлять степенью смешивания. Например, таким способом можно было бы изменять количество проросшего мха на стене, на разных ее участках, определяя разные значения в дополнительном атрибуте и тем самым изменяя соотношение смешивания цветов двух текстур. Нулевое значение в этом атрибуте могло бы соответствовать полному отсутствию мха, а значение, равное единице, соответствовало бы варианту смешивания на основе значения альфа-канала в текстуре мха.

См. также

- Рецепт «Наложение двухмерной текстуры».

Использование карт прозрачности для удаления пикселей

Создать эффект дырки в поверхности объекта можно с помощью текстуры, используя значение альфа-канала из нее для определения степени прозрачности

участков поверхности объекта. Однако для этого нужно сделать буфер глубины доступным только для чтения и отображать все полигоны в направлении с заднего плана к переднему, чтобы избежать проблем перекрытия. С этой целью нам пришлось бы отсортировать все полигоны, исходя из местоположения камеры, чтобы обеспечить правильный порядок их отображения. Но это же так трудоемко!

В шейдерах на GLSL можно избежать всех этих неприятностей, воспользовавшись ключевым словом `discard` и с его помощью удаляя требуемые фрагменты, если соответствующее значение альфа-канала в текстуре, представляющей карту прозрачности, окажется ниже некоторого порогового значения. При полном удалении фрагментов нет необходимости изменять содержимое буфера глубины, потому что при удалении фрагментов их глубина вообще никак не учитывается. И не потребуется сортировать полигоны по глубине, потому что не возникает проблемы перекрытия.

На рис. 4.3 справа изображен чайник, из которого удалены фрагменты в соответствии с текстурой слева. Фрагментный шейдер удаляет фрагменты, соответствующие текселям со значением альфа-канала меньше определенного порога.

Рис. 4.3 ❖ Чайник (справа), из которого удалены фрагменты в соответствии с текстурой слева

Если использовать текстуру, цвет пикселей в которой имеет значение альфа-канала, на его основе можно определять, какие фрагменты удалить, а какие оставить: если значение ниже некоторого порогового значения, соответствующий фрагмент удаляется.

Так как в поверхности появятся отверстия, через которые наблюдатель сможет заглянуть внутрь объекта, необходимо использовать методику двустороннего отображения поверхностей объекта.

Подготовка

1. Используйте пару шейдеров с настройками из предыдущего рецепта «Наложение нескольких текстур».

2. Загрузите основную текстуру в текстурный слот 0, а карту прозрачности – в текстурный слот 1.

Как это делается...

Чтобы реализовать удаление фрагментов на основе значений альфа-канала из текстуры, нужно выполнить следующие шаги:

1. Взять вершинный шейдер из предыдущего рецепта «Наложение нескольких текстур» и внести в него изменения, описанные ниже.
2. Заменить объявление uniform-переменной типа sampler2D следующим кодом:

```
layout(binding=0) uniform sampler2D BaseTex;
layout(binding=1) uniform sampler2D AlphaTex;
```

3. Изменить функцию main, как показано ниже:

```
void main() {
  vec4 baseColor = texture( BaseTex, TexCoord );
  vec4 alphaMap = texture( AlphaTex, TexCoord );

  if(alphaMap.a < 0.15 )
    discard;
  else {
    if( gl_FrontFacing ) {
      FragColor = vec4(phongModel(Position,Normal),1.0 ) *
                  baseColor;
    } else {
      FragColor = vec4(phongModel(Position,-Normal),1.0) *
                  baseColor;
    }
  }
}
```

Как это работает...

В функции main внутри фрагментного шейдера извлекается цвет из базовой текстуры и сохраняется в baseColor. Затем извлекается цвет из текстуры, играющей роль карты прозрачности, и сохраняется в alphaMap. Если значение альфа-канала в alphaMap меньше установленного порога (0.15 в данном примере), фрагмент отбрасывается с помощью ключевого слова discard.

В противном случае вычисляются компоненты освещения в соответствии с моделью затенения по Фонгу и вектором нормали, ориентированным согласно принадлежности фрагмента лицевой или обратной стороне поверхности. Результат вычислений умножается на базовый цвет из BaseTex.

И еще...

Этот прием чрезвычайно прост и с успехом может служить заменой традиционным приемам смешивания. Он отлично подходит для создания отверстий в поверхностях объектов или имитации разрушений. Если карта прозрачности содержит значения альфа-канала, постепенно изменяющиеся по всей карте (например, карта

прозрачности может содержать значения альфа-канала, постепенно изменяющиеся с высотой), ее можно использовать для анимации медленного разрушения объекта. Для этого мы могли бы постепенно изменять пороговое значение от `0.0` до `1.0`, чтобы создать анимационный эффект постепенного разрушения объекта.

См. также

- Рецепт «Наложение нескольких текстур».

Использование карт нормалей

Использование карт нормалей (normal mapping) – это один из приемов отображения «деформаций», которые отсутствуют в геометрии поверхностей. Он может пригодиться для отображения вмятин, вздутий или складок на поверхностях, не предоставляя информации о вершинах, полностью описывающей эти деформации. Фактически базовая поверхность остается гладкой, но выглядит деформированной из-за изменения векторов нормалей за счет применения текстуры (с картой нормалей). Этот прием тесно связан с такими приемами рельефного текстурирования, как «bump mapping» и «displacement mapping». Применяя карты нормалей, мы можем изменять векторы нормалей в точках поверхности, исходя из значений, хранящихся в текстурах. В результате поверхность приобретает рельефный вид, и при этом не требуется описывать геометрию этого рельефа в самой поверхности.

Карта нормалей – это текстура, информация из которой интерпретируется не как цвет, а как векторы нормалей. Обычно векторы нормалей в таких картах кодируются следующим образом: красная составляющая цвета используется как координата X, зеленая составляющая – как координата Y, синяя составляющая – как координата Z. При применении такой «текстуры» ее значения используются в вычислениях освещенности фрагментов в роли векторов нормалей, но не как цвет. Таким способом можно заставить поверхность выглядеть деформированной (имеющей вмятины и вздутия, которые фактически отсутствуют в геометрии этой поверхности).

На рис. 4.4 приводится изображение головы огра (с любезного разрешения Кинана Крейна (Keenan Crane)) с наложенной картой нормалей и без нее. Вверху слева представлена базовая текстура, определяющая раскраску головы огра. В этом примере она используется для определения рассеянного компонента в модели затенения по Фонгу. Вверху справа показана голова огра с векторами нормалей по умолчанию после наложения текстуры. Внизу слева изображена текстура с картой нормалей. Внизу справа – голова огра, после наложения текстуры с раскраской и карты нормалей. Обратите внимание на детали рельефа, которые видны на карте нормалей.

Карту нормалей можно создать множеством способов. Многие инструменты трехмерного моделирования, такие как Maya, Blender или 3D Studio Max, способны генерировать карты нормалей. Карты нормалей можно также создавать на основе шкалы яркостей текстур. Данную возможность, например, реализует рас-

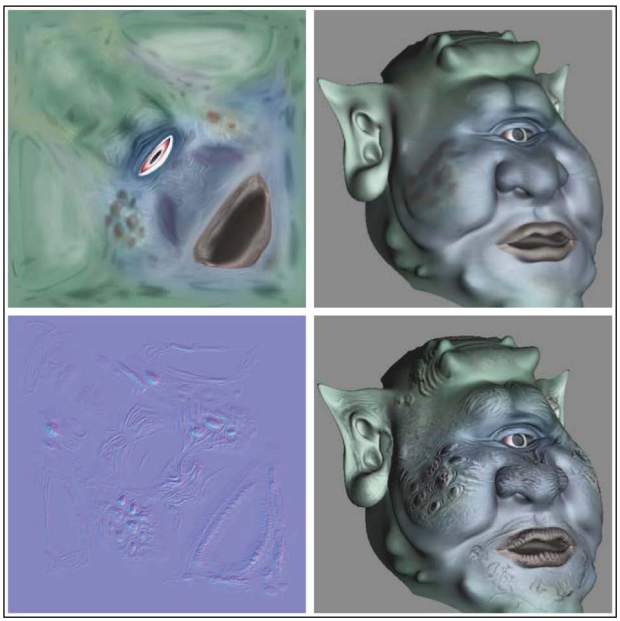

Рис. 4.4 ❖ Базовая текстура с раскраской (вверху слева); голова огра с векторами нормалей по умолчанию после наложения базовой текстуры (вверху справа), карта нормалей (внизу слева); голова огра после наложения базовой текстуры и карты нормалей (внизу справа)

ширение для Adobe Photoshop, созданное в компании NVIDIA **(http://developer.nvidia.com/object/photoshop_dds_plugins.html)**.

Карты нормалей интерпретируются как векторы в **пространстве касательных (tangent space)** (также называется **системой локальных координат объекта (object local coordinate system)**). Начало координат в пространстве касательных находится в точке поверхности, а нормаль к поверхности совпадает с осью Z `(0, 0, 1)`. То есть оси X и Y являются касательными к поверхности. На рис. 4.5 показано, как располагаются оси координат в пространстве касательных в разных точках поверхности.

Преимущество такой системы координат заключается в том, что векторы, хранящиеся в карте нормалей, можно интерпретировать как искажения истинных нормалей, и они не зависят от системы координат объекта. Это избавляет от необходимости преобразовывать нормали, складывать нормали с искажениями и повторно нормализовать их. Вместо этого значения из карты нормалей можно использовать в вычислениях освещенности непосредственно, без каких-либо изменений и преобразований.

Чтобы выполнить всю необходимую работу, нужно вычислить параметры освещенности в пространстве касательных. А для этого в вершинном шейдере нужно

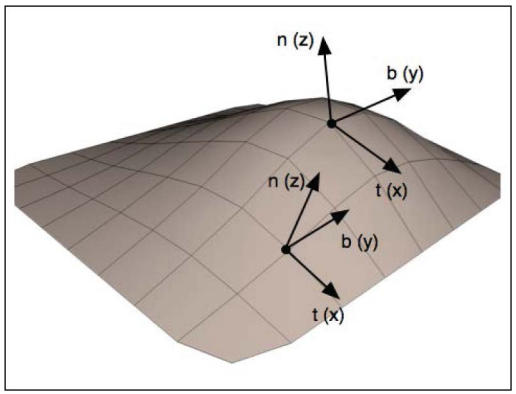

Рис. 4.5 ❖ Оси координат в пространстве касательных в разных точках поверхности

преобразовать векторы, участвующие в вычислениях, в систему координат пространства касательных и передать их фрагментному шейдеру, где будет вычисляться окончательная освещенность фрагментов. Чтобы определить преобразование из системы видимых координат (системы координат наблюдателя) в систему координат пространства касательных, необходимы три ортогональных нормализованных вектора (в системе видимых координат), определяющих систему координат пространства касательных. Ось Z определяется вектором нормали (n), ось X определяется вектором *касательной* (tangent vector, t), и ось Y определяется так называемым *бинормальным* вектором (binormal vector, b). Преобразовать видимые координаты точки P в координаты пространства касательных можно умножением на них следующей матрицы:

$$\begin{bmatrix} S_x \\ S_y \\ S_z \end{bmatrix} = \begin{bmatrix} t_x & t_y & t_O \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix},$$

где S – координаты точки в пространстве касательных, а P – координаты той же точки в видимых координатах. Чтобы применить это преобразование в вершинном шейдере, основная программа должна передать хотя бы два вектора из трех, определяющие локальную систему координат объекта, а также координаты вершины. Обычно передаются вектор нормали (n) и вектор касательной (t). Если вектор касательной известен, бинормальный вектор вычисляется как векторное произведение векторов нормали и касательной.

Иногда вектор касательной включается в структуры данных, которые используются для определения вершин объектов. Если информация о касательных отсутствует, аппроксимацию вектора касательной можно вывести из изменений координат текстуры по поверхности (см. статью «Computing Tangent Space Basis Vectors for an Arbitrary Mesh» Эрика Ленгиэля (Eric Lengyel) на сай-

те проекта Terathon Software 3D Graphics Library: **http://www.terathon.com/code/tangent.html**).

Необходимо позаботиться о согласованном определении векторов касательных по всей поверхности. Иными словами, направления векторов касательных не должны существенно отличаться в соседних вершинах. В противном случае это приведет к неестественным искажениям освещенности.

В следующем примере внутри вершинного шейдера выполняется чтение координат вершины, вектора нормали, вектора касательной и координат текстуры. Затем позиция вершины, нормаль и вектор касательной преобразуются в видимые координаты, после чего вычисляется бинормальный вектор (в системе видимых координат). Далее вычисляются направление взгляда (`v`) и направление на источник света (`s`) (также в системе видимых координат), и затем они преобразуются в систему координат пространства касательных. Преобразованные векторы `v` и `s`, а также координаты текстуры (непреобразованные) передаются во фрагментный шейдер, где вычисляются компоненты освещения в соответствии с моделью затенения по Фонгу и с использованием векторов пространства касательных и вектора нормали, извлеченного из карты нормалей.

Подготовка

Определите в приложении атрибут с индексом `0` для передачи координат вершин, атрибут с индексом `1` для передачи нормалей вершин, атрибут с индексом `2` для передачи координат текстуры и атрибут с индексом `3` для передачи вектора касательной. В этом примере четвертая координата в векторе касательной должна содержать «направленность» (то есть левосторонняя или правосторонняя) системы координат пространства касательных (`-1` или `+1`). На это значение будет умножаться результат векторного произведения.

Загрузите карту нормалей в текстурный слот с порядковым номером `1`, базовую текстуру – в текстурный слот с порядковым номером `0`.

Как это делается...

Для наложения карты нормалей нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;
layout (location = 3) in vec4 VertexTangent;

struct LightInfo {
  vec4 Position;  // Позиция источника света в видимых координатах
  vec3 Intensity; // Компоненты A,D,S освещенности
};
uniform LightInfo Light;

out vec3 LightDir;
out vec2 TexCoord;
```

```
out vec3 ViewDir;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
  // Преобразовать нормаль и касательную в видимые координаты
  vec3 norm = normalize(NormalMatrix * VertexNormal);
  vec3 tang = normalize(NormalMatrix *
                        vec3(VertexTangent));

  // Вычислить бинормальный вектор
  vec3 binormal = normalize( cross( norm, tang ) ) *
                  VertexTangent.w;

  // Матрица преобразования в пространство касательных
  mat3 toObjectLocal = mat3(
    tang.x, binormal.x, norm.x,
    tang.y, binormal.y, norm.y,
    tang.z, binormal.z, norm.z ) ;

  // Получить позицию в видимых координатах
  vec3 pos = vec3( ModelViewMatrix *
                   vec4(VertexPosition,1.0) );

  // Преобразовать направление на источник света и
  // направление взгляда в систему координат пространства
  // касательных
  LightDir = normalize( toObjectLocal *
             (Light.Position.xyz - pos) );
  ViewDir = toObjectLocal * normalize(-pos);

  // Передать координаты текстуры
  TexCoord = VertexTexCoord;

  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить фрагментный шейдер:

```
in vec3 LightDir;
in vec2 TexCoord;
in vec3 ViewDir;

layout(binding=0) uniform sampler2D ColorTex;
layout(binding=1) uniform sampler2D NormalMapTex;

struct LightInfo {
  vec4 Position;  // Позиция источника света в видимых координатах
  vec3 Intensity; // Компоненты A,D,S освещенности
};
uniform LightInfo Light;

struct MaterialInfo {
```

```
  vec3 Ka;          // Коэффициент отражения фонового света
  vec3 Ks;          // Коэффициент зеркального отражения
  float Shininess; // Показатель степени зеркального отражения
};
uniform MaterialInfo Material;

layout( location = 0 ) out vec4 FragColor;

vec3 phongModel( vec3 norm, vec3 diffR ) {
  vec3 r = reflect( -LightDir, norm );
  vec3 ambient = Light.Intensity * Material.Ka;
  float sDotN = max( dot(LightDir, norm), 0.0 );
  vec3 diffuse = Light.Intensity * diffR * sDotN;

  vec3 spec = vec3(0.0);
  if( sDotN > 0.0 )
    spec = Light.Intensity * Material.Ks *
           pow( max( dot(r,ViewDir), 0.0 ),
                Material.Shininess );

  return ambient + diffuse + spec;
}

void main() {
  // Извлечь нормаль из карты нормалей
  vec4 normal = 2.0 * texture( NormalMapTex, TexCoord ) -
                1.0;

  // Базовая текстура определяет параметры отражения
  vec4 texColor = texture( ColorTex, TexCoord );

  FragColor = vec4( phongModel(normal.xyz, texColor.rgb),
                    1.0 );
}
```

Как это работает...

Вершинный шейдер начинается с преобразования векторов нормали и касательной в систему видимых координат умножением на нормальную матрицу (с последующей нормализацией). Затем вычисляется бинормальный вектор как векторное произведение векторов нормали и касательной. Результат умножается на координату `w` вектора касательной, определяющую направленность системы координат пространства касательных. Она может иметь только два значения: `-1` или `+1`.

Далее создается матрица для преобразования видимых координат в координаты пространства касательных и сохраняется в `toObjectLocal`. Координаты вершины преобразуются в систему видимых координат и сохраняются в переменной `pos`. Затем вычитанием `pos` из координат источника вычисляется направление на источник света. Результат умножается на `toObjectLocal`, чтобы перейти к системе координат пространства касательных, нормализуется и сохраняется в выходной переменной `LightDir`. Это значение – направление на источник света в пространстве касательных, которое будет использоваться во фрагментном шейдере для вычисления компонентов освещения в модели затенения по Фонгу.

Аналогично путем нормализации -pos вычисляется направление взгляда и умножением на toObjectLocal преобразуется в систему координат касательных. Результат сохраняется в выходной переменной ViewDir.

Координаты текстуры передаются фрагментному шейдеру в неизменном виде, простым присваиванием их выходной переменной TexCoord.

Векторы, определяющие направление взгляда и направление на источник света, поступают во фрагментный шейдер через переменные LightDir и ViewDir. Функция phongModel немного изменена, по сравнению с предыдущими рецептами. В первом параметре она принимает вектор нормали, а во втором – коэффициент отражения рассеянного света, значение которого извлекается из базовой текстуры. Функция вычисляет параметры освещения в модели затенения по Фонгу, где diffR определяет коэффициент отражения рассеянного света, а LightDir и ViewDir – направление на источник света и направление взгляда.

В функции main из карты нормалей извлекается вектор нормали и сохраняется в переменной normal. Так как текстуры хранят значения от 0 до 1, а компоненты векторов нормалей могут иметь значения от -1 до +1, необходимо преобразовать значения в этот диапазон. Такое преобразование реализовано как умножение значения на 2.0 с последующим вычитанием 1.0 из результата.

Затем из базовой текстуры извлекается цвет, который используется как коэффициент отражения рассеянного света, и сохраняется в переменной texColor. Под конец вызывается функция phongModel со значениями normal и texColor. Она вычисляет освещенность в соответствии с моделью затенения по Фонгу, используя LightDir, ViewDir и norm, а возвращаемый ею результат применяется к выходному фрагменту путем присваивания переменной FragColor.

См. также

- Рецепт «Наложение нескольких текстур».
- Рецепт «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL».

Имитация отражения с помощью кубической текстуры

Текстуры можно использовать для имитации поверхностей с абсолютной отражающей способностью (то есть зеркальных, подобно хромированным деталям). Чтобы добиться такого эффекта, требуется текстура, представляющая отражение обстановки, окружающей объект. Такую текстуру можно наложить на поверхность объекта и получить изображение зеркального объекта. Этот типичный прием известен как **наложение карты окружения (environment mapping)**. Суть приема наложения карты окружения заключается в создании текстуры, представляющей окружение и ее наложение на поверхность объекта. Он часто используется для создания эффектов отражения и преломления.

Кубическая текстура (cube map) – это одна из наиболее распространенных разновидностей текстур, применяемых для наложения карты окружения. Кубическая текстура – это набор из трех отдельных изображений, представляющих окружение, спроецированное на шесть граней куба. Эти шесть изображений представляют, как выглядит окружение с точки зрения наблюдателя, находящегося в центре куба. Пример кубической текстуры показан на рис. 4.6. Изображения даны так, как если бы куб был «развернут» и положен на горизонтальную поверхность. Четыре изображения в середине представляют боковые грани куба, а верхнее и нижнее изображения – верхнюю и нижнюю грани, соответственно.

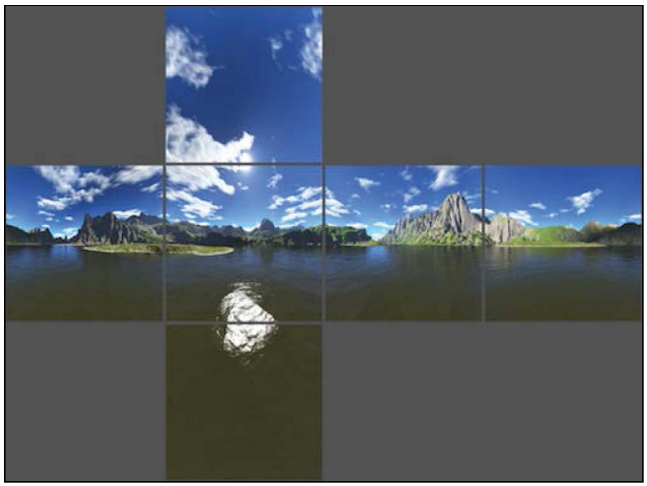

Рис. 4.6 ❖ Пример кубической текстуры

В OpenGL имеется встроенная поддержка кубических текстур (в виде цели `GL_TEXTURE_CUBE_MAP`). Доступ к текселям текстуры осуществляется по трехмерным координатам текстуры (s, t, r). Эти координаты интерпретируются как вектор, направленный из центра куба. Линия, определяемая вектором, начинается в центре куба и простирается до пересечения с одной из его граней. Затем осуществляется обращение к точке пересечения линии с гранью.

Следует признать, что переход от трехмерных координат кубической текстуры к двухмерным координатам, используемым для доступа к отдельным точкам на изображении грани, является довольно сложной операцией. В ней трудно разобраться самостоятельно. Хорошее описание данного процесса можно найти на сайте разработчиков NVIDIA: **http://developer.nvidia.com/content/cube-map-ogl-tutorial**. Однако все не так плохо. Если вы позаботитесь о правильной ориентации текстур в развертке куба, тонкости преобразований координат можно игнорировать и визуализировать координаты текстуры как трехмерный вектор, упоминавшийся выше.

В этом примере демонстрируется применение кубической текстуры для имитации зеркальной поверхности. Кубическая текстура будет использоваться здесь

также для рисования окружения вокруг зеркального объекта (иногда этот прием называют **скайбокс (skybox)**).

Подготовка

Приготовьте шесть изображений для кубической текстуры. В этом примере будут использоваться соглашения по именованию файлов изображений, описанные далее. Имя каждого файла начинается с базовой части имени (хранится в переменной `baseFileName`), за которой следует символ подчеркивания, один из шести суффиксов (`posx`, `negx`, `posy`, `negy`, `posz` и `negz`) и расширение (`.tga`). Суффиксы `posx`, `posy` и другие определяют оси координат, проходящие через грани куба из его центра (положительное направление оси X (positive x), положительное направление оси Y (positive y) и т. д.).

Все изображения должны быть квадратами (предпочтительно со стороной, кратной степени 2) одинакового размера. Их потребуется ориентировать в соответствии с правилами, которыми руководствуется OpenGL для доступа к ним. Как упоминалось выше, это может вызывать некоторые трудности. Один из вариантов заключается в том, чтобы загрузить изображения в их первоначальной ориентации и затем нарисовать окружение (skybox, о том, как это делается, рассказывается далее). Затем повторять переориентацию текстур (методом проб и ошибок), пока они не будут расположены правильно. Можно поступить по-другому: прочитайте описание преобразования на сайте NVIDIA, ссылка на которое приводилась выше, и определите правильную ориентацию, опираясь на преобразования координат текстур.

Определите в приложении атрибут с индексом `0` для передачи координат вершин и атрибут с индексом `1` для передачи нормалей вершин.

Вершинному шейдеру в этом примере потребуется матрица модели (матрица, описывающая преобразования из системы координат объекта в мировую систему координат), отдельная от матрица вида модели (model-view matrix), которая должна передаваться в шейдер из основной программы через отдельную uniform-переменную `ModelMatrix`.

Вершинному шейдеру потребуется также местоположение камеры в мировых координатах, которое также должно передаваться основной программой через uniform-переменную `WorldCameraPosition`.

Как это делается...

Чтобы получить изображение объекта с зеркальной поверхностью с применением кубической текстуры map, а также отобразить саму кубическую текстуру, нужно выполнить следующие шаги:

1. В основной программе загрузить шесть изображений кубической текстуры, как показано ниже:

```
glActiveTexture(GL_TEXTURE0);

GLuint texID;
```

```
glGenTextures(1, &texID);
glBindTexture(GL_TEXTURE_CUBE_MAP, texID);

const char * suffixes[] = { "posx", "negx", "posy",
                            "negy", "posz", "negz" };
GLuint targets[] = {
  GL_TEXTURE_CUBE_MAP_POSITIVE_X,
  GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
  GL_TEXTURE_CUBE_MAP_POSITIVE_Y,
  GL_TEXTURE_CUBE_MAP_NEGATIVE_Y,
  GL_TEXTURE_CUBE_MAP_POSITIVE_Z,
  GL_TEXTURE_CUBE_MAP_NEGATIVE_Z
};
GLint w,h;
glTexStorage2D(GL_TEXTURE_CUBE_MAP, 1, GL_RGBA8, 256, 256);
for( int i = 0; i < 6; i++ ) {
  string texName = string(baseFileName) +
                   "_" + suffixes[i] + ".tga";
  GLubyte *data = TGAIO::read(texName.c_str(), w, h);
  glTexSubImage2D(targets[i], 0, 0, 0, w, h,
                  GL_RGBA, GL_UNSIGNED_BYTE, data);
  delete [] data;
}

// Типовые настройки кубической текстуры
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
                GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
                GL_CLAMP_TO_EDGE);
```

2. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;

out vec3 ReflectDir;     // Направление отраженного луча
uniform bool DrawSkyBox; // Выполняется рисование окружения?
uniform vec3 WorldCameraPosition;
uniform mat4 ModelViewMatrix;
uniform mat4 ModelMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
  if( DrawSkyBox ) {
    ReflectDir = VertexPosition;
```

```
  } else {
    // Вычислить направление отражения в мировых координатах
    vec3 worldPos = vec3( ModelMatrix *
                          vec4(VertexPosition,1.0) );
    vec3 worldNorm = vec3(ModelMatrix *
                          vec4(VertexNormal, 0.0));
    vec3 worldView = normalize( WorldCameraPosition -
                                worldPos );

    ReflectDir = reflect(-worldView, worldNorm );
  }
  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

3. Добавить фрагментный шейдер:

```
in vec3 ReflectDir; // Направление луча отражения

// Кубическая текстура
layout(binding=0) uniform samplerCube CubeMapTex;

uniform bool DrawSkyBox;    // Выполняется рисование окружения?
uniform float ReflectFactor;// Коэффициент отражения
uniform vec4 MaterialColor; // Цвет "оттенка" объекта

layout( location = 0 ) out vec4 FragColor;

void main() {
  // Получить цвет текселя из текстуры
  vec4 cubeMapColor = texture(CubeMapTex,ReflectDir);
  if( DrawSkyBox )
    FragColor = cubeMapColor;
  else
    FragColor = mix(MaterialColor, CubeMapColor, ReflectFactor);
}
```

4. В функции отображения, в основной программе, присвоить uniform-переменной `DrawSkyBox` значение `true` и затем нарисовать куб, окружающий всю сцену с центром в начале координат. Это будет окружение объекта. Затем присвоить uniform-переменной `DrawSkyBox` значение `false` и нарисовать объкт(ы) внутри сцены.

Как это работает...

Кубическая текстура в OpenGL состоит из шести отдельных изображений. Чтобы инициализировать кубическую текстуру, ее нужно связать и затем загрузить все изображения в шесть «слотов» внутри текстуры. В предыдущем примере (в коде основной программы) сначала устанавливается связь текстуры с текстурным слотом `0` вызовом `glActiveTexture`. Затем вызовом `glGenTextures` создается новый объект текстуры, и его дескриптор сохраняется в переменной `texID`, после чего объект текстуры связывается с целью `GL_TEXTURE_CUBE_MAP` вызовом `glBindTexture`. Потом в цикле загружаются шесть файлов, и их содержимое копируется в память

OpenGL вызовом glTexSubImage2D. Обратите внимание, что в первом аргументе этой функции передается цель текстуры – значения GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X и т. д. После завершения цикла кубическая текстура полностью инициализирована шестью изображениями.

Вслед за этим выполняется дополнительная настройка кубической текстуры окружения. Здесь используется линейная фильтрация и для всех трех координат текстуры определяется режим перехода через границы GL_CLAMP_TO_EDGE. Это помогает избавиться от появления цветных границ между гранями куба.

Основная задача вершинного шейдера заключается в вычислении вектора направления отражения и передаче этого вектора во фрагментный шейдер, где он будет использоваться для доступа к текселям кубической текстуры. Результат вычислений сохраняется в выходной переменной ReflectDir. Когда выполняется рисование самого объекта (переменная DrawSkyBox имеет значение false), направление отражения можно вычислить (в мировых координатах) как отражение вектора к наблюдателю относительно вектора нормали.

Направление отражения вычисляется в мировых координатах по той простой причине, что если бы использовались видимые координаты, отражение не изменялось бы при изменении местоположения камеры.

В ветке else, внутри функции main, сначала выполняется преобразование позиции вершины в мировые координаты, с сохранением результата в переменной worldPos. Затем то же преобразование выполняется для вектора нормали (результат сохраняется в worldNorm). Обратите внимание, что для преобразования вектора нормали используется матрица ModelMatrix. В связи с этим важно присвоить четвертой координате вектора нормали значение 0.0, чтобы избежать влияния компонента матрицы модели, описывающего перемещение. Кроме того, компоненты масштабирования в матрице модели должны описывать однородное масштабирование по всем осям; в противном случае вектор нормали будет преобразован неправильно.

Вектор направления на наблюдателя вычисляется в мировых координатах и сохраняется в переменной worldView.

Наконец, выполняется отражение worldView относительно вектора нормали, и результат сохраняется в выходной переменной ReflectDir. Этот вектор будет использоваться фрагментным шейдером для доступа к текселям кубической текстуры, чтобы применить их цвет к фрагментам. Представить себе это можно как луч света, распространяющийся от камеры, ударяющийся о поверхность, отражающийся от нее и попадающий на грань куба. Цвет, который оказывается в точке падения луча на грань куба, и есть тот цвет, которым нужно окрасить фрагмент.

Если выполняется рисование окружения (переменная DrawSkyBox имеет значение true), позиция вершины используется в качестве вектора направления отражения. Почему? Все просто: когда отображается окружение, необходимо, чтобы местоположение в окружении соответствовало местоположению в кубической текстуре (окружение в действительности – это всего лишь изображение кубической текстуры). Внутри фрагментного шейдера переменная ReflectDir будет ис-

пользоваться как координаты для доступа к кубической текстуре. Поэтому, если нужно получить точку на кубической текстуре, соответствующую точке на кубе с центром в начале координат, нужно получить вектор, указывающий на эту точку. Нужным вектором в данном случае являются координаты вершины минус позиция центра координат (который в данном случае имеет координаты `(0,0,0)`). Соответственно, можно просто использовать координаты вершины.

Окружение часто отображается с началом координат в точке местоположения камеры и перемещается по мере движения камеры (чтобы камера всегда оставалась в центре окружения). Эта возможность не предусмотрена в данном рецепте; однако ее можно реализовать преобразованием окружения с применением компонента матрицы вида, описывающего вращение (не перемещения).

Внутри фрагментного шейдера просто осуществляется доступ к кубической текстуре с помощью вектора в переменной `ReflectDir`.

```
vec4 cubeMapColor = texture(CubeMapTex, ReflectDir)
```

При рисовании окружения цвет, извлеченный из текстуры, используется непосредственно, без каких-либо изменений. Но при рисовании объекта требуется смешать цвет окружения с цветом материала объекта. Это поможет придать «оттенок» объекту. Коэффициент отражения определяется переменной `ReflectFactor`. Значение `1.0` соответствует полному отражению, а значение `0.0` – отсутствию отражения. На рис. 4.7 изображен чайник, нарисованный с разными значениями `ReflectFactor`. Слева использован коэффициент отражения `0.5`, а справа – коэффициент отражения `0.85`. Базовый цвет материала – серый. (В качестве кубической текстуры использованы фотографии интерьера собора Святого Петра, Рим. ©Paul Debevec.)

Рис. 4.7 ❖ Чайник, нарисованный с коэффициентами отражения 0.5 (слева) и 0.85 (справа)

И еще...

Существуют два важных обстоятельства, о которых следует помнить всем, кто предполагает использовать данный прием в будущем. Во-первых, объекты будут отражать только окружение. Они не будут отражать изображения других объ-

ектов, находящихся в сцене. Чтобы обеспечить отражение объектов, необходимо сгенерировать карту окружения, как оно видится с позиции каждого объекта, создав шесть изображений, по одному для каждой грани куба. Затем для каждого зеркального объекта использовать соответствующую карту окружения. Конечно, если объекты будут перемещаться относительно друг друга, карты окружения придется повторно генерировать после каждого такого перемещения. Подобная масса вычислений может сделать невозможным создание интерактивной картинки.

Второй момент связан со смещением отражения на поверхностях перемещающихся объектов. Шейдеры в данном примере вычисляют направление отражения и интерпретируют его как вектор, берущий начало в центре окружения. То есть независимо от того, где находится объект, отражение на нем будет выглядеть, как если бы он всегда находился в центре. Иными словами, окружающая обстановка интерпретируется как находящаяся бесконечно далеко от объекта. В главе 19 книги «GPU Gems», написанной Рандимом Фернандо (Randima Fernando), Addison-Wesley Professional, 2009, приводится весьма интересное обсуждение этой проблемы и даются некоторые варианты ее решения.

См. также

- Рецепт «Наложение двухмерной текстуры».

Имитация преломления с помощью кубической текстуры

Как известно, прозрачные объекты способны изменять направление распространения лучей света, проходящих через них. Этот эффект называют **преломлением (refraction)**. При отображении прозрачных объектов этот эффект можно смоделировать с использованием карты окружения, накладывая текстуру окружения на объект так, чтобы получалась имитация отклонения лучей света. Иными словами, можно изменить направление лучей от камеры при прохождении через объект и таким способом получать цвет фрагментов на поверхности объекта.

Как и в предыдущем рецепте, здесь будет использоваться кубическая текстура с изображением окружения. Мы будем отслеживать направление лучей от камеры через объект до пересечения с кубической текстурой.

Преломление света описывается законом Снеллиуса[1], который определяет соотношение между углом падения и углом преломления (см. рис. 4.8).

Согласно закону Снеллиуса, угол падения (a_i) определяется как угол между падающим лучом и вектором нормали к поверхности, а угол преломления (a_t) – как угол между исходящим лучом и вектором, противоположным вектору нормали. Среды, через которые распространяются падающий и преломленный лучи, характеризуются коэффициентами преломления (n_1 и n_2 на рис. 4.8). Отношение

[1] Его еще называют законом Снелля. – *Прим. перев.*

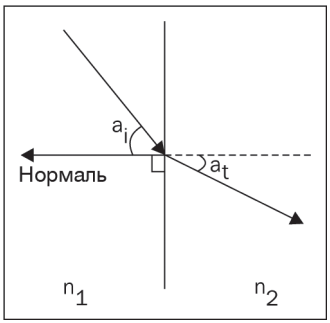

Рис. 4.8 ❖ Соотношение между углом падения и углом преломления

коэффициентов преломления определяет величину угла отклонения луча на границе двух сред.

Опираясь на закон Снеллиуса и добавив немного математики, можно вывести формулу вычисления вектора преломленного луча для заданного отношения коэффициентов преломления, вектора нормали и вектора падающего луча:

$$\frac{\sin a_i}{\sin a_t} = \frac{n_2}{n_1}.$$

Однако в этом нет никакой необходимости, потому что в языке GLSL уже имеется встроенная функция `refract`, вычисляющая вектор преломленного луча. В данном примере будет использоваться эта функция.

Прозрачные объекты пропускают далеко не весь свет, падающий на их поверхность. Часть света отражается. В данном примере этот эффект также будет воспроизведен, но с использованием очень упрощенной модели – более полное решение будет обсуждаться в конце рецепта.

Подготовка

Определите в приложении атрибут с индексом `0` для передачи координат вершин и атрибут с индексом `1` для передачи нормалей вершин. Как и в предыдущем рецепте, в этом примере потребуется передать матрицу модели через uniform-переменную `ModelMatrix`.

Реализуйте загрузку кубической текстуры, как было показано в предыдущем рецепте. Сохраните ее в текстурный слот `0`.

Присвойте uniform-переменной `WorldCameraPosition` координаты камеры в системе мировых координат. Присвойте uniform-переменной `Material.Eta` отношение коэффициентов преломления окружения `n1` и материала объекта `n2` (`n1/n2`). Присвойте uniform-переменной `Material.ReflectionFactor` значение, соответствующее доле света, отражаемого поверхностью (это значение должно быть достаточно маленьким).

Как и в предыдущем примере, если желательно отобразить окружение, присвойте uniform-переменной `DrawSkyBox` значение `true`, затем нарисуйте большой куб, окружающий сцену, и потом присвойте переменной `DrawSkyBox` значение `false`.

Как это делается...

Чтобы получить изображение объекта с отражающей и преломляющей поверхностью, а также самой кубической текстуры, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 ReflectDir; // Направление отражения
out vec3 RefractDir; // Направление преломленного луча

struct MaterialInfo {
  float Eta; // Отношение коэффициентов преломления
  float ReflectionFactor; // Процент отраженного света
};
uniform MaterialInfo Material;

uniform bool DrawSkyBox;

uniform vec3 WorldCameraPosition;
uniform mat4 ModelViewMatrix;
uniform mat4 ModelMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
  if( DrawSkyBox ) {
    ReflectDir = VertexPosition;
  } else {
    vec3 worldPos = vec3( ModelMatrix *
                    vec4(VertexPosition,1.0) );
    vec3 worldNorm = vec3(ModelMatrix *
                     vec4(VertexNormal, 0.0));
    vec3 worldView = normalize( WorldCameraPosition -
                                worldPos );

    ReflectDir = reflect(-worldView, worldNorm );
    RefractDir = refract(-worldView, worldNorm,
                         Material.Eta );
  }
  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить фрагментный шейдер:

```
in vec3 ReflectDir;
in vec3 RefractDir;

layout(binding=0) uniform samplerCube CubeMapTex;
```

```
uniform bool DrawSkyBox;
struct MaterialInfo {
  float Eta; // Отношение коэффициентов преломления
  float ReflectionFactor; // Процент отраженного света
};
uniform MaterialInfo Material;

layout( location = 0 ) out vec4 FragColor;

void main() {
  // Получить цвет текселя из текстуры
  vec4 reflectColor = texture(CubeMapTex, ReflectDir);
  vec4 refractColor = texture(CubeMapTex, RefractDir);

  if( DrawSkyBox )
    FragColor = reflectColor;
  else
    FragColor = mix(refractColor, reflectColor,
                    Material.ReflectionFactor);
}
```

3. В функции отображения, в основной программе, присвоить uniform-переменной `DrawSkyBox` значение `true` и затем нарисовать куб, окружающий всю сцену с центром в начале координат. Это будет окружение объекта. Затем присвоить uniform-переменной `DrawSkyBox` значение `false` и нарисовать объект(ы) внутри сцены.

Как это работает...

Оба шейдера очень похожи на шейдеры в предыдущем рецепте.

Вершинный шейдер преобразует координаты местоположения, векторы нормали и направления в мировые координаты (`worldPos`, `worldNorm` и `worldView`). Затем они используются для вычисления направления отражения с помощью функции `reflect`, а результат сохраняется в выходной переменной `ReflectDir`. Направление распространения преломленного луча вычисляется с использованием встроенной функции `refract` (которой передается отношение коэффициентов преломления `Material.Eta`). Вычисления выполняются на основе закона Снеллиуса, и результат сохраняется в выходной переменной `RefractDir`.

Фрагментный шейдер использует векторы `ReflectDir` и `RefractDir` для получения цвета текселя в кубической текстуре. Цвет отраженного луча сохраняется в переменной `reflectColor`, а цвет преломленного луча – в переменой `refractColor`. Затем эти два цвета смешиваются с учетом отражающей способности материала `Material.ReflectionFactor`. Результатом является смешанный цвет отраженного и преломленного лучей.

На рис. 4.9 изображен чайник, нарисованный в предположении, что доля отраженного света составляет 10% и доля преломленного света 90%. (Cubemap © Paul Debevec.)

Рис. 4.9 ❖ Изображение чайника, полученное с долей отраженного света 10% и долей преломленного света 90%

И еще...

Этот прием имеет те же недостатки, что описывались в разделе «И еще...» в предыдущем рецепте «Имитация отражения с помощью кубической текстуры».

Как и в других приемах, используемых для создания изображений в масштабе реального времени, здесь применяется упрощенная физическая модель. В этом приеме имеется множество аспектов, которые можно было бы улучшить и сделать изображение более реалистичным.

Формулы Френеля

Доля отраженного света в действительности зависит от угла его падения на поверхность. Например, если взглянуть на поверхность озера, присев на корточки на берегу, можно увидеть, что большая часть падающего света отражается от поверхности воды и в отражении легко можно наблюдать окружающий ландшафт. Однако если посмотреть на воду сверху вниз, находясь в лодке, количество отраженного света окажется намного меньше и сквозь толщу воды легко наблюдать все, что находится под поверхностью. Этот эффект описывается формулами Френеля (названными так в честь французского физика Огюста Френеля (Augustin-Jean Fresnel)).

Формулы Френеля описывают количество отраженного света как функцию от угла падения и поляризации света, а также отношения коэффициентов преломления. Если проигнорировать поляризацию, для нас не составит труда реализовать формулы Френеля в предыдущих шейдерах. Как это сделать, очень хорошо опи-

сывается в книге «The OpenGL Shading Language, 3rd Edition» Ренди Дж. Роста (Randi J Rost), Addison-Wesley Professional, 2009[1].

Хроматическая аберрация

Как известно, белый цвет состоит из множества отдельных цветов. Угол преломления света фактически зависит от длины его волны. Этим объясняется эффект расщепления белого света на спектр при прохождении границы между материалами. Наиболее широко известным примером является эффект радуги, производимый при прохождении света через призму.

Этот эффект можно смоделировать программно, предоставив немного разные значения `Eta` для красного, зеленого и синего компонентов света. Для этого можно было бы хранить три разных значения `Eta`, вычислять три разных направления преломления (для красного, зеленого и синего компонентов) и использовать эти направления для извлечения текселей из кубической текстуры. Из первого текселя можно взять только красный компонент, из второго – зеленый, из третьего – синий и объединить эти компоненты, чтобы создать окончательный цвет фрагмента.

Преломление через обе стороны объекта

Важно также отметить, что в продемонстрированной здесь упрощенной модели учитывается преломление только на одной границе поверхности объекта. В действительности же, проходя через прозрачную стенку, луч света пересекает две границы. Однако такое упрощение не приводит к значительной потере реалистичности. Как это часто бывает, при создании изображений в масштабе реального времени нас больше интересует похожесть конечного результата, а не точность его соответствия физическим законам.

См. также

- Рецепт «Имитация отражения с помощью кубической текстуры».

Наложение проецируемой текстуры

Имеется возможность наложить на объект текстуру, как если бы эта текстура проецировалась гипотетическим «проектором слайдов», расположенным где-то внутри сцены. Этот прием часто называют **наложением проецируемой текстуры (projective texture mapping)**, и он позволяет воспроизвести очень интересный эффект.

На рис. 4.10 показан пример наложения проецируемой текстуры. Текстура с изображением цветка слева (автор Стен Шебс (Stan Shebs), Wikimedia Commons) проецируется на чайник и горизонтальную поверхность под ним.

---

[1] На русском языке имеется только более старое издание книги: Рост Р. OpenGL. Трехмерная графика и язык программирования шейдеров. Для профессионалов. – Питер, 2005. – ISBN: 5-469-00383-3. – *Прим. перев.*

Рис. 4.10 ❖ Пример наложения (справа) проецируемой текстуры (слева)

Чтобы спроецировать текстуру на поверхность, достаточно определить координаты текстуры, опираясь на относительное положение поверхности объекта, и координаты источника проецирования («проектора слайдов»). Для простоты представим, что проектор – это камера, находящаяся где-то в пределах сцены. Так же, как задается система координат с центром в местоположении камеры, можно задать систему координат с центром в местоположении проектора и матрицу вида (**V**), описывающую преобразование в систему координат проектора. Далее нужно определить матрицу перспективной проекции (**P**), описывающую преобразование усеченного конуса видимого объема (в системе координат проектора) в прямоугольный объем, имеющий размер 2, с центром в начале координат. Совместив все это и добавив дополнительную матрицу масштабирования и смещения этого объема в объем с размером 1 (центр смещенного объема окажется в точке с координатами `(0.5, 0.5, 0.5)`), мы получим следующую матрицу преобразований:

$$M = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} PV.$$

Основная цель состоит в том, чтобы преобразовать усеченный конус видимого объема в диапазон от `0` до `1` по осям X и Y. Матрицу, представленную выше, как раз можно использовать именно для этого! Она преобразует мировые координаты точки, лежащей где-то в пределах конуса от проектора, в диапазон от `0` до `1`, после чего их можно использовать для доступа к текселям в текстуре. Обратите внимание, что координаты являются однородными (homogeneous) и их значения требуется делить на координату `w`, прежде чем их можно будет использовать в качестве координат действительного местоположения.

Подробнее математический аппарат, на котором основан данный прием, описывается в статье Касса Эверитта (Cass Everitt) на сайте NVIDIA: **http://developer.nvidia.com/content/projectivetexture-mapping**.

В этом примере будет реализован прием наложения единственной проецируемой текстуры.

Подготовка

Определите в приложении атрибут с индексом 0 для передачи координат вершин и атрибут с индексом 1 для передачи нормалей вершин. Основное приложение должно также задавать свойства материала и параметры освещения для модели затенения по Фонгу (см. фрагментный шейдер в следующем разделе). Приложение также должно передавать матрицу модели (для преобразования в мировые координаты) в uniform-переменной ModelMatrix.

Как это делается...

Чтобы наложить проецируемую текстуру, нужно выполнить следующие шаги:

1. В основном приложении загрузить текстуру в текстурный слот 0. Связать объект текстуры с целью GL_TEXTURE_2D и выполнить следующие настройки текстуры:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                GL_CLAMP_TO_BORDER);
```

2. Также в основном приложении необходимо подготовить матрицу преобразования для «проектора» и передать ее через uniform-переменную ProjectorMatrix, как показано ниже. Обратите внимание, что здесь используются функции из библиотеки GLM, обсуждавшейся в главе 1 «Введение в GLSL».

```
vec3 projPos = vec3(2.0f,5.0f,5.0f);
vec3 projAt = vec3(-2.0f,-4.0f,0.0f);
vec3 projUp = vec3(0.0f,1.0f,0.0f);

mat4 projView = glm::lookAt(projPos, projAt, projUp);
mat4 projProj = glm::perspective(30.0f, 1.0f, 0.2f,
                                 1000.0f);
mat4 projScaleTrans = glm::translate(vec3(0.5f)) *
                      glm::scale(vec3(0.5f));
mat4 m = projScaleTrans * projProj * projView;

// Установить значение uniform-переменной
int loc =
  glGetUniformLocation(progHandle,"ProjectorMatrix");
glUniformMatrix4fv(loc, 1, GL_FALSE, &m[0][0]);
```

3. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
```

```
out vec3 EyeNormal;   // Нормаль в видимых координатах
out vec4 EyePosition; // Местоположение в видимых координатах
out vec4 ProjTexCoord;

uniform mat4 ProjectorMatrix;
uniform vec3 WorldCameraPosition;
uniform mat4 ModelViewMatrix;
uniform mat4 ModelMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
  vec4 pos4 = vec4(VertexPosition,1.0);

  EyeNormal = normalize(NormalMatrix * VertexNormal);
  EyePosition = ModelViewMatrix * pos4;
  ProjTexCoord = ProjectorMatrix * (ModelMatrix * pos4);
  gl_Position = MVP * pos4;
}
```

4. Добавить фрагментный шейдер:

```
in vec3 EyeNormal;   // Нормаль в видимых координатах
in vec4 EyePosition; // Местоположение в видимых координатах
in vec4 ProjTexCoord;

layout(binding=0) uniform sampler2D ProjectorTex;

struct MaterialInfo {
  vec3 Kd;
  vec3 Ks;
  vec3 Ka;
  float Shininess;
};
uniform MaterialInfo Material;

struct LightInfo {
  vec3 Intensity;
  vec4 Position; // Позиция источника света в видимых координатах
};
uniform LightInfo Light;

layout( location = 0 ) out vec4 FragColor;

vec3 phongModel( vec3 pos, vec3 norm ) {
  vec3 s = normalize(vec3(Light.Position) - pos);
  vec3 v = normalize(-pos.xyz);
  vec3 r = reflect( -s, norm );
  vec3 ambient = Light.Intensity * Material.Ka;
  float sDotN = max( dot(s,norm), 0.0 );
  vec3 diffuse = Light.Intensity * Material.Kd * sDotN;
  vec3 spec = vec3(0.0);
  if( sDotN > 0.0 )
    spec = Light.Intensity * Material.Ks *
```

```
            pow( max( dot(r,v), 0.0 ), Material.Shininess);

  return ambient + diffuse + spec;
}

void main() {
  vec3 color = phongModel(vec3(EyePosition), EyeNormal);
  vec4 projTexColor = vec4(0.0);
  if( ProjTexCoord.z > 0.0 )
    projTexColor = textureProj(ProjectorTex,ProjTexCoord);

  FragColor = vec4(color,1.0) + projTexColor * 0.5;
}
```

Как это работает...

Загружая текстуру в приложение OpenGL, важно не забыть установить режим перехода через границы для направлений *s* и *t* как `GL_CLAMP_TO_BORDER`. Потому что при выходе координат текстуры за диапазон от `0` до `1` было бы нежелательно продолжать проецировать тексели из текстуры. В этом режиме при использовании цвета границы по умолчанию текстура будет возвращать `(0,0,0,0)` при выходе за ее пределы диапазона от `0` до `1` включительно.

В основном приложении также определяется матрица преобразований для «проектора слайдов». Сначала вызывается функция `glm::lookAt` из библиотеки GLM, которая создает матрицу вида для проектора. В данном примере проектор находится в точке с координатами `(5, 5, 5)`, «смотрит» в точку с координатами `(-2, -4,0)` и имеет «вектор направления вверх» `(0, 1, 0)`. Эта функция действует подобно `gluLookAt`. Она возвращает матрицу для преобразования в систему координат с центром в точке `(5, 5, 5)`, ориентированную в соответствии со вторым и третьим аргументами.

Далее вызовом `glm::perspective` создаются матрица проекции и матрица `M` масштабирования/смещения (показана во введении к этому рецепту). Созданные матрицы сохраняются в переменных `projProj` и `projScaleTrans`, соответственно. Окончательная матрица, являющаяся результатом произведения `projScaleTrans`, `projProj` и `projView`, сохраняется в переменной `m` и передается в uniform-переменную `ProjectorTex`.

В вершинном шейдере имеются три выходные переменные – `EyeNormal`, `EyePosition` и `ProjTexCoord`. Первые две – это вектор нормали к вершине и координаты вершины в видимых координатах. В функции `main` шейдер преобразует входные переменные и присваивает результаты выходным переменным.

Чтобы получить матрицу `ProjTexCoord`, сначала выполняется преобразование позиции в мировые координаты (умножением на матрицу `ModelMatrix`) и затем применяется преобразование проектора.

В функции `main` фрагментного шейдера сначала производятся вычисления в соответствии с моделью затенения по Фонгу, и результат сохраняется в переменной `color`. Следующий шаг – извлечение цвета текселя из текстуры. Однако перед этим необходимо проверить координату Z в переменной `ProjTexCoord`. Если

она имеет отрицательное значение, следовательно, точка находится за проектором, и поиск в текстуре выполнять не требуется. В противном случае вызывается функция `textureProj`, возвращающая цвет текселя из текстуры, который сразу же сохраняется в `projTexColor`.

Функция `textureProj` предназначена для доступа к текстурам с проецированными координатами. Перед обращением к текстуре она делит первые три координаты своего второго аргумента на его последнюю координату. В данном случае это именно то, что требуется. Выше уже упоминалось, что после преобразования с применением матрицы проектора получаются однородные координаты, которые нужно разделить на координату `w` перед обращением к текстуре, что и делает функция `textureProj`.

В заключение цвет текселя из проецируемой текстуры прибавляется к основному цвету, полученному в соответствии с моделью затенения по Фонгу. При этом цвет текстуры немного ослабляется, чтобы он не стал подавляющим.

И еще...

Прием, описанный в этом рецепте, имеет один большой недостаток. Он не поддерживает тени, поэтому проецируемая текстура будет «просвечивать» через любые объекты в сцене и появляться на объектах за ними (относительно проектора). Позднее мы увидим несколько примеров исправления этого недостатка.

См. также

- Рецепт «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL».
- Рецепт «Наложение двухмерной текстуры».

Отображение в текстуру

Иногда бывает желательно создавать текстуры «на лету», в процессе выполнения программы. Текстура может быть шаблоном и генерироваться в соответствии с некоторым внутренним алгоритмом (так называемая **процедурная текстура (procedural texture)**) или представлять некоторый фрагмент сцены. Примером последнего случая может служить имитация экрана видеомонитора, на котором отображается другая часть «мира», возможно, наблюдаемая с помощью скрытой камеры в другой комнате. Изображение на экране монитора может постоянно обновляться, по мере движения объектов в другой комнате, путем отображения «картинки» от скрытой камеры в текстуру, представляющую экран!

На рис. 4.11 показан куб, на грани которого наложена текстура, полученная отображением чайника в нее.

В последних версиях OpenGL отображение непосредственно в текстуру было сильно упрощено введением **объектов буфера кадра (Framebuffer Object, FBO)**. То есть можно создать отдельный целевой буфер для отображения (FBO), связать текстуру с этим объектом и затем отобразить содержимое объекта FBO так же, как

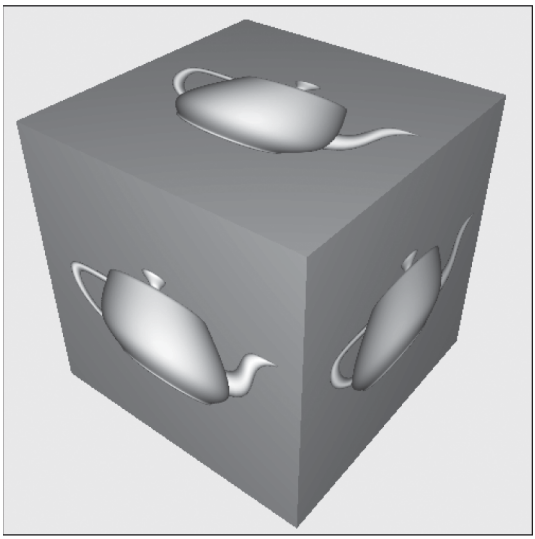

Рис. 4.11 ❖ Куб с текстурой на гранях, полученной путем отображения чайника в нее

содержимое буфера кадра по умолчанию. Для этого нужно лишь подставить объект FBO и убрать его по окончании.

В общих чертах процесс отображения в текстуру включает следующие шаги:

1. Связать с FBO.
2. Отобразить текстуру.
3. Отвязать от FBO (вернуться к использованию буфера кадра по умолчанию).
4. Отобразить сцену с использованием созданной текстуры.

Чтобы задействовать эту разновидность текстур, на стороне GLSL почти ничего не потребуется делать. Фактически для шейдеров такие текстуры будут выглядеть как любые другие. Однако необходимо сделать несколько важных замечаний, касающихся выходных переменных фрагментного шейдера.

В этом примере описываются шаги по созданию FBO и «впеканию» в него текстуры, а также рассказывается, как подготовить шейдер для работы с текстурой.

Подготовка

В этом примере используются шейдеры из рецепта «Наложение двухмерной текстуры» с небольшими изменениями. Подготовьте основную программу, как описывается в том рецепте, и замените в шейдерах имя `Tex1` переменной типа `sampler2D` на `Texture`.

Как это делается...

Чтобы сгенерировать текстуру и задействовать ее в сцене (на втором проходе), нужно выполнить следующие шаги:

1. Выполнить подготовку объекта буфера кадра в основной программе:

```
GLuint fboHandle; // Дескриптор FBO

// Создать и связать буфер кадра
glGenFramebuffers(1, &fboHandle);
glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);

// Создать объект текстуры
GLuint renderTex;
glGenTextures(1, &renderTex);
glActiveTexture(GL_TEXTURE0); // Использовать текстурный слот 0
glBindTexture(GL_TEXTURE_2D, renderTex);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, 512, 512);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);

// Связать текстуру с объектом буфера кадра
glFramebufferTexture2D(GL_FRAMEBUFFER,GL_COLOR_ATTACHMENT0,
                       GL_TEXTURE_2D, renderTex, 0);

// Создать буфер глубины
GLuint depthBuf;
glGenRenderbuffers(1, &depthBuf);
glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
                      512, 512);

// Связать буфер глубины с объектом буфера кадра
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
                          GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER, depthBuf);

// Установить цель для вывода результатов фрагментного шейдера
GLenum drawBufs[] = {GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, drawBufs);

// Отвязать буфер кадра и вернуть буфер по умолчанию
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

2. Создать простую текстуру 1×1, которую можно использовать как «текстуру без текстуры». Обратите внимание, что эта текстура помещается в текстурный слот 1:

```
// Однопиксельная белая текстура
GLuint whiteTexHandle;
GLubyte whiteTex[] = { 255, 255, 255, 255 };
glActiveTexture(GL_TEXTURE1);
glGenTextures(1, &whiteTexHandle);
glBindTexture(GL_TEXTURE_2D,whiteTexHandle);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, 1, 1);
glTexSubImage2D(GL_TEXTURE_2D,0,0,0,1,1,GL_RGBA,
                GL_UNSIGNED_BYTE,whiteTex);
```

3. В функции отображения, в основном приложении, выполнить следующие операции:

```
// Связать с буфером кадра для текстуры
glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);
glViewport(0,0,512,512); // Определить область текстуры

// Использовать "белую" текстуру
int loc = glGetUniformLocation(programHandle, "Texture");
glUniform1i(loc, 1);

// Подготовить матрицу проекции и матрицу вида
// для сцены, отображаемой в текстуру.
// (Не забудьте учесть размеры области текстуры.)

renderTextureScene();

// Отвязать FBO текстуры (вернуться к буферу кадра по умолчанию)
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glViewport(0,0,width,height); // Область главного окна

// Использовать текстуру, связанную с объектом FBO
int loc = glGetUniformLocation(programHandle, "Texture");
glUniform1i(loc, 0);

// Здесь нужно сбросить матрицу проекции и матрицу вида

renderScene();
```

Как это работает...

Начнем исследования с изучения кода, реализующего создание объекта буфера кадра (шаг 1). Объект FBO представляет квадрат со стороной 512 пикселей, который далее предполагается использовать как текстуру. Сначала, вызовом функции `glGenFramebuffers`, создается новый объект FBO, затем вызовом функции `glBindFramebuffer` этот объект связывается с целью `GL_FRAMEBUFFER`. Далее создается объект текстуры, в который будет отображаться изображение, и с помощью функции `glActiveTexture` активируется текстурный слот `0`. Остальная часть кода типична для создания любой другой текстуры. Сначала выделяется пространство для текстуры вызовом `glTexStorage2D`. Нам не требуется копировать в это пространство какие-либо данные (вызовом `glTexSubImage2D`), потому что позднее мы будем писать в эту память, когда будем отображать изображение в FBO.

Затем текстура связывается с объектом FBO вызовом функции `glFramebufferTexture2D`. Эта функция подключит объект текстуры к точке подключения в текущем связанном объекте буфера кадра. Первый аргумент (`GL_FRAMEBUFFER`) указывает, что текстура должна быть подключена к объекту FBO, который в настоящий момент связан с целью `GL_FRAMEBUFFER`. Второй аргумент – это точка подключения. Объекты буфера кадра имеют несколько точек подключения для буферов цвета, один для буфера глубины и еще несколько других точек. Это дает возможность использовать несколько буферов цвета из фрагментных шейдеров. Подробнее об

этом мы поговорим позже. Аргумент GL_COLOR_ATTACHMENT0 указывает, что данная текстура должна быть подключена к точке подключения буфера цвета с индексом 0. Третий аргумент (GL_TEXTURE_2D) – это цель для текстуры, а четвертый (renderTex) – дескриптор текстуры. Последний аргумент (0) определяет уровень разрешения (mip-map level) текстуры, подключенной к объекту FBO. В данном случае предусматривается только один уровень разрешения, поэтому используется значение 0.

Так как нам требуется реализовать отображение в объект FBO с проверкой глубины, необходимо также подключить буфер глубины. Следующие несколько строк кода создают буфер глубины. Функция glGenRenderbuffer создает объект renderbuffer, а функция glRenderbufferStorage выделяет память для него. Второй аргумент в вызове glRenderbufferStorage определяет внутренний формат буфера, а так как данный буфер будет использоваться как буфер глубины, функции передается значение GL_DEPTH_COMPONENT.

Далее вызовом glFramebufferRenderbuffer буфер глубины подключается к точке GL_DEPTH_ATTACHMENT в объекте FBO.

Выходные переменные шейдера подключаются к объекту FBO вызовом glDrawBuffers. Второй аргумент в вызове glDrawBuffers – это массив буферов FBO, связанных с выходными переменными. Элемент массива с индексом *i* соответствует выходной переменной фрагментного шейдера с индексом *i*. В данном случае имеется только одна выходная переменная (FragColor) – переменная с индексом 0. Эта инструкция связывает выходную переменную с GL_COLOR_ATTACHMENT0.

Последняя инструкция в шаге 1 отвязывает объект FBO и выполняет возврат к использованию буфера кадра по умолчанию.

На шаге 2 в текстурном слоте 1 создается белая текстура 1×1. Эта вспомогательная текстура будет использоваться при отображении основной текстуры, чтобы не пришлось ничего менять в шейдере. Так как шейдер умножает цвет текстуры на результат вычислений в соответствии с моделью затенения по Фонгу, эта текстура будет работать как «фон без текстуры», потому что при умножении на нее цвет фрагмента не изменится. При отображении текстуры необходимо использовать «фон без текстуры», но при отображении сцены будет использоваться текстура, подключенная к объекту FBO.

В общем случае использовать текстуру 1×1, конечно же, не требуется. В данном примере она применяется только для того, чтобы обеспечить возможность рисования в объекте FBO без применения текстуры к сцене. Если при отображении текстуры потребуется применить другую текстуру, тогда здесь можно было бы использовать ее.

На шаге 3 (в функции отображения) выполняется связывание с FBO, задействуется текстура «пустого фона» в текстурном слоте 1 и производится отображение текстуры. Обратите внимание, что настройка области текстуры (glViewport), а также подготовка матриц вида и проекции требует особого внимания. Так как объект FBO описывает кадр размером 512×512, в программе выполняется вызов glViewport(0,0,512,512). Аналогичные изменения необходимо внести в матрицы вида и проекции, чтобы обеспечить соответствие отношению сторон области текстуры для сцены, отображаемой в объект FBO.

После отображения текстуры объект FBO отвязывается, производится возврат к прежней области отображения и матрицам вида и проекции, задействуется текстура из объекта FBO (текстурный слот `0`) и выполняется рисование сцены!

И еще...

Так как объекты FBO имеют несколько точек для подключения буферов цвета, есть возможность организовать вывод из фрагментного шейдера в несколько целей. Обратите внимание, что до сих пор все наши фрагментные шейдеры имели единственную выходную переменную с индексом `0`. Соответственно, объект FBO настраивался так, чтобы текстура соответствовала буферу цвета с индексом `0`. В последующих главах мы увидим примеры использования нескольких таких точек подключения для реализации, например, отложенного освещения и затенения (deferred shading).

См. также

- Рецепт «Наложение двухмерной текстуры».

Использование объектов-семплеров

Объекты-семплеры впервые появились в версии OpenGL 3.3. Они обеспечивают удобный способ определения параметров отображения текстур. Традиционно эти параметры определялись с помощью функции `glTexParameter`, обычно в момент создания текстуры. Параметры определяют режим отображения, правила «оборачивания» (wrapping) и скрепления (clamping) и т. д. для заданной текстуры. По сути, подобный способ фактически объединяет текстуру и параметры ее отображения в единый объект. Если одну и ту же текстуру потребуется отобразить разными способами (например, с линейной фильтрацией и без нее), у нас на выбор есть два варианта: либо изменить параметры текстуры вызовами функции `glTexParameter`, либо использовать две копии одной текстуры.

Кроме того, иногда бывает желательно использовать один и тот же набор параметров для отображения нескольких текстур. С помощью приемов, которые мы видели до сих пор, сделать это будет непросто. С помощью объекта-семплера можно определить параметры один раз и использовать их для отображения нескольких разных текстур.

Объекты-семплеры отделяют параметры отображения от объектов текстур. Можно создавать объекты-семплеры, определяющие разные параметры отображения, и применить их к разным текстурам или связать с одной и той же текстурой. Единственный объект-семплер может быть связан со множеством текстур, что дает возможность определить определенный набор параметров один раз и использовать его со множеством объектов текстур.

Объекты-семплеры определяются на стороне OpenGL (не в шейдерах GLSL), что обеспечивает их прозрачность для GLSL.

В этом рецепте будет показано, как определить два объекта-семплера и применить их к одной текстуре. На рис. 4.12 изображен результат. Слева и справа изображена одна и та же текстура. Слева использовался объект-семплер с настроенной фильтрацией по ближайшему соседу, а справа – с линейной фильтрацией.

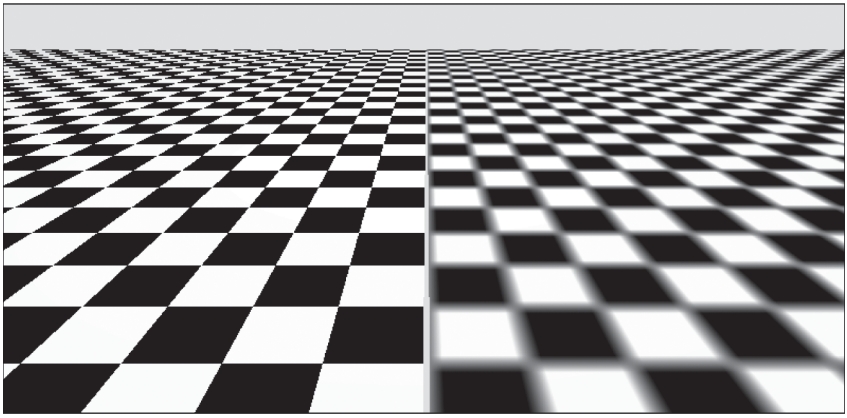

Рис. 4.12 ❖ Результат отображения одной и той же текстуры с разными объектами-семплерами

Подготовка

В этом примере используются шейдеры из рецепта «Наложение двухмерной текстуры». Код шейдеров почти не изменился – в нем лишь используются объекты-семплеры для изменения переменной `Tex1`.

Как это делается...

Чтобы подготовить текстуру и объекты-семплеры, нужно выполнить следующие шаги.

1. Создать и заполнить объект текстуры как обычно, но на этот раз не устанавливать параметры вызовом функции `glTexParameter`.

```
GLuint texID;
glGenTextures(1, &texID);
glBindTexture(GL_TEXTURE_2D, texID);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, w, h);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, w, h, GL_RGBA,
  GL_UNSIGNED_BYTE, data);
```

2. Связать текстуру с текстурным слотом `0`, который используется шейдером.

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texID);
```

3. Далее создать два объекта-семплера и сохранить их дескрипторы в отдельных переменных:

```
GLuint samplers[2];
glGenSamplers(2, samplers);
linearSampler = samplers[0];
nearestSampler = samplers[1];
```

4. Настроить в `linearSampler` параметры линейной интерполяции:

```
glSamplerParameteri(linearSampler, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR);
glSamplerParameteri(linearSampler, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);
```

5. Настроить в `nearestSampler` параметры интерполяции по ближайшему соседу:

```
glSamplerParameteri(nearestSampler, GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
glSamplerParameteri(nearestSampler, GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);
```

6. Перед отображением выполнить привязку требуемого объекта-семплера:

```
glBindSampler(0, nearestSampler);
// Отобразить объекты с фильтрацией по ближайшему соседу
glBindSampler(0, linearSampler);
// Отобразить объекты с линейной фильтрацией
```

Как это работает...

Объекты-семплеры просты в использовании и существенно упрощают переключение между разными наборами параметров отображения для одной текстуры или использование одних и тех же параметров с разными текстурами. Шаги 1 и 2 создают текстуру и связывают ее с текстурным слотом `0`. Обычно параметры отображения текстуры устанавливаются с помощью `glTexParameteri`, но в данном примере они будут устанавливаться с помощью объектов-семплеров и функции `glSamplerParameter`. Шаг 3 создает объекты-семплеры и сохраняет их дескрипторы в переменных. Шаги 4 и 5 определяют параметры отображения с помощью `glSamplerParameter`. Эта функция является почти полной копией `glTexParameter`, за исключением того, что в первом аргументе принимает дескриптор объекта-семплера вместо цели текстуры. В этих шагах определяются параметры для каждого из двух объектов-семплеров (режим линейной фильтрации для `linearSampler` и режим фильтрации по ближайшему соседу для `nearestSampler`).

В конце, с помощью `glBindSampler`, выполняется привязка объектов-семплеров к соответствующему текстурному слоту непосредственно перед отображением текстуры. В шаге 6 с текстурным слотом `0` связывается объект `nearestSampler`, отображается несколько объектов, затем с текстурным слотом `0` связывается `linearSampler` и отображается еще несколько объектов. В результате одна и та же текстура отображается с разными параметрами благодаря связыванию разных объектов-семплеров с текстурным слотом.

См. также

- Рецепт «Наложение двухмерной текстуры».

Глава 5

Обработка изображений и приемы работы с экранным пространством

В этой главе описываются следующие рецепты:

- применение фильтра выделения границ;
- применение фильтра размытия по Гауссу;
- преобразование диапазона яркостей HDR с помощью тональной компрессии;
- эффект размытости на границах ярких участков;
- повышение качества изображения с помощью гамма-коррекции;
- сглаживание множественной выборкой;
- отложенное освещение и затенение;
- реализация порядконезависимой прозрачности.

Введение

В этой главе мы сосредоточимся на приемах работы непосредственно с пикселями в буфере кадра. Эти приемы обычно требуют выполнения нескольких проходов. В первом проходе создается массив исходной информации о пикселях, а на последующих применяются какие-либо эффекты или выполняется дополнительная обработка пикселей. Для реализации приемов используется возможность, предоставляемая в OpenGL, непосредственного отображения в текстуру или во множество текстур (см. рецепт «Отображение в текстуру» в главе 4 «Текстуры»).

Поддержка отображения в текстуру в сочетании с мощью фрагментных шейдеров открывает широчайшие возможности. Внутри фрагментного шейдера можно реализовать такие приемы обработки изображений, как изменение яркости, контрастности, насыщенности и резкости, перед тем как фрагментный шейдер передаст информацию о цвете дальше. Можно применить различные фильтры свертки, такие как выделение границ, размытие или увеличение резкости. В этой главе мы

более детально разберем реализацию подобных фильтров в рецепте «Применение фильтра выделения границ».

Набор взаимосвязанных приемов обеспечивает отображение в текстуру дополнительной информации, помимо традиционной информации о цвете, и затем на последующих проходах подвергает эту информацию дополнительной обработке, чтобы получить окончательное изображение. Эти приемы попадают в категорию, которую часто называют **отложенное освещение и затенение (deferred shading)**.

В этой главе рассматривается несколько примеров применения упомянутых приемов. Сначала будет представлен пример применения фильтров свертки для очерчивания границ и размытия на границах ярких участков. Затем мы перейдем к обсуждению таких важных тем, как гамма-коррекция и сглаживание множественной выборкой. В заключение будет приведен полный пример применения приема отложенного освещения и затенения.

Большинство приемов, представленных в этой главе, выполняют несколько проходов. Чтобы применить фильтр, оперирующий пикселями изображения, сцену сначала нужно отобразить в промежуточный буфер (текстуру). Затем, на заключительном проходе, полученная текстура будет отображаться на экран с применением фильтра в процессе отображения. В последующих рецептах вы увидите несколько разных вариаций на эту тему.

Применение фильтра выделения границ

Выделение границ – это прием обработки изображений, который выявляет области, существенно отличающиеся по яркости изображения. Он позволяет определять границы объектов и изменения в топологии поверхности. Применяется в системах распознавания образов, обработки изображений, анализа изображений и определения структуры изображений. Также может использоваться для создания интересных визуальных эффектов. Например, с его помощью можно создавать трехмерные сцены, напоминающие рисунок карандашом, как показано на рис. 5.1. Чтобы создать такое изображение, сначала было выполнено обычное отображение чайника и тора, а затем во втором проходе к полученному изображению был применен фильтр выделения границ.

Фильтр выделения границ, который используется в данном рецепте, основан на фильтре свертки (convolution filter), или ядре свертки (convolution kernel). Фильтр свертки – это матрица, определяющая преобразования пикселей путем замены их суммами произведений значений соседних пикселей и предопределенных весов. В качестве простого примера рассмотрим фильтр свертки, изображенный на рис. 5.2.

Фильтр 3×3 выделен серым фоном на гипотетической карте пикселей. Жирным выделены числа, представляющие значения фильтра (веса), а обычными числами показаны значения пикселей. Роль значений пикселей может играть яркость в черно-белой шкале или значение одного из компонентов RGB. Применение фильтра к центральному пикселю в серой области заключается в умножении чи-

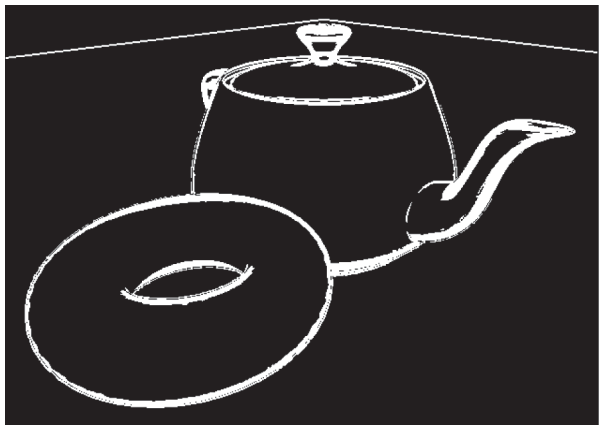

Рис. 5.1 ❖ Результат применения фильтра выделения границ

10	11	12	13	14
1 17	0 18	1 19	20	21
0 24	2 25	0 26	27	28
1 31	0 32	1 33	34	35
38	39	40	41	42

Рис. 5.2 ❖ Фильтр свертки

сел в соответствующих ячейках и суммировании результатов. В данном случае новым значением центрального пикселя (25) будет сумма (17 + 19 + 2 * 25 + 31 + 33), или 150.

Конечно, чтобы применить фильтр свертки, нужно иметь доступ к пикселям оригинального изображения и отдельный буфер для сохранения результатов. Решить эту проблему можно за счет использования двухпроходного алгоритма. В первом проходе будет выполняться отображение сцены в текстуру; а затем, во втором проходе, будут выполняться чтение данных из текстуры, применение фильтра и вывод результатов на экран.

Одним из простейших приемов выделения границ на основе фильтров свертки является так называемый оператор Собеля (Sobel operator). Результатом приме-

нения оператора Собеля в каждой точке изображения является аппроксимация градиента яркости в этой точке. Достигается это за счет применения двух фильтров 3×3, результатами которых являются вертикальный и горизонтальный компоненты градиента. Полученную величину градиента затем можно использовать как признак границы. Когда значение градиента выше некоторого порогового значения, предполагается, что пиксель лежит на границе.

Ниже показаны фильтры 3×3, используемые оператором Собеля:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

Если принять, что результатом применения S_x является s_x, а результатом применения S_y является s_y, тогда оценка величины градиента может быть определена по следующей формуле:

$$g = \sqrt{s_x^2 + s_y^2}.$$

Если значение g окажется выше установленного порога, данный пиксель считается принадлежащим границе, и он выделяется в окончательном изображении.

В этом примере данный фильтр будет применяться во втором проходе двухпроходного алгоритма. В первом проходе будет выполнено отображение сцены с соответствующим освещением, но результаты отображения будут сохранены в текстуре. Во втором проходе текстура будет наложена на прямоугольник экрана, и в процессе наложения будет применен фильтр.

Подготовка

Создайте объект буфера кадра (как это сделать, описывается в рецепте «Отображение в текстуру» в главе 4 «Текстуры») с размерами, соответствующими размерам главного окна. Подключите объект текстуры из первого текстурного слота к первой точке подключения буферов цвета в объекте FBO. В первом проходе сцена будет отображаться непосредственно в текстуру. Определите для этой текстуры фильтры `GL_TEXTURE_MAG_FILTER` и `GL_TEXTURE_MIN_FILTER` со значением `GL_NEAREST` – данный алгоритм не предполагает никакой интерполяции.

Передайте информацию о вершине через атрибут с индексом `0`, вектор нормали в атрибуте с индексом `1` и координаты текстуры – в атрибуте с индексом `2`.

В основном приложении установите значения следующих uniform-переменных:

- `Width`: ширина окна на экране в пикселях;
- `Height`: высота окна на экране в пикселях;
- `EdgeThreshold`: квадрат порогового значения g, выше которого пиксель считается «принадлежащим границе»;
- `RenderTex`: текстура, связанная с объектом FBO.

Также в основном приложении должны быть инициализированы другие uniform-переменные, используемые в вычислениях освещенности.

Как это делается...

Чтобы создать шейдерную программу, которая применяет фильтр Собеля для выделения границ, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 Position;
out vec3 Normal;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
  Normal = normalize( NormalMatrix * VertexNormal);
  Position = vec3( ModelViewMatrix *
                   vec4(VertexPosition,1.0) );

  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить фрагментный шейдер:

```
in vec3 Position;
in vec3 Normal;

// Текстура с результатом первого прохода
layout( binding=0 ) uniform sampler2D RenderTex;

uniform float EdgeThreshold; // Квадрат порогового значения

// Подпрограмма, используемая для выбора функциональности
// pass1 и pass2.
subroutine vec4 RenderPassType();
subroutine uniform RenderPassType RenderPass;

// Другие uniform-переменные для модели затенения по Фонгу
// можно объявить здесь...

layout( location = 0 ) out vec4 FragColor;
const vec3 lum = vec3(0.2126, 0.7152, 0.0722);

vec3 phongModel( vec3 pos, vec3 norm )
{
  // Здесь вычисляется освещенность в соответствии с моделью ADS
}

// Оценка яркости значения в формате RGB.
float luminance( vec3 color ) {
  return dot(lum, color);
}

subroutine (RenderPassType)
```

```
vec4 pass1()
{
  return vec4(phongModel( Position, Normal ),1.0);
}

subroutine( RenderPassType )
vec4 pass2()
{
  ivec2 pix = ivec2(gl_FragCoord.xy);
  float s00 = luminance(
  texelFetchOffset(RenderTex, pix, 0,
                   ivec2(-1,1)).rgb);
  float s10 = luminance(
                texelFetchOffset(RenderTex, pix, 0,
                                 ivec2(-1,0)).rgb);
  float s20 = luminance(
                texelFetchOffset(RenderTex, pix, 0,
                                 ivec2(-1,-1)).rgb);
  float s01 = luminance(
                texelFetchOffset(RenderTex, pix, 0,
                                 ivec2(0,1)).rgb);
  float s21 = luminance(
                texelFetchOffset(RenderTex, pix, 0,
                                 ivec2(0,-1)).rgb);
  float s02 = luminance(
                texelFetchOffset(RenderTex, pix, 0,
                                 ivec2(1,1)).rgb);
  float s12 = luminance(
                texelFetchOffset(RenderTex, pix, 0,
                                 ivec2(1,0)).rgb);
  float s22 = luminance(
                texelFetchOffset(RenderTex, pix, 0,
                                 ivec2(1,-1)).rgb);

  float sx = s00 + 2 * s10 + s20 - (s02 + 2 * s12 + s22);
  float sy = s00 + 2 * s01 + s02 - (s20 + 2 * s21 + s22);

  float g = sx * sx + sy * sy;

  if( g > EdgeThreshold ) return vec4(1.0);
  else return vec4(0.0,0.0,0.0,1.0);
}

void main()
{
  // Следующая инструкция будет вызывать pass1() или pass2()
  FragColor = RenderPass();
}
```

В первом проходе в функции отображения нужно выполнить следующие шаги:

1. Выбрать объект буфера кадра (FBO) и очистить буферы цвета/глубины.
2. Выбрать функцию `pass1` подпрограммы (см. рецепт «Использование подпрограмм для выбора функциональности в шейдере» в главе 2 «Основы шейдеров GLSL»).

3. Настроить матрицы модели, вида и проекции и нарисовать сцену.

Во втором проходе в функции отображения нужно выполнить следующие шаги:

1. Вернуться к использованию буфера кадра по умолчанию и очистить буферы цвета/глубины.
2. Выбрать функцию `pass2` подпрограммы.
3. Определить матрицы модели, вида и проекции как единичные матрицы.
4. Нарисовать прямоугольник (или два треугольника), заполняющий весь экран (от координаты `-1` до `+1` по осям X и Y), с координатами текстуры в диапазоне от `0` до `1` по каждой оси.

Как это работает...

В первом проходе вся сцена целиком отображается в текстуру. Предварительно выбирается функция `pass1` подпрограммы, которая просто вычисляет освещенность объектов в соответствии с моделью затенения по Фонгу (см. рецепт «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL»).

В начале второго прохода выбирается функция `pass2` подпрограммы и отображается единственный прямоугольник, покрывающий весь экран. Цель этой операции – вызвать фрагментный шейдер для каждого пикселя в изображении. Функция `pass2` извлекает значения восьми соседних пикселей текстуры, содержащих результаты, полученные в первом проходе, и вычисляет яркость вызовом функции `luminance`. Затем к полученным значениям применяются горизонтальный и вертикальный фильтры Собеля, и результаты сохраняются в переменных `sx` и `sy`.

Функция `luminance` определяет яркость по значению RGB, вычисляя взвешенную сумму компонентов. Веса определяются рекомендацией «ITU-R Recommendation Rec. 709». За дополнительной информацией обращайтесь к статье «luma» в англоязычной Википедии[1].

Затем вычисляется квадрат градиента (чтобы избежать вычисления квадратного корня) и сохраняется в переменной `g`. Если значение `g` больше величины порогового значения `EdgeThreshold`, считается, что пиксель лежит на границе, и фрагменту присваивается белый цвет. В противном случае фрагменту присваивается черный цвет.

И еще...

Оператор Собеля дает слишком приблизительный результат и слишком чувствителен к частым, резким изменениям яркости. Быстрый поиск в Википедии дает еще множество других приемов выделения границ, обеспечивающих более высокую точность. Компенсировать частые, резкие изменения яркости можно также добавлением прохода «размытия» между отображением и проходом выделения границ. Проход «размытия» сгладит частые, резкие флуктуации и поможет повысить точность результатов, получаемых в проходе выделения границ.

---

[1] **https://en.wikipedia.org/wiki/Luma_(video)**. К сожалению, аналогичной статьи на русском языке нет, однако от себя могу порекомендовать ознакомительную статью: **http://webmascon.com/topics/colors/10d.asp**. – *Прим. перев.*

Приемы оптимизации

Прием, представленный в этом рецепте, требует выполнить восемь операций извлечения пикселей из текстуры. Эти операции могут привести к существенному замедлению работы шейдера, соответственно, уменьшение их числа поможет увеличить скорость работы. В главе 24 книги «GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics», вышедшей под редакцией Рандима Фернандо (Randima Fernando) (Addison-Wesley Professional, 2004), обсуждаются различные пути уменьшения числа операций доступа к текстуре в фильтрах за счет использования так называемых «вспомогательных» текстур.

См. также

- Статья Д. Зиу (D. Ziou) и С. Таббоне (S. Tabbone) (1998) «Edge detection techniques: An overview», International Journal of Computer Vision, Vol 24, Issue 3.
- Статья «Frei-Chen edge detector»: **http://rastergrid.com/blog/2011/01/freichen-edge-detector/**[1].
- Рецепт «Использование подпрограмм для выбора функциональности в шейдере» в главе 2 «Основы шейдеров GLSL».
- Рецепт «Отображение в текстуру» в главе 4 «Текстуры».
- Рецепт «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL».

Применение фильтра размытия по Гауссу

Фильтр размытия может пригодиться в самых разных ситуациях, где требуется снизить уровень шума в изображениях. Как отмечалось в предыдущем рецепте, применение фильтра размытия перед проходом выделения границ может повысить качество результатов за счет устранения влияния частых и резких флуктуаций в изображении. В основе любого фильтра размытия лежит идея смешивания цветов соседних пикселей с использованием взвешенных сумм. Веса обычно уменьшаются с увеличением расстояния от целевого пикселя (в двухмерном экранном пространстве), соответственно, более далекие пиксели оказывают меньшее влияние на цвет целевого пикселя.

Для взвешивания соседних пикселей реализация **размытия по Гауссу (Gaussian blur)** использует двухмерную гауссову функцию:

$$G(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}.$$

Член σ^2 в уравнении – это **дисперсия** (квадрат стандартного отклонения) гауссова распределения, определяющая ширину гауссовой кривой. Максимум гаус-

[1] От себя могу порекомендовать статьи на русском языке: «Алгоритмы выделения контуров изображений» **(http://habrahabr.ru/post/114452/)** и «Точное выделение контуров на изображениях» **(http://habrahabr.ru/post/128803/)**. – *Прим. перев.*

совой функции находится в точке с координатами (0,0) и соответствует позиции целевого пикселя, а с увеличением любой из координат – X или Y – ее значение уменьшается. На рис. 5.3 приводится график двухмерной гауссовой функции со значением дисперсии σ^2, равным 4.0.

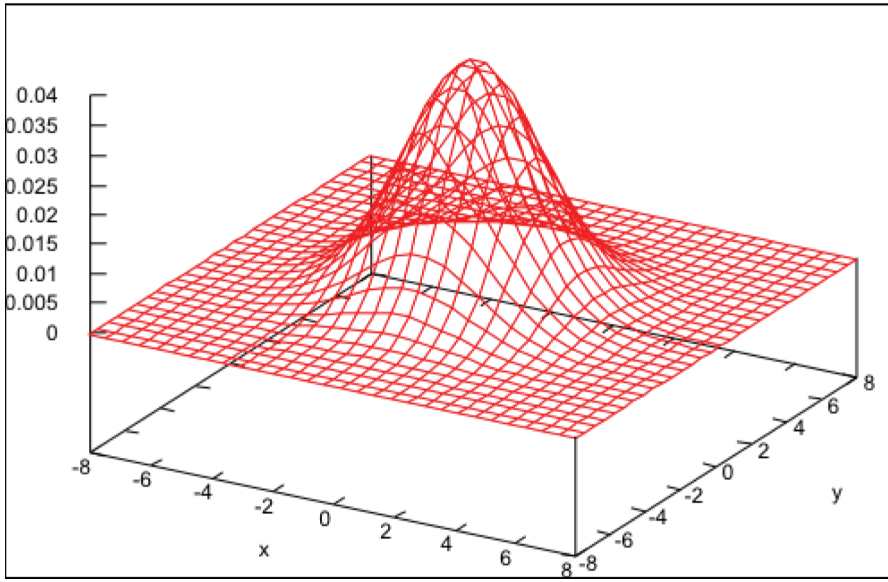

Рис. 5.3 ❖ График двухмерной гауссовой функции со значением дисперсии 4.0

На рис. 5.4 показан фрагмент изображения до (слева) и после (справа) выполнения операции размытия по Гауссу.

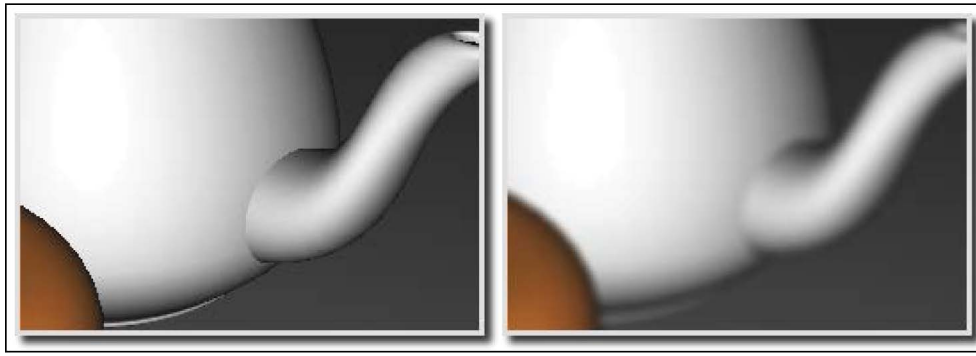

Рис. 5.4 ❖ Фрагмент изображения до (слева) и после (справа) размытия по Гауссу

Чтобы выполнить размытие по Гауссу, необходимо для каждого пикселя вычислить сумму значений всех пикселей в изображении, взвешенную значениями гауссовой функции для данного пикселя (где x и y являются координатами каждого пикселя в системе координат с центром в целевом пикселе). В результате получится новое значение пикселя. Однако в этом алгоритме имеются две проблемы:

- он имеет сложность $O(n^2)$ (где n – число пикселей в изображении), соответственно, производительность его будет слишком низкой;
- сумма весов должна быть равна единице, чтобы избежать изменения общей яркости изображения.

Так как в случае с графикой гауссова функция применяется к ограниченному пространству и в учет принимаются не все значения функции (в бесконечной области определения), сумма весов определенно не будет равна единице.

Решить обе эти проблемы можно, ограничив область размытия и нормализовав значения гауссовой функции. В данном примере будет использоваться фильтр размытия по Гауссу размером 9×9. То есть учитываться будет только 81 пиксель, находящийся по соседству с целевым.

В таком случае внутри фрагментного шейдера потребуется выполнить операцию обращения к текстуре 81 раз. Учитывая, что шейдер вызывается для каждого пикселя, при таком подходе общее число обращений к текстуре размером 800×600 составит $800 \cdot 600 \cdot 81 = 38\,880\,000$. Огромный объем работы, согласны? Но все не так плохо, как кажется. Есть возможность существенно уменьшить число обращений к текстуре, выполнив размытие по Гауссу в два прохода.

Фактически двухмерная гауссова функция является произведением двух одномерных гауссовых функций

$$G(x, y) = G(x)\ G(y),$$

где одномерная гауссова функция вычисляется по формуле

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}.$$

То есть если за C_{ij} принять цвет пикселя с координатами (i, j), необходимая нам сумма вычисляется по формуле

$$G_{lm} \leftarrow \sum_{i=-4}^{4} \sum_{j=-4}^{4} G(i,j) C_{l+im+j}.$$

Ее можно переписать, учитывая тот факт, что двухмерная гауссова функция является произведением двух одномерных гауссовых функций:

$$G_{lm} \leftarrow \sum_{i=-4}^{4} G(i) \sum_{j=-4}^{4} G(j) C_{l+im+j}.$$

Отсюда следует, что размытие по Гауссу можно вычислить в два прохода. В первом проходе можно вычислить сумму по j (вертикальную сумму) и сохранить результаты во временной текстуре. Во втором проходе можно вычислить сумму по

i (горизонтальную сумму), используя результаты, полученные на предыдущем проходе.

Прежде чем перейти к программному коду, следует сделать еще одно важное замечание. Как уже отмечалось выше, сумма весов должна быть равна единице, чтобы получилось истинное взвешенное среднее. Поэтому веса необходимо нормализовать, как показано ниже:

$$G_{lm} \leftarrow \sum_{i=-4}^{4} \frac{G(i)}{k} \sum_{j=-4}^{4} \frac{G(j)}{k} C_{l+im+j},$$

где k – это просто сумма исходных весов.

$$k = \sum_{i=-4}^{4} G(i).$$

Уф-ф! Мы уменьшили исходную сложность алгоритма $O(n^2)$ до $O(n)$. А теперь перейдем к коду.

Размытие по Гауссу в этом рецепте будет реализовано в виде трех проходов и с использованием двух текстур. В первом проходе будет выполнено отображение всей сцены в текстуру. Затем во втором проходе будет выполнено первое (вертикальное) суммирование по текстуре, полученной после первого прохода, с сохранением результатов во второй текстуре. Наконец, в третьем проходе будет выполнено горизонтальное суммирование по текстуре, полученной после второго прохода, а результаты переданы в буфер кадра по умолчанию.

Подготовка

Подготовьте два объекта буферов кадра (см. рецепт «Отображение в текстуру» в главе 4 «Текстуры») и две соответствующие текстуры. Первый объект FBO должен иметь буфер глубины, потому что будет использоваться в первом проходе. Второй объект FBO можно создать без буфера глубины, так как во втором и третьем проходах будет отображаться единственный прямоугольник, чтобы обеспечить запуск фрагментного шейдера для каждого пикселя.

Как и в предыдущем рецепте, для выбора функциональности на том или ином проходе будет использоваться подпрограмма. В основной программе следует также инициализировать следующие uniform-переменные:

- `Width`: ширина экрана (окна) в пикселях;
- `Height`: высота экрана (окна) в пикселях;
- `Weight[]`: массив нормализованных гауссовых весов;
- `Texture0`: ссылка на текстурный слот `0`;
- `PixOffset[]`: массив смещений от целевого пикселя.

Как это делается...

Чтобы создать шейдерную программу, реализующую размытие по Гауссу, нужно выполнить следующие шаги:

1. Добавить тот же вершинный шейдер, что был показан в предыдущем рецепте «Применение фильтра выделения границ».

2. Добавить фрагментный шейдер:

```
in vec3 Position; // координаты вершины
in vec3 Normal;   // вектор нормали к вершине
layout(binding=0) uniform sampler2D Texture0;

subroutine vec4 RenderPassType();
subroutine uniform RenderPassType RenderPass;

// Другие uniform-переменные для модели затенения по Фонгу
// можно объявить здесь...
layout( location = 0 ) out vec4 FragColor;

uniform int PixOffset[5] = int[](0,1,2,3,4);
uniform float Weight[5];

vec3 phongModel( vec3 pos, vec3 norm )
{
  // Реализация модели затенения по Фонгу
}

subroutine (RenderPassType)
vec4 pass1()
{
  return vec4(phongModel( Position, Normal ),1.0);
}

subroutine( RenderPassType )
vec4 pass2()
{
  ivec2 pix = ivec2(gl_FragCoord.xy);
  vec4 sum = texelFetch(Texture0, pix, 0) * Weight[0];
  for( int i = 1; i < 5; i++ )
  {
    sum += texelFetchOffset( Texture0, pix, 0,
                ivec2(0,PixOffset[i])) * Weight[i];
    sum += texelFetchOffset( Texture0, pix, 0,
                ivec2(0,-PixOffset[i])) * Weight[i];
  }
  return sum;
}

subroutine( RenderPassType )
vec4 pass3()
{
  ivec2 pix = ivec2(gl_FragCoord.xy);
  vec4 sum = texelFetch(Texture0, pix, 0) * Weight[0];
  for( int i = 1; i < 5; i++ )
  {
    sum += texelFetchOffset( Texture0, pix, 0,
                ivec2(PixOffset[i],0)) * Weight[i];
    sum += texelFetchOffset( Texture0, pix, 0,
                ivec2(-PixOffset[i],0)) * Weight[i];
  }
```

```
  return sum;
}

void main()
{
  // Следующая инструкция будет вызывать pass1(), pass2() или pass3()
  FragColor = RenderPass();
}
```

3. В основном приложении вычислить гауссовы веса для смещений в uniform-переменной `PixOffset` и сохранить результаты в массиве `Weight`. Реализовать это можно, как показано ниже:

```
char uniName[20];
float weights[5], sum, sigma2 = 4.0f;

// Вычислить и суммировать веса
weights[0] = gauss(0,sigma2); // одномерная гауссова функция
sum = weights[0];
for( int i = 1; i < 5; i++ ) {
  weights[i] = gauss(i, sigma2);
  sum += 2 * weights[i];
}

// Нормализовать веса и сохранить в uniform-переменной
for( int i = 0; i < 5; i++ ) {
  snprintf(uniName, 20, "Weight[%d]", i);
  prog.setUniform(uniName, weights[i] / sum);
}
```

В первом проходе в функции отображения нужно выполнить следующие шаги:

1. Выбрать объект буфера кадра для отображения, включить проверку глубины и очистить буферы цвета/глубины.
2. Выбрать функцию `pass1` подпрограммы.
3. Нарисовать сцену.

Во втором проходе в функции отображения нужно выполнить следующие шаги:

1. Выбрать объект промежуточного буфера кадра, выключить проверку глубины и очистить буферы цвета/глубины.
2. Выбрать функцию `pass2` подпрограммы.
3. Определить матрицы модели, вида и проекции как единичные матрицы.
4. Связать текстуру, полученную в первом проходе, с текстурным слотом `0`.
5. Нарисовать полноэкранный квадрат.

В третьем проходе в функции отображения нужно выполнить следующие шаги:

1. Вернуться к использованию буфера кадра по умолчанию и очистить буфер цвета.
2. Выбрать функцию `pass3` подпрограммы.
3. Связать текстуру, полученную во втором проходе, с текстурным слотом `0`.
4. Нарисовать полноэкранный квадрат.

Как это работает...

В реализации вычисления гауссовых весов (см. фрагмент кода в п. 3) функция `gauss` вычисляет одномерную гауссову функцию, принимая в первом аргументе значение координаты X, а во втором – дисперсию (квадрат стандартного отклонения). Обратите внимание, что здесь достаточно вычислить лишь положительные смещения, потому что гауссова функция симметрична относительно начала координат. Так как здесь вычисляются лишь положительные значения, следует с особым вниманием отнестись к процедуре суммирования весов. В данном случае мы удваиваем все ненулевые значения, потому что они используются дважды (для положительных и отрицательных смещений).

В первом проходе (функция `pass1` подпрограммы) производится отображение сцены в текстуру с применением модели затенения по Фонгу (см. рецепт «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL»).

Во втором проходе (функция `pass2` подпрограммы) вычисляется взвешенная вертикальная сумма, и результат сохраняется в еще одной текстуре. Здесь выполняется чтение пикселей в вертикальном направлении из текстуры, созданной в первом проходе, по смещениям, заданным в массиве `PixOffset`. Суммирование взвешенных значений выполняется с использованием весов из массива `Weight`, причем сразу в двух направлениях – вверх и вниз – на расстояние до четырех пикселей.

Третий проход (функция `pass3` подпрограммы) очень похож на второй. Здесь накапливаются взвешенные суммы по горизонтали для текстуры, полученной после второго прохода. Благодаря этому суммы, полученные во втором проходе, объединяются в общие взвешенные суммы, как описывалось выше. В результате создается сумма по области 9×9 пикселей. На этом проходе цвет пикселя попадает в буфер кадра по умолчанию, содержимое которого попадает на экран.

И еще...

Предыдущую реализацию можно оптимизировать, уменьшив число обращений к текстуре наполовину. Если прибегнуть к автоматической линейной интерполяции при обращении к текстуре (при использовании режима `GL_LINEAR` увеличения и уменьшения), за одно обращение можно получать информацию сразу о двух текселях! Подробнее этот прием описывается в статье Даниеля Ракуса (Daniel Rákos): **http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/**.

Разумеется, описанный здесь прием можно приспособить для реализации размытия по более широкому диапазону текселей, увеличив размеры массивов `Weight` и `PixOffset` и пересчитав веса, и/или можно было бы использовать иные значения дисперсии `sigma2`, изменив тем самым форму гауссовой кривой.

См. также

- Статью о двусторонней фильтрации: **http://people.csail.mit.edu/sparis/bf_course/**.

- Рецепт «Отображение в текстуру» в главе 4 «Текстуры».
- Рецепт «Применение фильтра выделения границ».
- Рецепт «Использование подпрограмм для выбора функциональности в шейдере» в главе 2 «Основы шейдеров GLSL».

Преобразование диапазона яркостей HDR с помощью тональной компрессии

При отображении на большинстве устройств вывода (мониторы или телевизоры) поддерживается точность цветопередачи всего по 8 бит на компонент RGB, или 24 бита на пиксель. Соответственно, яркость каждого компонента цвета ограничена градациями от 0 до 255. Внутри OpenGL компоненты цвета представлены вещественными значениями, что обеспечивает более широкий диапазон и более высокую точность цветопередачи, но перед отображением вещественные значения в диапазоне [`0.0, 1.0`] преобразуются в 8-битные значения в диапазоне [`0, 255`].

Реальные сцены, однако, имеют гораздо более широкий диапазон яркостей. Например, источники света, видимые в сцене, или зеркальные блики от них могут в сотни, а то и в тысячи раз превосходить по яркости объекты, освещаемые этими источниками. Имея точность 8 бит на канал или вещественный диапазон [`0.0, -1.0`], невозможно представить такой диапазон яркостей. Если задействовать более широкий диапазон вещественных значений, можно обеспечить более точное внутреннее представление яркости, но в конечном итоге значения все равно придется сжимать до 8-битного диапазона.

Процесс вычисления освещенности/затенения с использованием более широкого динамического диапазона часто называют **расширенный динамический диапазон отображения (High Dynamic Range rendering, HDR rendering)**. Фотографам хорошо знакомо это понятие. Когда фотограф желает охватить более широкий диапазон яркостей, чем это позволяет какая-то одна экспозиция, он/она может сделать несколько снимков с разными экспозициями. Данный прием называют **расширенный динамический диапазон съемки (HDR imaging)**, и он очень близок к понятию расширенного динамического диапазона отображения. Конвейеры постобработки, включающие реализацию приема HDR, в настоящее время считаются неотъемлемой частью любого игрового движка.

Тональная компрессия (tone mapping) – это процесс преобразования тональных значений изображения из более широкого диапазона в более узкий, пригодный для отображения на данном устройстве. В компьютерной графике под тональной компрессией обычно понимают отображение значений из некоторого диапазона произвольной ширины в 8-битный диапазон. Цель – обеспечить отображение светлых и темных частей изображения так, чтобы не допустить полного исчезновения деталей.

Например, присутствие в сцене источника яркого света может привести к тому, что модель затенения начнет возвращать значения яркости больше `1.0`. Если просто передать устройству вывода фрагмент со значением яркости больше 1.0, оно

усечет его до 255, и фрагмент будет выглядеть белым. В результате получится изображение, напоминающее фотографию, полученную с чрезмерно большой экспозицией. Или, если нормализовать яркости до диапазона [0, 255], темные части будут выглядеть слишком темными или вообще окажутся невидимыми. Тональная компрессия поможет нам управлять яркостью источника света, а также обеспечить прорисовку темных областей.

Данное описание приема тональной компрессии и методики расширенного динамического диапазона отображения/съемки весьма поверхностно. За дополнительными подробностями обращайтесь к книге Эрика Рейнхарда (Erick Reinhard) и др. «High Dynamic Range Imaging».

Математическая функция, используемая для преобразования динамического диапазона в более узкий диапазон, называется **оператор тональной компрессии (Tone Mapping Operator, TMO)**. Вообще, существуют две «разновидности» таких операторов – локальные операторы и глобальные операторы. Локальный оператор определяет новое значение для заданного пикселя, используя его текущее значение и, возможно, значения соседних пикселей. Глобальному оператору требуется некоторая дополнительная информация обо всем изображении. Например, одному глобальному оператору может потребоваться средняя яркость по всем пикселям в изображении. Другие глобальные операторы могут использовать гистограмму частот встречаемости значений яркости во всем изображении.

В данном рецепте будет использоваться простой глобальный оператор, описанный в книге Томаса Акенина-Мюллера (Tomas Akenine-Möller) и Эрика Хайнса (Eric Haines) «Real Time Rendering». Этот оператор применяет среднее логарифмическое яркостей всех пикселей в изображении. Среднее логарифмическое определяется как экспонента от среднего логарифмов значений яркостей:

$$\bar{L}_w = \exp\left(\frac{1}{N}\sum_{x,y}\ln(0.001 + L_w(x,y))\right),$$

где $L_w(x, y)$ – яркость пикселя с координатами (x, y). Член 0.0001 используется с целью избежать получения нуля для черных пикселей. Полученное среднее логарифмическое используется затем в операторе тональной компрессии:

$$L(x,y) = \frac{a}{\bar{L}_w} L_w(x,y).$$

Член a в этом уравнении является ключевым. Он действует подобно времени экспозиции в камере. Чаще всего этому члену дают значение в диапазоне от 0.18 до 0.72. Так как этот оператор тональной компрессии сжимает темные и светлые значения слишком сильно, мы будем использовать немного измененное уравнение, сжимающее темные цвета меньше и включающее значение максимальной яркости (L_{white}), которое поможет уменьшить чрезмерную яркость некоторых пикселей:

$$L_d(x,y) = \frac{L(x,y)\left(1 + \frac{L(x,y)}{L_{white}^2}\right)}{1 + L(x,y)}.$$

Этот оператор будет использоваться в данном рецепте. Реализация сначала будет отображать сцену в буфер высокого разрешения, затем вычислять среднее логарифмическое яркости и во втором проходе применять предыдущий оператор.

Однако прежде чем перейти к реализации, необходимо коснуться еще одной детали. Все предыдущие уравнения оперируют яркостью. Имея значение цвета в формате RGB, можно вычислить его яркость, но как после изменения яркости вновь вернуться к формату RGB, соответствующему новой яркости, без изменения **тональности**?

Под тональностью понимается воспринимаемый цвет независимо от яркости. Например, серый и белый – это один и тот же цвет, но с разными уровнями яркости.

Решение заключается в использовании другого пространства цветов. Если преобразовать сцену в пространство цветов, где яркость является отдельным от тональности компонентом, можно будет изменять яркость независимо. Пространство цветов **CIE XYZ** – это то, что нам нужно. В формате CIE XYZ компонент Y описывает яркость цвета, а его тональность можно получить по производным компонентам (x и y). Производное пространство цвета обозначается как **CIE xyY** и в точности соответствует нашим потребностям. Компонент Y содержит яркость, а компоненты x и y содержат информацию о тональности. За счет преобразования исходного изображения в пространство цветов CIE xyY яркость отделяется от тональности, что дает возможность изменять яркость, не изменяя тональности воспринимаемого цвета.

Итак, общий процесс выглядит так: цвет пикселя преобразуется из формата RGB в формат CIE XYZ и затем в формат CIE xyY, далее изменяется его яркость и выполняется обратная последовательность преобразований в формат RGB. Преобразование из формата RGB в формат CIE XYZ (и обратно) можно описать с помощью матрицы (см. ссылку, касающуюся матриц, в разделе «См. также» в конце этого рецепта).

Преобразование XYZ в xyY выполняется так:

$$x = \frac{X}{X+Y+Z}; \quad y = \frac{Y}{X+Y+Z}.$$

Наконец, преобразование xyY обратно в XYZ выполняется по следующим формулам:

$$X = \frac{Y}{y}x; \quad Z = \frac{Y}{y}(1-x-y).$$

На рис. 5.5 показан результат применения данного оператора тональной компрессии. Слева изображена сцена без применения тональной компрессии. В сцену преднамеренно были включены три сильных источника света, чтобы получить очень широкий диапазон яркостей. В этой сцене почти не видно деталей из-за присутствия большого числа значений больше `1.0`, которые просто усекаются до максимальной интенсивности. Справа изображена та же сцена, с теми же источниками света, но обработанная оператором компрессии тональности. Обратите

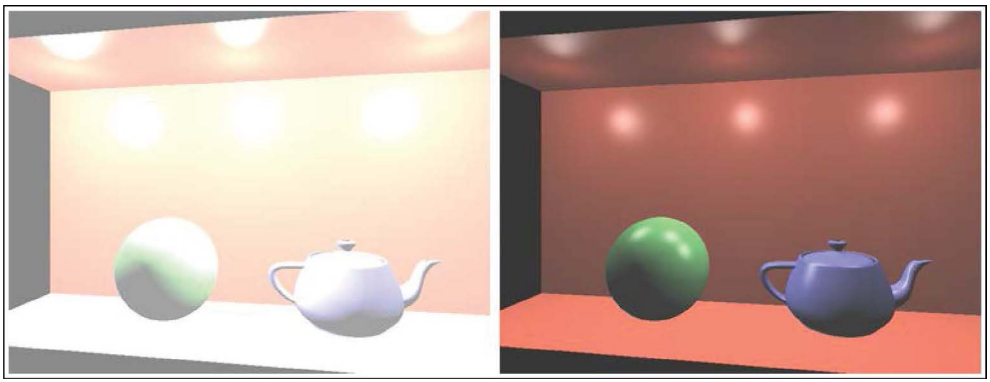

Рис. 5.5 ❖ Сцена до (слева) и после (справа) применения тональной компрессии

внимание, что на шаре и на чайнике были восстановлены правильные блики от источников света.

Подготовка

В данном рецепте выполняется следующая последовательность действий:

1. Отображение сцены в текстуру высокого разрешения.
2. Вычисление среднего логарифмического яркости (в основной программе).
3. Отображение прямоугольника, заполняющего весь экран, чтобы инициировать вызов фрагментного шейдера для каждого пикселя. Фрагментный шейдер читает значение пикселя из текстуры, созданной в шаге 1, применяет оператор тональной компрессии и выводит результаты на экран.

Для этого в основной программе создайте текстуру высокого разрешения (используя формат `GL_RGB32F` или подобный ему), подключите ее к буферу кадра, имеющему буфер глубины. Добавьте фрагментный шейдер с подпрограммой для обработки каждого прохода. От вершинного шейдера требуется только, чтобы он передавал позицию вершины и нормаль к ней в видимых координатах.

Как это делается...

Чтобы реализовать прием тональной компрессии, нужно выполнить следующие шаги:

1. В первом проходе достаточно просто отобразить сцену в текстуру высокого разрешения. Выполнить привязку к буферу кадра с подключенной текстурой и отобразить сцену, как обычно. Применить любой прием вычисления освещенности по вашему выбору.
2. Вычислить среднее логарифмическое яркости пикселей по всей текстуре. Для этого потребуется в основной программе извлечь данные из текстуры и обойти в цикле все пиксели. Данный шаг реализован в основной программе

для простоты, если перенести эти вычисления в шейдер, они наверняка будут выполняться быстрее.

```
GLfloat *texData = new GLfloat[width*height*3];
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, hdrTex);
glGetTexImage(GL_TEXTURE_2D, 0, GL_RGB, GL_FLOAT, texData);
float sum = 0.0f;
int size = width*height;
for( int i = 0; i < size; i++ ) {
  float lum = computeLuminance(
          texData[i*3+0], texData[i*3+1], texData[i*3+2]));
  sum += logf( lum + 0.00001f );
}
delete [] texData;
float logAve = expf( sum / size );
```

3. Присвоить uniform-переменной `AveLum` значение `logAve`. Вернуться к использованию буфера кадра по умолчанию и нарисовать прямоугольник, покрывающий экран. Внутри фрагментного шейдера применить оператор тональной компрессии к значениям из текстуры, полученной в шаге 1.

```
// Извлечь цвет в высоком разрешении из текстуры
vec4 color = texture( HdrTex, TexCoord );

// Преобразовать в формат XYZ
vec3 xyzCol = rgb2xyz * vec3(color);

// Преобразовать в формат xyY
float xyzSum = xyzCol.x + xyzCol.y + xyzCol.z;
vec3 xyYCol = vec3(0.0);
if( xyzSum > 0.0 ) // Избежать деления на нуль
  xyYCol = vec3( xyzCol.x / xyzSum,
                 xyzCol.y / xyzSum, xyzCol.y);

// Применить опретор компрессии тональности
// (xyYCol.z или xyzCol.y)
float L = (Exposure * xyYCol.z) / AveLum;

L = (L * ( 1 + L / (White * White) )) / ( 1 + L );

// Сохранить новую яркость, преобразовав обратно в XYZ
if( xyYCol.y > 0.0 ) {
  xyzCol.x = (L * xyYCol.x) / (xyYCol.y);
  xyzCol.y = L;
  xyzCol.z = (L * (1 - xyYCol.x - xyYCol.y))/xyYCol.y;
}

// Преобразовать обратно в формат RGB и передать в выходной буфер
FragColor = vec4( xyz2rgb * xyzCol, 1.0);
```

Как это работает...

На первом шаге выполняется отображение сцены в HDR-текстуру. На шаге 2 вычисляется среднее логарифмическое яркости пикселей в текстуре (в основном приложении).

На шаге 3 отображается единственный прямоугольник, покрывающий весь экран, чтобы обеспечить выполнение фрагментного шейдера для каждого пикселя. Внутри шейдера из текстуры извлекается значение текущего пикселя, и к нему применяется оператор тональной компрессии. В этих вычислениях используются две «настроечные» переменные. Переменная `Exposure` соответствует члену a в уравнении оператора, а переменная `White` – члену L_{white}. Для обработки изображения в данном примере использовались значения `0.35` и `0.928` соответственно.

И еще...

Тональная компрессия не имеет точного математического обоснования выбора параметров. Часто параметры подбираются экспериментальным путем, пока не будет получено приемлемое изображение.

Мы могли бы улучшить производительность предыдущего примера, реализовав выполнение шага 2 на GPU, в вычислительном шейдере (см. главу 10 «Вычислительные шейдеры») или используя какой-то иной прием. Например, можно было бы записывать логарифмы в текстуру, затем итеративно свернуть весь кадр в текстуру 1×1. Окончательный результат был бы доступен в виде единственного пикселя. Однако благодаря гибкости вычислительного шейдера эту процедуру можно было бы оптимизировать еще больше.

См. также

- Брюс Джастин Линдблюм (Bruce Justin Lindbloom) создал очень полезный веб-ресурс, посвященный теме преобразований пространств цветов. Кроме всего прочего, там можно найти матрицы преобразований формата RGB в формат XYZ: **http://www.brucelindbloom.com/index.html?Eqn_XYZ_to_RGB.html**.
- Рецепт «Отображение в текстуру» в главе 4 «Текстуры».

Эффект размытости на границах ярких участков

Эффект размытости на границах ярких участков (bloom) – это визуальный эффект, создающий ощущение, что яркие участки изображения как бы «наезжают» на более темные. Этот эффект соответствует тому, как камеры и человеческий глаз видят границы с высоким контрастом. Ярко освещенные участки «наезжают» на другие области из-за эффекта, получившего название **диск Эйри** – дифракционный узор, возникающий при прохождении света через равномерно освещённое круглое отверстие.

На рис. 5.6 показан эффект размытости на границах ярких участков, воспроизведенный в анимационном фильме «Elephant's Dream» (© 2006, Blender Foundation/Netherlands Media Art Institute/**www.elephantsdream.org**). Яркий белый свет от источника за дверью «наезжает» на более темные участки изображения.

Чтобы воспроизвести такой эффект, требуется определить, какие части изображения являются достаточно светлыми, выделить их, выполнить размытие и встро-

Рис. 5.6 ❖ Эффект размытости на границах ярких участков в анимационном фильме «Elephant's Dream»

ить их в исходное изображение. Обычно эффект размытости на границах ярких участков связан с приемом отображения расширенного динамического диапазона (HDR), используя который, можно отображать гораздо более широкий диапазон яркостей пикселей (без образования артефактов). Эффект размытости на границах ярких участков получается более точным, когда используется в сочетании с приемом HDR-отображения, благодаря возможности представления более широкого диапазона яркостей.

Несмотря на то что прием HDR-отображения позволяет получить более качественный результат, эффект размытости на границах ярких участков все же можно воспроизвести и с использованием стандартных значений цветов. Результат может выглядеть не так эффектно, но принцип остается тем же.

Следующий пример реализует эффект размытости на границах ярких участков за пять проходов, выполняющих четыре основных шага:

1. В первом проходе выполняется отображение сцены в HDR-текстуру.
2. Во втором проходе извлекаются части изображения, имеющие уровень яркости выше некоторого порогового значения. Мы будем называть его **проходом сглаживания яркости (bright-pass filter)**. При применении этого фильтра также будет выполняться сохранение данных в буфер с более низким разрешением, чтобы получить дополнительное размытие при обратном чтении из этого буфера.
3. В третьем и четвертом проходах к ярким участкам применяется размытие по Гауссу (см. рецепт «Применение фильтра размытия по Гауссу» выше).
4. В пятом проходе применяется тональная компрессия, и ее результаты добавляются в изображение, полученное в проходе выделения ярких участков.

Описанный процесс изображен на рис. 5.7. Слева вверху показана сцена, отображенная в буфер HDR, в которой часть деталей исчезает при выводе на устройство отображения. Проход сглаживания яркости производит меньшее по размеру изображение (от четверти до одной восьмой части от размеров исходного изображения), в котором остаются только пиксели, соответствующие участкам, яркость которых превышает пороговое значение. Пиксели показаны белым цветом, потому что в данном примере их значения больше единицы. К этому уменьшенному изображению применяется двухпроходное размытие по Гауссу, а к оригинальному – тональная компрессия. Окончательное изображение формируется посредством объединения изображений, прошедших тональную компрессию и размытие по Гауссу. Накладывая его как текстуру, мы используем линейный фильтр, чтобы

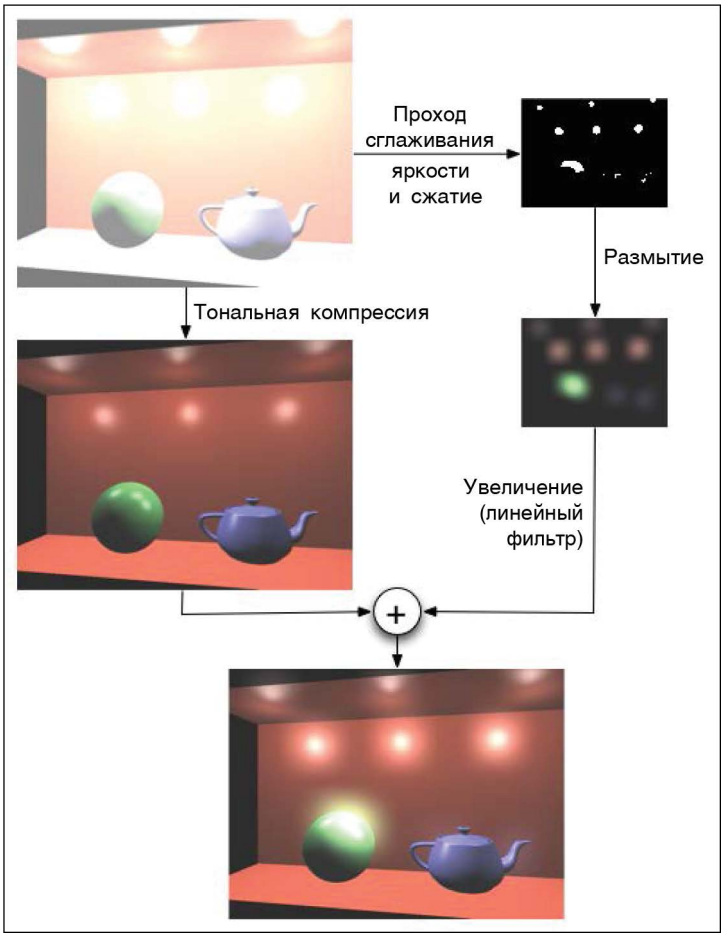

Рис. 5.7 ❖ Процесс воспроизведения эффекта размытости на границах ярких участков

получить дополнительное размытие. Окончательный результат показан внизу. Обратите внимание, как яркие блики на сфере наезжают на стену сзади.

Подготовка

Для реализации данного рецепта потребуются два объекта буфера кадра, каждый со своей текстурой. Первый будет использоваться для HDR-отображения оригинальной сцены, а второй – для двухпроходного размытия по Гауссу. Внутри фрагментного шейдера оригинальное изображение доступно через переменную `HdrTex`, а текстура, полученная в результате размытия по Гауссу, – через переменную `BlurTex`.

Значение uniform-переменной `LumThresh` определяет порог, используемый во втором проходе. Любые пиксели со значением выше этого порога будут извлечены и подвергнуты размытию в последующих проходах.

Вершинный шейдер просто передает координаты вершины и нормаль к ней в системе видимых координат.

Как это делается...

Чтобы воссоздать эффект размытости на границах ярких участков, нужно выполнить следующие шаги:

1. В первом проходе отобразить сцену в буфер кадра высокого разрешения, связанный с текстурой.
2. Во втором проходе переключиться на буфер кадра с текстурой высокого разрешения, меньшей, чем размер полного изображения. В примере кода используется текстура, размер которой в восемь раз меньше оригинального изображения. Нарисовать квадрат во весь экран, чтобы обеспечить вызов фрагментного шейдера для каждого пикселя, который выберет из текстуры высокого разрешения только пиксели, значения которых превышают `LumThresh`, и запишет их во вторую текстуру. Остальные пиксели должны остаться черными.

```
vec4 val = texture(HdrTex, TexCoord);
if( luminance(val.rgb) > LumThresh )
  FragColor = val;
else
  FragColor = vec4(0.0);
```

3. В третьем и четвертом проходах применить размытие по Гауссу к текстуре, полученной после второго прохода. Это можно реализовать с применением единственного буфера кадра и двух текстур, переключаясь между ними, читая из одной и записывая в другую. За подробностями обращайтесь к рецепту «Применение фильтра размытия по Гауссу» выше в этой главе.
4. В пятом и заключительном проходе переключиться на линейную фильтрацию из текстуры, полученной после четвертого прохода. Вернуться к буферу кадра по умолчанию (экрану). Применить оператор тональной компрессии, описанный в рецепте «Преобразование диапазона яркостей HDR

с помощью тональной компрессии», к оригинальной текстуре (HdrTex) и затем объединить ее с текстурой, полученной после шага 3. Линейная фильтрация с увеличением должна обеспечить дополнительное размытие.

```
// Извлечь цвет из текстуры высокого разрешения
vec4 color = texture( HdrTex, TexCoord );

// Применить тональную компрессию, сохранить разультат в toneMapColor
...
///////// Объединить текстуры //////////
vec4 blurTex = texture(BlurTex1, TexCoord);

FragColor = toneMapColor + blurTex;
```

Как это работает...

Из-за нехватки места в книге я не могу привести код фрагментного шейдера целиком. Однако вы найдете этот код в репозитории GitHub. Во фрагментном шейдере реализованы пять функций подпрограммы, по одной для каждого прохода. Первый проход отображает сцену, как обычно, в HDR-текстуру. В этом проходе активным является объект буфера кадра, связанного с текстурой в переменной HdrTex, поэтому вывод фрагментного шейдера будет записываться непосредственно в эту текстуру.

Во втором проходе выполняются чтение пикселей из HdrTex и вывод только тех из них, значения яркости которых превышают пороговое значение LumThresh. Для пикселей со значением яркости ниже LumThresh записывается значение (0,0,0,0). Вывод сохраняется во втором буфере кадра, содержащем меньшую по размерам текстуру (в восемь раз меньше оригинала).

В третьем и четвертом проходах выполняется размытие по Гауссу (см. рецепт «Применение фильтра размытия по Гауссу» выше в этой главе). В этих проходах требуется переключаться между BlurTex1 и BlurTex2, поэтому необходимо сделать все, чтобы обеспечить своевременное переключение текстур в буфере кадра.

В пятом проходе выполняются возврат к буферу кадра по умолчанию и объединение текстур HdrTex и BlurTex1. BlurTex1 содержит результат размытия, выполненного в шаге 4, а HdrTex содержит оригинальную текстуру со сценой. Далее к текстуре HdrTex применяется тональная компрессия, и затем к ней добавляется содержимое BlurTex1. При извлечении пикселей из BlurTex1 выполняется линейная фильтрация, дающая дополнительное размытие.

И еще...

Обратите внимание, что здесь оператор тональной компрессии применяется к оригинальному изображению, но не применяется к размытому. Тональную компрессию можно было бы применить и к размытому изображению, но часто в этом нет необходимости.

Следует также помнить, что эффект размытости на границах ярких участков может выглядеть негармонично при злоупотреблении им. Хорошего, как говорится, помаленьку.

См. также

- Статью Франческо Каруцци (Francesco Caruzzi) «HDR meets Black&White 2» в сборнике «Shader X6».
- Рецепт «Отображение в текстуру» в главе 4 «Текстуры».
- Рецепт «Применение фильтра выделения границ».
- Рецепт «Использование подпрограмм для выбора функциональности в шейдере» в главе 2 «Основы шейдеров GLSL».

Повышение качества изображения с помощью гамма-коррекции

Для многих книг, посвященных OpenGL и трехмерной компьютерной графике, характерно замалчивание темы гамма-коррекции. Фрагментный шейдер вычисляет освещенность фрагментов и отправляет результаты непосредственно в выходной буфер, без каких-либо дополнительных изменений. Однако иногда фактические результаты могут не совпадать с ожидаемыми. Такое возможно, например, из-за того, что компьютерные мониторы (и старые, с лучевой трубкой, и новые, жидкокристаллические) имеют нелинейные характеристики отображения светимости пикселей. Например, серое значение `0.5` без гамма-коррекции может не выглядеть как наполовину менее яркое, чем значение `1.0`. Оно может казаться темнее.

Нижняя кривая на рис. 5.8 соответствует характеристике типичного монитора (гамма-фактор равен 2.2). Ось X на графике обозначает яркость, а ось Y – воспринимаемую яркость. Пунктирная линия представляет линейную зависимость значений яркости. Верхняя кривая – результат применения гамма-коррекции к линейным значениям. Нижняя кривая – характеристика типичного монитора. Серое значение `0.5` могло бы выглядеть как значение `0.218` на экране с такой характеристикой.

Нелинейную характеристику типичного монитора обычно можно смоделировать с использованием простой степенной функции. Воспринимаемая яркость пикселя (P) пропорциональна вычисленной яркости пикселя (I), возведенной в степень, показатель которой часто называют «гамма-фактором»:

$$P = I^{\gamma}.$$

В зависимости от технических характеристик устройства значение гамма-фактора обычно находится в диапазоне между `2.0` и `2.4`. Иногда для калибровки монитора требуется более точно определять это значение.

Для компенсации нелинейности можно применить гамма-коррекцию перед передачей результатов в выходной буфер кадра. Для этого требуется возвести значение каждого пикселя в степень, компенсирующую нелинейность характеристик монитора. Поставленную задачу можно решить путем возведения в степень $1/\gamma$ линейных значений:

$$I = (I^{\frac{1}{\gamma}})^{\gamma}.$$

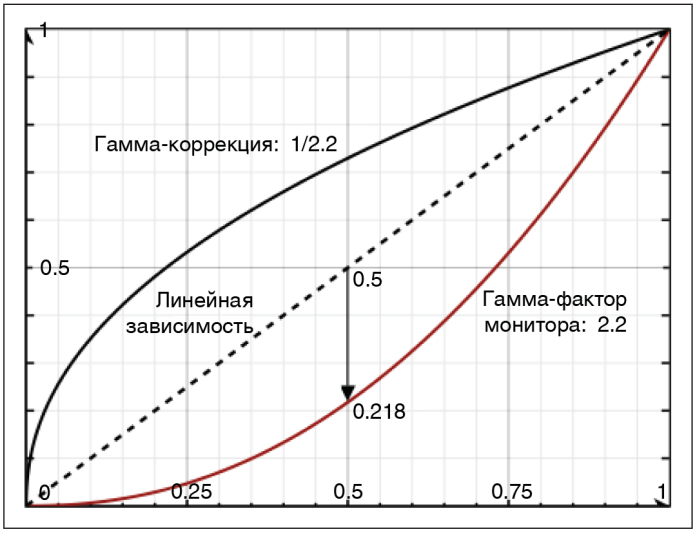

Рис. 5.8 ❖ Гамма-коррекция нелинейной характеристики монитора

При отображении, вычисляя освещенность фрагментов, можно игнорировать нелинейность характеристик монитора. Иногда это называют «работой в линейном пространстве». Когда окончательный результат будут готов к записи в выходной буфер кадра, можно применить гамма-коррекцию, возведя значение пикселя в степень $1/\gamma$ непосредственно перед записью. Это – важный шаг, который поможет улучшить качество отображения результатов.

Например, взгляните на изображения на рис. 5.9. Изображение слева получено без гамма-коррекции. После вычисления освещенности результат сразу отправлялся в буфер кадра. Изображение справа получено уже с гамма-коррекцией.

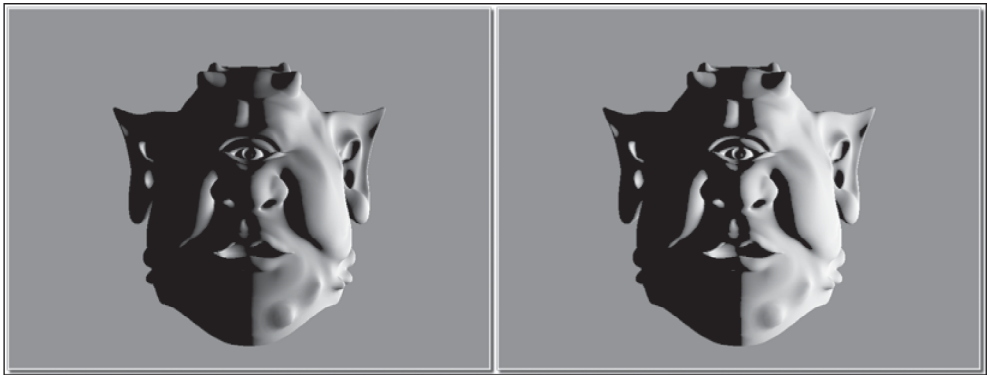

Рис. 5.9 ❖ Одно и то же изображение, полученное без гамма-коррекции (слева) и с гамма-коррекцией (справа)

Нетрудно заметить, что изображение слева выглядит темнее, чем изображение справа. Однако гораздо важнее различия в переходах между темными и светлыми участками. На изображении справа линия терминатора выглядит четче, и вместе с тем переходы между светлыми и темными участками не такие резкие.

Применение гамма-коррекции – это один из важнейших приемов, способный значительно повышать качество результатов модели вычисления освещенности.

Как это делается...

Добавить поддержку гамма-коррекции в приложение совсем несложно, достаточно лишь выполнить следующие шаги:

1. Определить uniform-переменную `Gamma` и присвоить ей значение, соответствующее устройству.
2. Добавить следующий (или подобный) код во фрагментный шейдер:

```
vec3 color = lightingModel( ... );
FragColor = vec4( pow( color, vec3(1.0/Gamma) ), 1.0 );
```

3. Если шейдер работает с данными, извлекаемыми из текстуры, необходимо убедиться, что текстура не подвергалась гамма-коррекции, чтобы не применить коррекцию дважды (см. раздел «И еще..» в конце этого рецепта).

Как это работает...

Цвет определяется в соответствии с моделью вычисления освещенности/затенения и сохраняется в переменной `color`. Предполагается, что цвет вычисляется в «линейном пространстве» – характеристики монитора никак не учитываются в процессе вычислений (здесь также предполагается, что мы не имеем текстуры, уже прошедшей гамма-коррекцию).

Чтобы применить коррекцию во фрагментном шейдере, следует возвести значение цвета пикселя в степень `1.0 / Gamma` и сохранить результат в выходной переменной `FragColor`. Конечно, обратную величину к значению `Gamma` можно вычислить за пределами фрагментного шейдера, чтобы уменьшить объем вычислений на одну операцию деления.

В данном рецепте гамма-коррекция не применяется к значению альфа-канала, потому что обычно это нежелательно.

И еще...

В общем случае поддержка гамма-коррекции – хорошая идея, но иногда требуется проявлять дополнительную осторожность, чтобы не подвергнуть коррекции уже скорректированные данные. Например, текстурами могут быть фотографии или изображения, созданные с помощью других приложений, где они уже были подвергнуты гамма-коррекции перед сохранением в файлы. Соответственно, при наложении текстуры в нашем приложении мы рискуем повторно применить к ней гамма-коррекцию. Чтобы этого не произошло, нужно тщательно «декодировать» данные из текстуры, выполнив возведение в степень *gamma* перед наложением текстуры и расчетами освещенности.

Подробное обсуждение этой и других проблем, связанных с гамма-коррекцией, можно найти в главе 24 «The Importance of Being Linear» книги «GPU Gems 3», вышедшей под редакцией Хьюберта Нгуена (Hubert Nguyen) (Addison-Wesley Professional, 2007), которую я настоятельно рекомендую прочитать.

Сглаживание множественной выборкой

Сглаживание (anti-aliasing) – это прием устранения или уменьшения эффекта «зубчатости», появляющегося при попытке отобразить изображения с высоким разрешением на устройстве с низким разрешением. При отображении графики в реальном масштабе времени часто возникает визуальный эффект в виде зубчатых краев полигонов или искажения пестрых текстур.

На рис. 5.10 показан пример проявления эффекта зубчатости по краю объекта. Слева видно, что край имеет зубчатость. Это происходит из-за того, что каждый пиксель может находиться либо внутри полигона, либо за его пределами. Если пиксель находится внутри полигона, ему присваивается один цвет, если за его пределами – другой. Но так бывает не всегда. Некоторые пиксели могут находиться непосредственно на границе полигона. Часть области экрана, занимаемая пикселем, фактически находится внутри полигона, а другая часть – снаружи. Улучшить результат можно, если цвет для таких пикселей вычислять, исходя из доли занимаемой ими площади, которая находится внутри полигона. Результатом подобных вычислений может быть смешивание цвета полигона и цвета за его границами в соответствующей пропорции. Вам может показаться, что это должна быть довольно дорогостоящая операция. Возможно, вы и правы, однако результаты можно аппроксимировать по нескольким **субпикселям (samples)**.

Прием сглаживания множественной выборкой (multisample anti-aliasing) заключается в том, чтобы для каждого пикселя определить цвет его субпикселей и, комбинируя их, определить цвет всего пикселя. Субпиксели находятся в разных точках области, занимаемой данным пикселем. В большинстве случаев субпиксели будут находиться внутри полигона; но для пикселей, расположенных поблизости от границы полигона, некоторые субпиксели окажутся снаружи. Обычно фрагментный шейдер вызывается для каждого пикселя только один раз. Например, для 4-кратного сглаживания растеризация выполняется четырежды. Фрагментный шейдер вызывается один раз для каждого пикселя, а его результат интерполируется в зависимости от того, сколько из четырех субпикселей окажется внутри полигона.

На рис. 5.10 справа показан результат применения приема сглаживания множественной выборкой. Врезанное изображение показывает увеличенный фрагмент края тора. Слева изображение тора создано без применения приема сглаживания множественной выборкой (Multisample Anti-Aliasing, MSAA). Справа изображение того же тора создано уже с применением сглаживания.

Поддержка технологии MSAA в OpenGL реализована уже давно, и ее использование не вызывает никаких трудностей – достаточно просто включить или вы-

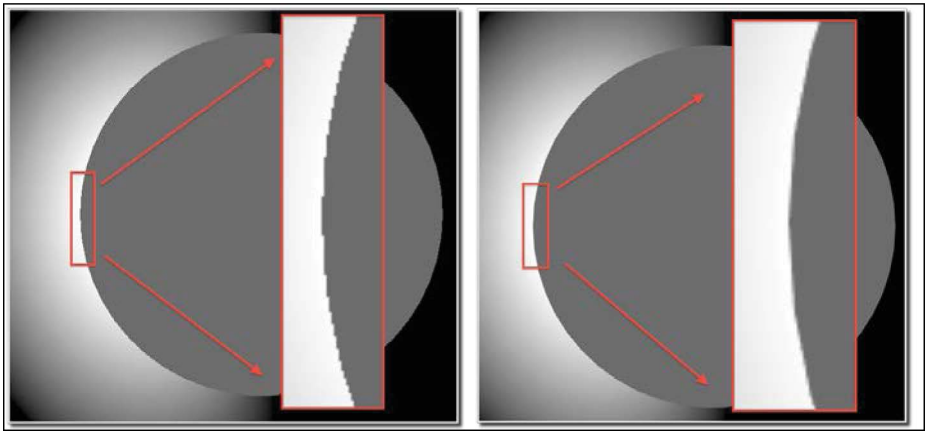

Рис. 5.10 ❖ Изображение тора,
созданное без сглаживания (слева) и со сглаживанием (справа)

ключить режим сглаживания. Для сглаживания используются дополнительные буферы, где во время обработки сохраняются субпиксели, на основе которых определяется окончательный цвет фрагмента. Почти вся работа выполняется автоматически, и программисту доступно не так много для изменения результата. В конце этого рецепта мы познакомимся с квалификаторами интерполяции, которые могут оказывать влияние на результаты.

В данном рецепте будет показан программный код, необходимый для включения сглаживания множественной выборкой.

Подготовка

К сожалению, точный порядок включения сглаживания множественной выборкой зависит от API оконной системы. В данном примере демонстрируется, как это делается с помощью GLFW. При использовании GLUT и других API поддержки OpenGL выполняются аналогичные операции.

Как это делается...

Чтобы создать буферы для сглаживания множественной выборкой, нужно выполнить следующие шаги:

1. В процессе создания окна приложения нужно выбрать контекст OpenGL, поддерживающий MSAA. Ниже показано, как это сделать с помощью GLFW:

```
glfwWindowHint(GLFW_SAMPLES, 8);
... // Другие настройки
window = glfwCreateWindow( WIN_WIDTH, WIN_HEIGHT,
                           "Window title", NULL, NULL );
```

2. Определить доступность буферов для сглаживания и количество субпикселей на пиксель можно следующим образом:

```
GLint bufs, samples;
glGetIntegerv(GL_SAMPLE_BUFFERS, &bufs);
glGetIntegerv(GL_SAMPLES, &samples);
printf("MSAA: buffers = %d samples = %d\n", bufs, samples);
```

3. Включить сглаживание множественной выборкой можно так:

```
glEnable(GL_MULTISAMPLE);
```

4. А выключить – так:

```
glDisable(GL_MULTISAMPLE);
```

Как это работает...

Как только что отмечалось, порядок создания контекста OpenGL с поддержкой сглаживания множественной выборкой зависит от API, используемого для взаимодействия с оконной системой. Предыдущий пример демонстрирует, как это сделать с помощью GLFW. После создания контекста OpenGL сглаживание включается простым вызовом функции `glEnable`.

В следующем разделе я расскажу о некоторых проблемах, связанных с интерполяцией значений переменных в шейдере при включенной поддержке сглаживания.

И еще...

В языке GLSL имеются два квалификатора управления интерполяцией, которые позволяют программисту настраивать некоторые аспекты множественной выборки: `sample` и `centroid`.

Но прежде чем перейти к изучению квалификаторов `sample` и `centroid`, обратимся к теории. Давайте посмотрим, как обрабатываются края полигона при выключенном сглаживании. Принадлежность фрагмента полигону зависит от того, где находится центр пикселя – внутри полигона или за его пределами. Если центр находится внутри полигона, пиксель окрашивается цветом полигона, в противном случае – цветом окружающей области. Этот алгоритм изображен на рис. 5.11. На нем показаны пиксели рядом с границей полигона, как они отображаются без сглаживания MSAA. Прямая линия представляет границу полигона. Серые пиксели считаются принадлежащими полигону, а белые – находящимися за его границами. Точки представляют центры пикселей.

Значения для интерполируемых переменных (входных переменных фрагментного шейдера) определяются по центрам фрагментов, которые всегда находятся внутри полигона.

При включенном сглаживании множественной выборкой фрагмент делится на несколько субпикселей. Если какой-то из субпикселей окажется внутри полигона, шейдер будет вызван как минимум один раз для данного пикселя (необязательно для каждого субпикселя). Для наглядности на рис. 5.12 показаны пиксели, расположенные рядом с границей полигона. Точки представляют субпиксели. Темные субпиксели находятся внутри полигона, а белые – за его пределами. Если какой-то субпиксель окажется внутри полигона, для данного пикселя будет вызван фраг-

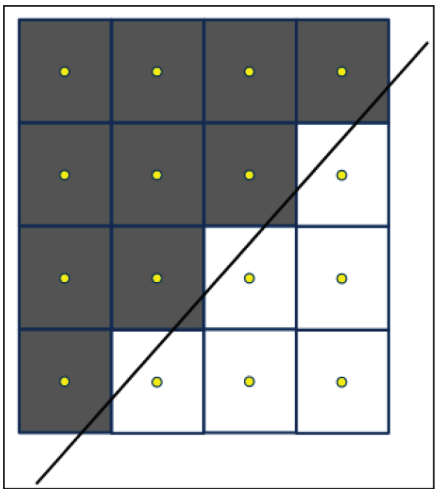

Рис. 5.11 ❖ Отображение пикселей без сглаживания MSAA

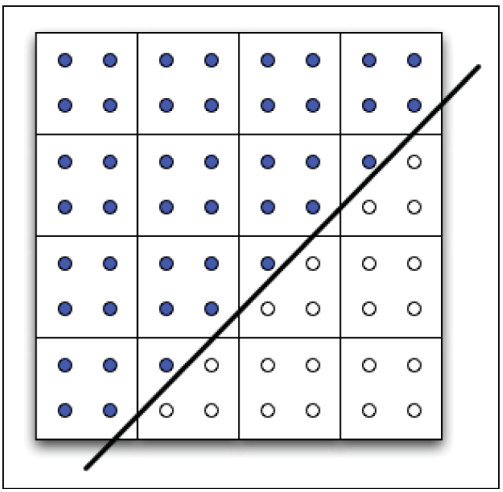

Рис. 5.12 ❖ Пиксели рядом с границей полигона

ментный шейдер (обычно только один раз). Обратите внимание, что центры некоторых пикселей находятся за пределами полигона. Поэтому при использовании MSAA на краях полигона фрагментный шейдер может вызываться чаще.

Мы достигли важного пункта. Значения выходных переменных фрагментного шейдера обычно интерполируются только для центра пикселя. Иными словами, значение, которое попадает во фрагментный шейдер, рассчитывается процедурой интерполяции для точки с координатами, соответствующими центру фрагмента, которая, как мы уже знаем, может оказаться за пределами полигона! Если положиться на тот факт, что значения входных переменных шейдеров интерполируются строго между соответствующими значениями в вершинах (и никак не за границами этого диапазона), это может привести к неожиданным результатам.

Например, рассмотрим следующий отрывок из фрагментного шейдера:

```
in vec2 TexCoord;

layout( location = 0 ) out vec4 FragColor;

void main()
{
  vec3 yellow = vec3(1.0,1.0,0.0);
  vec3 color = vec3(0.0);          // черный
  if( TexCoord.s > 1.0 )
    color = yellow;
  FragColor = vec4( color , 1.0 );
}
```

Этот шейдер окрашивает фрагменты полигона черным цветом, только если компонент `s` координат текстуры не больше единицы. В противном случае фрагмент

окрашивается черным цветом. Если отобразить квадрат с координатами текстуры, изменяющимися от нуля до единицы во всех направлениях, модно получить результат, изображенный на рис. 5.13 слева. На этом рисунке показан увеличенный край полигона, где координата s текстуры колеблется около `1.0`. Оба изображения были получены с помощью предыдущего шейдера. Изображение справа создано с использованием квалификатора `centroid` (подробнее об этом рассказывается ниже).

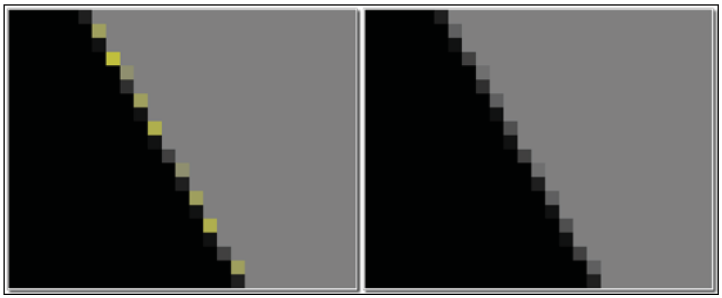

Рис. 5.13 ❖ Увеличенный край полигона, где координата s текстуры колеблется около 1.0

На изображении слева видно, что некоторые пиксели, расположенные вдоль края, имеют более светлый цвет (желтый, если вы рассматриваете цветное изображение). Это объясняется тем, что координаты текстуры интерполируются для центра пикселя, а не для какого-то из его субпикселей. Центр некоторых фрагментов на краю оказывается за пределами полигона, вследствие чего координата *s* текстуры оказывается больше единицы!

Существует возможность потребовать от OpenGL вычислить значение входной переменной путем интерполяции для некоторого местоположения, которое находится не только внутри пикселя, но и внутри полигона. Сделать это можно с помощью квалификатора `centroid`, как показано ниже:

```
centroid in vec2 TexCoord;
```

(Квалификатором также должна быть отмечена соответствующая выходная переменная в вершинном шейдере.) Если в предыдущий шейдер добавить квалификатор `centroid`, получится картина, как показано на рис. 5.13 справа.

Вообще говоря, квалификаторы `centroid` и `sample` следует использовать, когда известно, что интерполированные значения входных переменных не должны превышать значения этих же переменных в вершинах.

Квалификатор `sample` вынуждает OpenGL интерполировать значения входных переменных шейдера для фактических координат субпикселя.

```
sample in vec2 TexCoord;
```

Безусловно, в этом случае фрагментный шейдер будет вызываться для каждого субпикселя. Это позволит получить более точные результаты, но может при-

вести к неприемлемому падению производительности, особенно неприемлемому, если результаты применения квалификатора `centroid` (или без него) оказываются вполне удовлетворительными.

Отложенное освещение и затенение

Отложенное освещение и затенение (deferred shading) – это прием, основанный на переносе (или «откладывании») шага вычисления освещенности во второй проход. Обычно это делается, чтобы исключить возможность вычисления цвета для одного и того же пикселя более чем один раз. Основная идея состоит в следующем:

1. В первом проходе выполняется отображение сцены, но вместо вычисления цвета фрагмента в модели освещенности/затенения мы просто сохраняем всю информацию о геометрии (координаты, нормаль, координаты текстуры, коэффициент отражения и др.) во множестве промежуточных буферов, которые все вместе называют **g-буфером** (где «g» происходит от слова *geometry* – «геометрия»).
2. Во втором проходе выполняется чтение данных из g-буфера, вычисляется цвет фрагмента в модели освещенности/затенения и сохраняется окончательный цвет для каждого пикселя.

Использование приема отложенного освещения и затенения помогает избежать вычислений цвета для фрагментов, которые в конечном итоге оказываются невидимыми. Например, представьте пиксель в области, где перекрываются два полигона. Фрагментный шейдер может быть вызван по одному разу для каждого полигона, перекрывающего пиксель, однако только один результат из двух будет принят в качестве окончательного (предполагается, что смешивание цветов здесь не выполняется). Фактически время, потраченное на вычисление цвета для одного из полигонов, будет израсходовано впустую. Применение приема отложенного освещения и затенения позволяет отложить вычисление цвета до момента, когда вся геометрия будет обработана и принадлежность каждого пикселя будет определена. Соответственно, вычисление цвета будет выполнено для каждого пикселя только один раз, что может способствовать увеличению эффективности вычислений. Например, этот прием позволит использовать сотни источников света, потому что вычисление цвета для каждого пикселя будет выполнено только один раз.

Прием отложенного освещения и затенения очень прост для понимания и использования. Соответственно, его можно широко использовать для реализации сложных моделей освещения/затенения.

В данном рецепте рассматривается простой пример отложенного освещения и затенения. В g-буфере будет сохраняться следующая информация: местоположение, нормаль, цвет рассеянного освещения. Во втором проходе будет просто вычисляться цвет в соответствии с моделью освещения рассеянным светом, с использованием данных из g-буфера.

Этот рецепт задумывался только как учебный пример реализации приема отложенного освещения и затенения. Чтобы применить этот прием в действующем приложении, может потребоваться сохранить в g-буфере больший объем информации. Я думаю, что для вас не составит труда расширить этот пример для реализации более сложных моделей освещения/затенения.

Подготовка

В этом примере g-буфер будет содержать три текстуры для хранения координат, нормали и цвета рассеянного освещения. Этим трем текстурам соответствуют три uniform-переменные: `PositionTex`, `NormalTex` и `ColorTex`. Текстуры должны быть связаны с текстурными слотами `0`, `1` и `2` соответственно. Аналогично в вершинном шейдере предполагается, что информация о местоположении будет передаваться в атрибуте с индексом `0`, нормаль – в атрибуте с индексом `1`, и координаты текстуры – в атрибуте с индексом `2`.

Фрагментный шейдер имеет несколько uniform-переменных, имеющих отношение к свойствам света и материала, в частности структуры `Light` и `Material`, которые должны устанавливаться в основном приложении.

Также для хранения дескриптора объекта буфера кадра (FBO) потребуется переменная `deferredFBO` (типа `GLuint`).

Как это делается...

Чтобы создать шейдерную программу, реализующую отложенное освещение и затенение (только с учетом освещения рассеянным светом), нужно выполнить следующие шаги:

1. Создать объект буфера кадра для хранения g-буфера:

```
void createGBufTex(GLenum texUnit, GLenum format,
                   GLuint &texid )
{
  glActiveTexture(texUnit);
  glGenTextures(1, &texid);
  glBindTexture(GL_TEXTURE_2D, texid);
  glTexStorage2D(GL_TEXTURE_2D,1,format,width,height);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                  GL_NEAREST);
}
...
GLuint depthBuf, posTex, normTex, colorTex;

// Создать и связать FBO
glGenFramebuffers(1, &deferredFBO);
glBindFramebuffer(GL_FRAMEBUFFER, deferredFBO);

// Буфер глубины
glGenRenderbuffers(1, &depthBuf);
glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
```

```
                        width, height);

// Буферы для координат, нормалей и цвета
createGBufTex(GL_TEXTURE0, GL_RGB32F, posTex);  // Координаты
createGBufTex(GL_TEXTURE1, GL_RGB32F, normTex); // Нормаль
createGBufTex(GL_TEXTURE2, GL_RGB8, colorTex);  // Цвет

// Подключить изображения к буферу кадра
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
          GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthBuf);
glFramebufferTexture2D(GL_FRAMEBUFFER,
          GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, posTex, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER,
          GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, normTex, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER,
          GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D, colorTex, 0);

GLenumdrawBuffers[] = {GL_NONE, GL_COLOR_ATTACHMENT0,
          GL_COLOR_ATTACHMENT1,GL_COLOR_ATTACHMENT2};
glDrawBuffers(4, drawBuffers);
```

2. Добавить вершинный шейдер:

```
layout( location = 0 ) in vec3 VertexPosition;
layout( location = 1 ) in vec3 VertexNormal;
layout( location = 2 ) in vec2 VertexTexCoord;

out vec3 Position;
out vec3 Normal;
out vec2 TexCoord;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;
void main()
{
  Normal = normalize( NormalMatrix * VertexNormal);
  Position = vec3( ModelViewMatrix *
                   vec4(VertexPosition,1.0) );
  TexCoord = VertexTexCoord;
  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

3. Добавить фрагментный шейдер:

```
struct LightInfo {
  vec4 Position;  // Позиция источника света в видимых координатах
  vec3 Intensity; // Интенсивность рассеянного света
};
uniform LightInfo Light;

struct MaterialInfo {
  vec3 Kd; // Коэффициент отражения рассеянного света
};
```

```
uniform MaterialInfo Material;

subroutine void RenderPassType();
subroutine uniform RenderPassType RenderPass;

// Текстуры g-буфера
layout(binding = 0) uniform sampler2D PositionTex;
layout(binding = 1) uniform sampler2D NormalTex;
layout(binding = 2) uniform sampler2D ColorTex;
in vec3 Position;
in vec3 Normal;
in vec2 TexCoord;

layout (location = 0) out vec4 FragColor;
layout (location = 1) out vec3 PositionData;
layout (location = 2) out vec3 NormalData;
layout (location = 3) out vec3 ColorData;

vec3 diffuseModel( vec3 pos, vec3 norm, vec3 diff )
{
  vec3 s = normalize(vec3(Light.Position) - pos);
  float sDotN = max( dot(s,norm), 0.0 );
  vec3 diffuse = Light.Intensity * diff * sDotN;

  return diffuse;
}

subroutine (RenderPassType)
void pass1()
{
  // Сохранить позицию, нормаль и цвет освещения в g-буфере
  PositionData = Position;
  NormalData = Normal;
  ColorData = Material.Kd;
}

subroutine(RenderPassType)
void pass2()
{
  // Извлечь позицию, нормаль и цвет
  // из g-буфера
  vec3 pos = vec3( texture( PositionTex, TexCoord ) );
  vec3 norm = vec3( texture( NormalTex, TexCoord ) );
  vec3 diffColor = vec3( texture(ColorTex, TexCoord) );

  FragColor=vec4(diffuseModel(pos,norm,diffColor), 1.0);
}

void main() {
  // вызовет функцию pass1 или pass2 подпрограммы
  RenderPass();
}
```

В функции отображения в основном приложении в первом проходе нужно выполнить следующие шаги:

1. Связать объект буфера кадра `deferredFBO`.
2. Очистить буферы цвета/глубины, выбрать функцию `pass1` подпрограммы и включить проверку глубины (если необходимо).
3. Отобразить сцену как обычно.

Во втором проходе нужно выполнить следующие шаги:

1. Вернуться к буферу кадра по умолчанию (связать буфер кадра `0`).
2. Очистить буферы цвета/глубины, выбрать функцию `pass2` подпрограммы и выключить проверку глубины (если необходимо).
3. Отобразить прямоугольник (или два треугольника), покрывающий весь экран, с координатами текстуры от нуля до единицы в каждом направлении.

Как это работает...

При подготовке объекта буфера кадра (FBO), играющего роль g-буфера, использованы текстуры с внутренним форматом `GL_RGB32F` для представления координат и нормалей. Так как они должны хранить информацию о геометрии, а не о цвете, необходимо использовать формат высокого разрешения (с большим числом битов на пиксель). Буфер для хранения цвета рассеянного освещения создан с форматом `GL_RGB8`, так как для хранения информации о цвете не требуется высокое разрешение.

Затем три текстуры подключаются к объекту буфера кадра в точки подключения буферов цвета с индексами `0`, `1` и `2`, вызовом функции `glFramebufferTexture2D`. Затем вызовом `glDrawBuffers` они подключаются к выходным переменным фрагментного шейдера.

```
glDrawBuffers(4, drawBuffers);
```

Массив `drawBuffers` определяет взаимосвязь между компонентами буфера кадра и индексами выходных переменных фрагментного шейдера: *i*-му элементу в массиве соответствует выходная переменная с индексом *i*. Этот вызов связывает точки подключения буферов цвета `0`, `1` и `2` с выходными переменными, с индексами `1`, `2` и `3` соответственно. (Обратите внимание, что во фрагментном шейдере этим индексам соответствуют переменные `PositionData`, `NormalData` и `ColorData`.)

Вершинный шейдер играет роль простого «передаточного звена». Он всего лишь преобразует позицию и нормаль в видимые координаты (с началом в позиции камеры) и передает их дальше, фрагментному шейдеру. Координаты текстуры передаются без изменений.

Строго говоря, во втором проходе не требуется преобразовывать и передавать нормаль и позицию, так как они не используются во фрагментном шейдере. Однако, чтобы сохранить код простым, я не стал включать эту оптимизацию. Чтобы реализовать ее, достаточно добавить в вершинный шейдер подпрограмму, которая «выключала» бы преобразование во втором проходе. (Конечно, устанавливать значение переменной `gl_Position` необходимо в любом случае.)

Действия, выполняемые во фрагментном шейдере, зависят от значения переменной-подпрограммы `RenderPass`. Она будет вызывать либо функцию `pass1`, либо функцию `pass2`. В функции `pass1` сохраняются значения `Position`, `Normal` и `Material`.

Kd в соответствующих выходных переменных, что равносильно сохранению их в соответствующих текстурах, о которых только что рассказывалось.

В функции pass2 позиция, нормаль и цвет извлекаются из текстур и используются для вычисления освещения рассеянным светом. Результат сохраняется в выходной переменной FragColor. В этом проходе переменная FragColor должна быть связана с буфером кадра по умолчанию, чтобы результаты отобразились на экране.

И еще...

В сообществе специалистов по компьютерной графике постоянно ведутся дискуссии о достоинствах и недостатках приема отложенного освещения и затенения. Безусловно, этот прием подходит не для всех ситуаций. Его применимость в значительной степени зависит от конкретных требований, предъявляемых к приложению, и нужно тщательно взвесить все «за» и «против», прежде чем решить, стоит ли использовать отложенное освещение и затенение.

В последних версиях OpenGL появилась возможность создать контекст с поддержкой сглаживания множественной выборкой и отложенного освещения и затенения, задействовав флаг GL_TEXTURE_2D_MULTISAMPLE.

Еще одним существенным недостатком приема отложенного освещения и затенения является плохая поддержка смешивания полупрозрачных цветов. Фактически в простой реализации, такой как выше, смешивание вообще невозможно. Чтобы обеспечить возможность смешивания цветов, придется добавить в g-буфер дополнительные буферы с информацией о глубине дополнительных уровней графики.

Одним из важнейших преимуществ приема отложенного освещения и затенения является возможность в первом проходе сохранить информацию о глубине в текстуре и извлечь ее во втором проходе. Доступность полной информации о глубине в виде текстуры позволит реализовать такие эффекты, как глубина резкости (размытие удаленных объектов), прегражденние окружающего света в экранном пространстве (screen space ambient occlusion), объемные частицы, и многие другие.

За дополнительной информацией об отложенном освещении и затенении обращайтесь к главе 9 книги «GPU Gems 2», вышедшей под редакцией Мэтта Фарра (Matt Pharr) и Рандима Фернандо (Randima Fernando) (Addison-Wesley Professional 2005), и к главе 19 книги «GPU Gems 3», вышедшей под редакцией Хьюберта Нгуена (Hubert Nguyen) (Addison-Wesley Professional, 2007). Вместе они представляют отличное обсуждение преимуществ и недостатков приема отложенного освещения и затенения, и с их помощью вы сможете принять обоснованное решение о применении или неприменении этого приема в своем приложении.

См. также

- Рецепт «Отображение в текстуру» в главе 4 «Текстуры».

Реализация порядконезависимой прозрачности

Прозрачность – сложный эффект, и он с трудом поддается точному воспроизведению в конвейерных архитектурах, таких как OpenGL. Когда требуется получить данный эффект, обычно сначала рисуют все непрозрачные объекты, с включенным буфером глубины, затем буфер глубины делают доступным только для чтения (вызовом `glDepthMask`), запрещают проверку глубины и рисуют прозрачные объекты. При этом прозрачные объекты должны рисоваться в направлении «от дальних к ближним». То есть объекты, расположенные дальше от наблюдателя, должны рисоваться первыми, а объекты, расположенные ближе, – последними. Для этого перед отображением объекты нужно отсортировать по глубине.

На рис. 5.14 изображено множество маленьких, полупрозрачных шаров с несколькими полупрозрачными кубами, расположенными среди них. Рисунок справа получен путем отображения объектов в некотором произвольном порядке, с использованием стандартного механизма смешивания цветов в OpenGL. Результат выглядит неправильным, потому что смешивание выполнено в неправильном порядке. Кубы, которые здесь были нарисованы последними, выглядят так, как если бы они находились перед шарами, и сами шары смешиваются между собой в хаотичном порядке, что особенно заметно в середине блока. Рисунок слева нарисован с учетом порядка расположения объектов, поэтому картинка выглядит более реалистично, как и задумывалось.

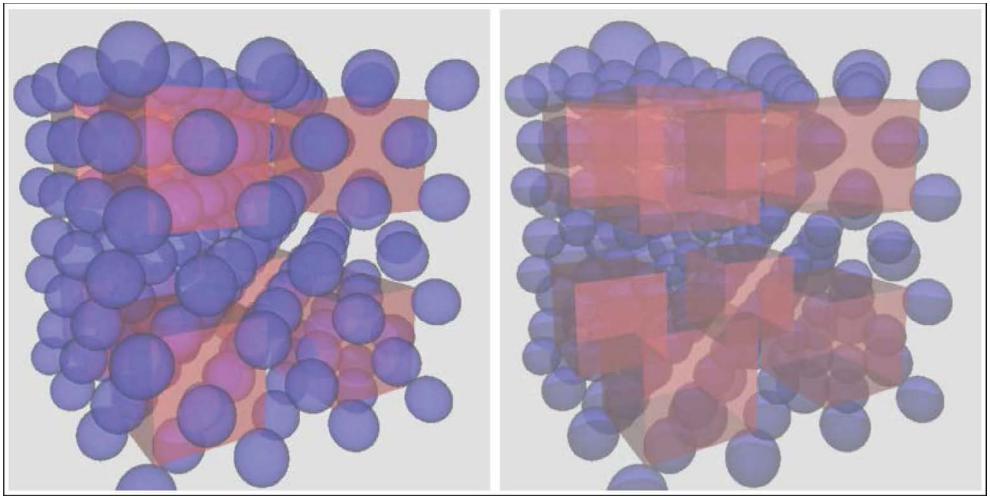

Рис. 5.14 ❖ Полупрозрачные шары и кубы, нарисованные с учетом (слева) и без учета (справа) их глубины

Под **порядконезависимой прозрачностью (Order Independent Transparency, OIT)** подразумевается возможность рисовать объекты в любом порядке и получать правильные результаты. Сортировка по глубине при этом выполняется на

некотором другом уровне, возможно, внутри фрагментного шейдера, чтобы избавить программиста от необходимости сортировать объекты перед отображением. Существуют разные способы реализации данной концепции. Наиболее часто используемым из них является сохранение списка цветов для каждого пикселя, сортировка их по глубине и смешивание во фрагментном шейдере. Именно он и будет применен для реализации OIT в данном рецепте. Попутно мы воспользуемся некоторыми новшествами, появившимися в версии OpenGL 4.3.

В версиях OpenGL 4.3 и 4.2 появились новые механизмы: **буферные объекты хранилища шейдера (Shader Storage Buffer Objects, SSBO)** и **загрузки/сохранения изображений** соответственно. Они поддерживают возможность произвольного чтения/записи данных внутри шейдера. До этих пор шейдеры были весьма ограничены в возможностях доступа к данным. Да, они могли читать данные из разных мест (таких как текстуры, uniform-переменные и т. д.), но возможность произвольной записи данных была весьма ограничена. Шейдеры могли записывать данные только в управляемые, изолированные области памяти, такие как выходные переменные и буферы трансформаций с обратной связью (transform feedback buffers). На то были вполне веские причины. Так как шейдеры могут обрабатывать данные параллельно и в произвольном порядке, очень сложно гарантировать непротиворечивость данных между экземплярами шейдеров. Данные, записанные одним экземпляром шейдера, были недоступны для другого экземпляра, независимо от порядка их выполнения. Но, как бы то ни было, существуют не менее веские причины в пользу возможности чтения/записи данных в разделяемые области памяти. С появлением механизмов SSBO и загрузки/записи изображений такая возможность появилась. Мы можем создавать буферы и текстуры (называемые изображениями), доступные для чтения/записи любым экземплярам шейдеров. Это особенно ценная возможность для вычислительных шейдеров, о которых рассказывается в главе 10 «Вычислительные шейдеры». Однако новые возможности имеют свою цену. Теперь программист должен быть особенно осторожен, дабы избежать проблем, связанных с непротиворечивостью данных в памяти, которые характерны при наличии общей памяти, доступной для записи параллельно выполняющимся потокам. Кроме того, программист должен знать, как сказывается на производительности применение механизмов синхронизации операций доступа к памяти.

Более подробное обсуждение проблем, связанных с доступом к памяти и шейдерами, можно найти в главе 11 книги «The OpenGL Programming Guide, 8th Edition»[1]. Там же описывается еще одна, похожая реализация OIT.

В этом рецепте для реализации порядконезависимой прозрачности будут использоваться механизмы SSBO и загрузки/записи изображений. Операции будут выполнены за два прохода. В первом проходе будет отображена геометрия сцены

[1] На русский язык переведено 4-е издание книги: Ву М., Девис Т., Нейдер Дж., Шрайнер Д. OpenGL. Руководство по программированию. – Питер, 2006. – ISBN: 5-94723-827-6. – *Прим. перев.*

и для каждого пикселя создан список фрагментов. По окончании первого прохода каждый пиксел будет иметь соответствующий связанный список всех фрагментов, записанных в этот пиксель, вместе с их глубинами и цветом. Во втором проходе будет отображен прямоугольник, покрывающий весь экран, чтобы вызвать фрагментный шейдер еще раз для каждого пикселя. На этот раз шейдер будет извлекать связанные списки, сортировать фрагменты по глубине (от большей к меньшей) и смешивать цвета в этом порядке. Затем окончательный цвет будет передан в выходную переменную.

Это была основная идея, а теперь обратимся к деталям ее реализации. Нам потребуются три области памяти, совместно используемых экземплярами шейдера.

1. **Атомарный счетчик**: переменная, хранящая целое число без знака, с помощью которой будет подсчитываться размер буфера для связанного списка. Ее можно рассматривать как индекс первой неиспользуемой ячейки в буфере.
2. **Текстура указателей на списки, размер которой соответствует размеру экрана**: текстура, каждый тексель которой хранит единственное целое число без знака. Значение индекса – номер ячейки в буфере, где начинается связанный список для данного пикселя.
3. **Буфер, содержащий все связанные списки**: каждый элемент в буфере соответствует фрагменту и содержит структуру с цветом и глубиной фрагмента, а также целое число, определяющее индекс следующего фрагмента.

Чтобы понять, как все это действует, рассмотрим простой пример. Допустим, что экран имеет размер 3×3 пикселя. В этом случае мы создадим текстуру с теми же размерами, хранящую указатели на списки, каждый тексель в которой инициализирован специальным значением – признаком конца списка. На рис. 5.15 это значение показано как `'x'`, но в действительности это число `0xffffffff`. Изначально счетчик равен нулю. Буфер, предназначенный для хранения связанных списков, имеет определенный размер, но первоначально интерпретируется как пустой. Начальное состояние памяти изображено на рис. 5.15.

Теперь предположим, что отображается фрагмент с координатами `(0,1)` и с глубиной `0.75`. Фрагментный шейдер в этом случае выполнит следующие шаги:

1. Увеличит атомарный счетчик на единицу, в результате счетчик получит новое значение `1`, но сам шейдер будет использовать прежнее значение (`0`) и интерпретировать его как индекс нового узла в связанном списке.
2. Запишет прежнее значение счетчика (`0`) в текстуру с указателями на списки (в элемент с координатами `(0,1)`). Это индекс нового начала связанного списка для данного пикселя. Сохранит прежнее значение из текстуры (`x`), оно понадобится на следующем шаге.
3. Добавит новый элемент в буфер, в ячейку с индексом, значение которого равно прежнему значению атомарного счетчика (`0`). Сохранит в элементе цвет фрагмента и его глубину. Сохранит в поле «следующий» предыдущее значение указателя на список из ячейки текстуры с координатами `(0,1)`, которое было сохранено на шаге 2. В данном случае это специальное число, обозначающее конец списка.

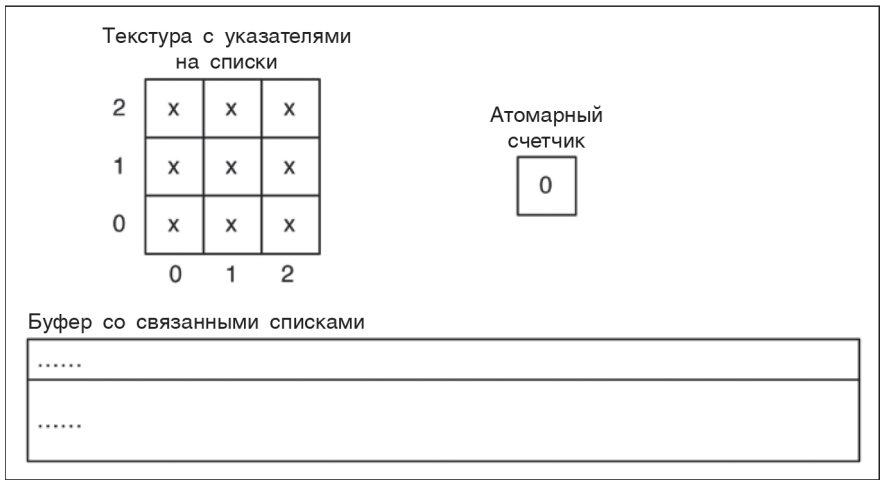

Рис. 5.15 ❖ Начальное состояние памяти

После обработки этого фрагмента состояние памяти будет выглядеть, как показано на рис. 5.16.

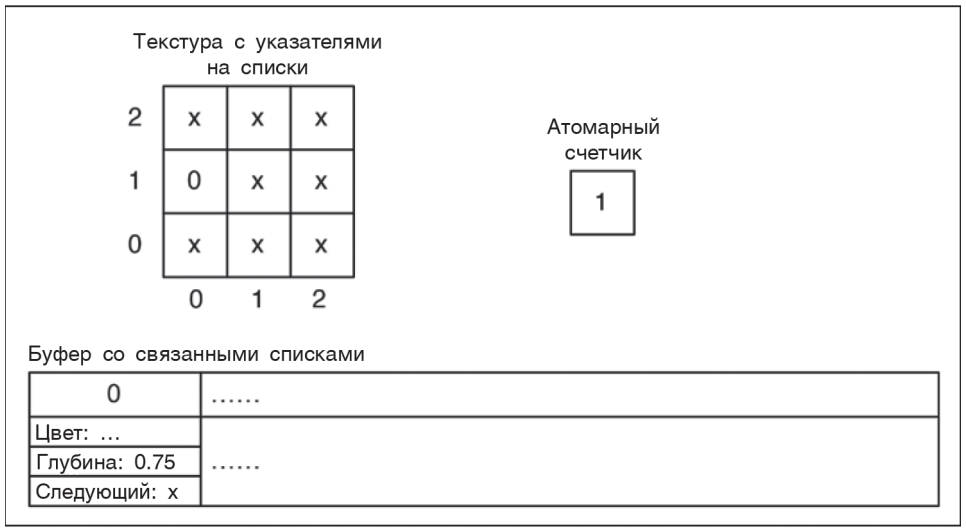

Рис. 5.16 ❖ Состояние памяти
после обработки первого фрагмента с координатами (0,1)

Теперь предположим, что в координатах `(0,1)` отображается еще один фрагмент с глубиной `0.5`. Фрагментный шейдер выполнит те же шаги, в результате чего состояние памяти будет выглядеть, как показано на рис. 5.17.

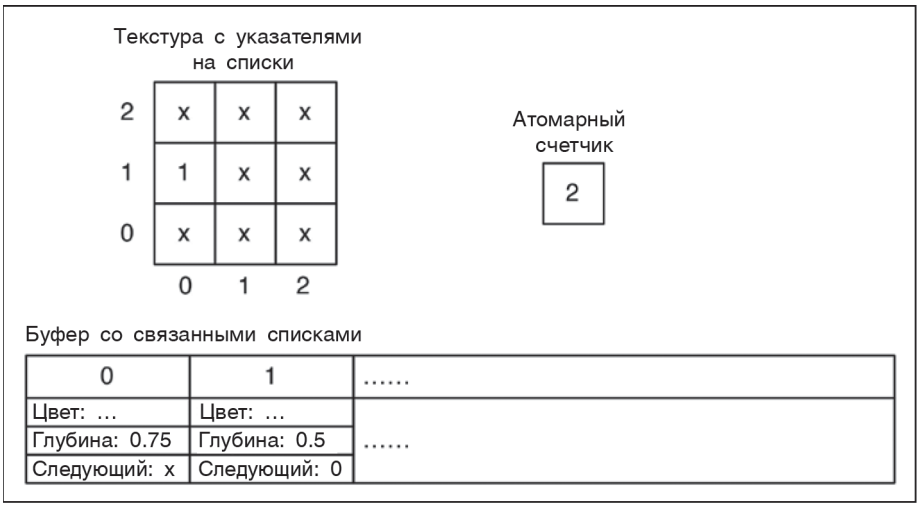

Рис. 5.17 ❖ Состояние памяти
после обработки второго фрагмента с координатами (0,1)

Теперь у нас имеется двухэлементный связанный список, начинающийся элементом буфера с индексом `1` и заканчивающийся элементом с индексом `0`. Предположим, что далее было отображено еще три фрагмента в следующем порядке: фрагмент в позиции `(1,1)` с глубиной `0.2`, фрагмент в позиции `(0,1)` с глубиной `0.3` и фрагмент в позиции `(1,1)` с глубиной `0.4`. Для каждого фрагмента будут выполнены все те же шаги, в результате которых состояние памяти будет выглядеть, как показано на рис. 5.18.

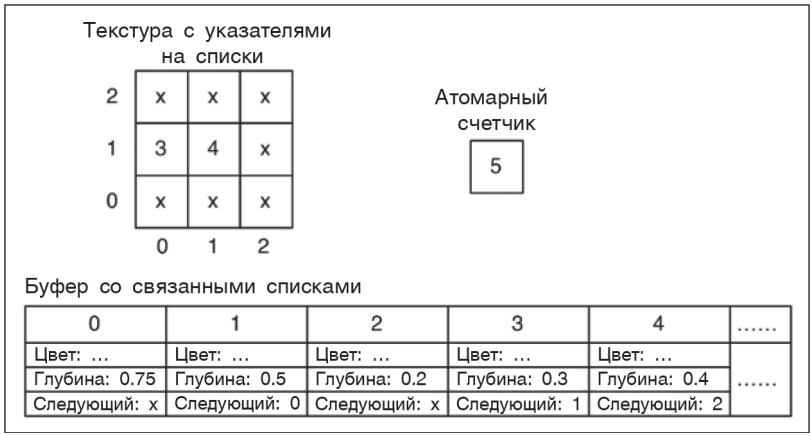

Рис. 5.18 ❖ Состояние памяти
после обработки еще трех фрагментов

Теперь в связанном списке для пикселя с координатами (0,1) хранятся три фрагмента {3, 1, 0} и в связанном списке для пикселя с координатами (1,1) – два фрагмента {4, 2}.

Имейте в виду, что из-за параллельной природы GPU фрагменты могут отображаться в любом порядке. Например, фрагменты двух разных полигонов могут пропускаться через конвейер в порядке, противоположном порядку выполнения инструкций их рисования. Программист не должен предполагать, что фрагменты будут обрабатываться в каком-то определенном порядке. В действительности инструкции в разных экземплярах фрагментных шейдеров могут перемежаться в произвольном порядке. Единственное, в чем можно быть уверенными, – инструкции внутри конкретного экземпляра шейдера будут выполняться по порядку. Соответственно, нужны некоторые гарантии, что после произвольного перемежения трех шагов, выполняемых разными экземплярами шейдеров, данные останутся в непротиворечивом состоянии. Например, представьте, что один экземпляр выполнил шаги 1 и 2, а другой экземпляр (обрабатывающий другой фрагмент, возможно, даже с теми же координатами) успел выполнить шаги 1, 2 и 3 до того, как первый экземпляр успел приступить к выполнению шага 3. Получится ли результат непротиворечивым? Я думаю, вы сами сможете убедиться, что данные действительно сохранят непротиворечивость, даже при том что на протяжении короткого промежутка времени в процессе вычислений целостность списка будет нарушена. Попробуйте сами исследовать разные варианты перемежения экземпляров шейдеров и убедитесь, что все работает.

Перемежаться могут не только инструкции в разных экземплярах шейдеров, но и субинструкции, составляющие инструкции. (Например, инструкция увеличения значения на единицу состоит из трех субинструкций: загрузки значения из памяти, его увеличения и сохранения в памяти.) Более того, в разных экземплярах шейдеров они могут выполняться одновременно. Следовательно, если не принять меры предосторожности, может возникнуть масса проблем, связанных с разрушением данных в памяти. Чтобы избежать их, следует использовать поддержку атомарных операций в языке GLSL.

В последних версиях OpenGL (4.2 и 4.3) появились инструменты, делающие такие алгоритмы возможными. В OpenGL 4.2 появились атомарные счетчики и возможность чтения/записи данных в произвольные элементы текстуры. В OpenGL 4.3 появились буферные объекты хранилищ шейдеров (SSBO). В данном примере будут использоваться все три новые особенности, а также различные атомарные операции и барьеры доступа к памяти.

Подготовка

В этом рецепте требуется выполнить достаточно сложные подготовительные операции, поэтому я буду сопровождать свои пояснения фрагментами программного кода. Прежде всего необходимо подготовить буфер для атомарного счетчика:

```
GLuint counterBuffer;
glGenBuffers(1, &counterBuffer);
glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 0, counterBuffer);
glBufferData(GL_ATOMIC_COUNTER_BUFFER, sizeof(GLuint), NULL,
             GL_DYNAMIC_DRAW);
```

Далее нужно создать буфер для хранения связанных списков:

```
GLuint llBuf;
glGenBuffers(1, &llBuf);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, llBuf);
glBufferData(GL_SHADER_STORAGE_BUFFER, maxNodes * nodeSize, NULL,
             GL_DYNAMIC_DRAW);
```

`nodeSize` в предыдущем фрагменте – это размер структуры `NodeType`, используемой внутри фрагментного шейдера (будет показан далее). Этот размер вычисляется с применением квалификатора `std430`. За подробностями о квалификаторе `std430` обращайтесь к спецификации OpenGL. В данном примере `nodeSize` вычисляется как `5 * sizeof(GLfloat) + sizeof(GLuint)`.

Также нужно создать текстуру для хранения указателей на начала списков. В качестве значений текстуры используются 32-разрядные целые без знака, а сама текстура связана с текстурным слотом `0`:

```
glGenTextures(1, &headPtrTex);
glBindTexture(GL_TEXTURE_2D, headPtrTex);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_R32UI, width, height);
glBindImageTexture(0, headPtrTex, 0, GL_FALSE, 0, GL_READ_WRITE,
                   GL_R32UI);
```

После отображения каждого кадра нужно очищать текстуру записью во все тексели значения `0xffffffff`. С этой целью создается буфер того же размера, что и текстура, каждое значение которого равно `0xffffffff`:

```
vector<GLuint> headPtrClear(width * height, 0xffffffff);
GLuint clearBuf;

glGenBuffers(1, &clearBuf);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, clearBuf);
glBufferData(GL_PIXEL_UNPACK_BUFFER,
             headPtrClear.size()*sizeof(GLuint),
             &headPtrClear[0], GL_STATIC_COPY);
```

Это все буферы, что нам нужны. Отметьте, что текстура с указателями связана с текстурным слотом `0`, буфер атомарного счетчика связан с индексом `0` точки привязки `GL_ATOMIC_COUNTER_BUFFER` (`glBindBufferBase`) и буфер для хранения списков связан с индексом `0` точки привязки `GL_SHADER_STORAGE_BUFFER`. Ниже мы еще вернемся к этому.

Вершинный шейдер просто играет роль передаточного звена и передает дальше позицию и нормаль вершины в видимых координатах.

Как это делается...

После подготовки всех буферов нужно выполнить два прохода отображения. Перед первым проходом следует очистить буферы, заполнив их значениями по умолчанию, и сбросить буфер атомарного счетчика, сохранив в нем нулевое значение.

```
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, clearBuf);
glBindTexture(GL_TEXTURE_2D, headPtrTex);
```

```
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height,
                GL_RED_INTEGER, GL_UNSIGNED_INT, NULL);
GLuint zero = 0;
glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 0, counterBuffer);
glBufferSubData(GL_ATOMIC_COUNTER_BUFFER, sizeof(GLuint), &zero);
```

В первом проходе отображается полная геометрия сцены. В общем случае требуется сначала отобразить все непрозрачные объекты и сохранить результаты в текстуре. Однако в данном примере мы пропустим этот шаг, чтобы не перегружать его лишним кодом и не отвлекаться от основной темы, и рассмотрим только отображение геометрии прозрачных объектов. Перед отображением прозрачных объектов нужно сделать буфер глубины доступным лишь для чтения (`glDepthMask`). Внутри фрагментного шейдера каждый фрагмент добавляется в соответствующий связанный список.

```
layout (early_fragment_tests) in;

#define MAX_FRAGMENTS 75

in vec3 Position;
in vec3 Normal;

struct NodeType {
  vec4 color;
  float depth;
  uint next;
};

layout(binding=0, r32ui) uniform uimage2D headPointers;
layout(binding=0, offset=0) uniform atomic_uint
                              nextNodeCounter;

layout(binding=0, std430) buffer linkedLists {
  NodeType nodes[];
};
uniform uint MaxNodes;

subroutine void RenderPassType();
subroutine uniform RenderPassType RenderPass;

...

subroutine(RenderPassType)
void pass1()
{
  // Получить индекс следующего пустого слота в буфере
  uint nodeIdx = atomicCounterIncrement(nextNodeCounter);

  // В буфере осталось место?
  if( nodeIdx < MaxNodes ) {
    // Обновить указатель на начало списка в текстуре
    uint prevHead = imageAtomicExchange(headPointers,
                      ivec2(gl_FragCoord.xy), nodeIdx);
    // Сохранить цвет и грубину фрагмента в элементе
```

```
    // списка. Сохранить в поле next указатель
    // на прежнее начало списка.
    nodes[nodeIdx].color = vec4(shadeFragment(), Kd.a);
    nodes[nodeIdx].depth = gl_FragCoord.z;
    nodes[nodeIdx].next = prevHead;
  }
}
```

Перед вторым проходом нужно гарантировать завершение записи всех данных в буферы. С этой целью здесь используется механизм барьеров.

```
glMemoryBarrier( GL_ALL_BARRIER_BITS );
```

Во втором проходе производится простое отображение прямоугольника экрана, чтобы обеспечить вызов фрагментного шейдера для каждого пикселя. Внутри фрагментного шейдера сначала выполняется копирование списка во временный массив.

```
struct NodeType frags[MAX_FRAGMENTS];
int count = 0;

// Получить индекс начала списка
uint n = imageLoad(headPointers, ivec2(gl_FragCoord.xy)).r;

// Скопировать список для данного фрагмента в массив
while( n != 0xffffffff && count < MAX_FRAGMENTS) {
  frags[count] = nodes[n];
  n = frags[count].next;
  count++;
}
```

Затем фрагменты сортируются методом вставки:

```
// Сортировать массив по значению глубины (от больших к меньшим)
for( uint i = 1; i < count; i++ )
{
  struct NodeType toInsert = frags[i];
  uint j = i;
  while( j > 0 && toInsert.depth > frags[j-1].depth ) {
    frags[j] = frags[j-1];
    j--;
  }
  frags[j] = toInsert;
}
```

В заключение выполняется смешивание фрагментов «вручную», и результат записывается в выходную переменную:

```
// Обойти массив и смешать цвета
vec4 color = vec4(0.5, 0.5, 0.5, 1.0); // Цвет фона
for( int i = 0; i < count; i++ ) {
  color = mix( color, frags[i].color, frags[i].color.a);
}

// Вывести получившийся цвет
FragColor = color;
```

Как это работает...

Для очистки буферов перед первым проходом выполняется связывание clearBuf с точкой привязки GL_PIXEL_UNPACK_BUFFER и вызывается glTexSubImage2D, чтобы скопировать данные из clearBuf в текстуру с указателями на списки. Обратите внимание, что когда к точке GL_PIXEL_UNPACK_BUFFER привязывается непустой буфер, функция glTexSubImage2D интерпретирует последний параметр как смещение в буфере. Таким способом инициируется копирование из clearBuf в headPtrTex. Очистка атомарного счетчика выполняется просто, но присутствие вызова функции glBindBufferBase может сбивать с толку. Если к точке привязывается несколько буферов (с разными индексами), как же тогда glBufferSubData узнает, какой буфер использовать? Как оказывается, выполняя привязку буфера вызовом glBindBufferBase, она также выполняет привязку к «универсальной» точке.

Фрагментный шейдер начинается с определения, разрешающего оптимизацию ранней проверки фрагмента.

```
layout (early_fragment_tests) in;
```

Это важно, потому что любые фрагменты, заслоненные непрозрачными объектами, не должны добавляться в список. Если отключить оптимизацию ранней проверки фрагментов, фрагментный шейдер будет вызываться и для фрагментов, проваливших проверку на глубину, и такие фрагменты будут добавлены в список. Предыдущая инструкция гарантирует, что фрагментный шейдер не будет вызываться для таких фрагментов.

Определение struct NodeType объявляет тип данных, которые будут храниться в списке. Для каждого фрагмента в списке должны сохраняться его цвет, глубина и указатель на следующий узел в связанном списке.

Следующие три инструкции объявляют объекты, имеющие отношение к хранилищу связанных списков. Первый, headPointers, – это объект изображения, где будут храниться индексы ячеек с началами списков, соответствующих фрагментам. Квалификатор layout указывает, что объект связан с нулевым слотом изображения (см. раздел «Подготовка» выше) и имеет тип r32ui (red, 32-bit, unsigned integer – красная составляющая, 32-разрядное целое без знака). Второй объект – это атомарный счетчик nextNodeCounter. Квалификатор layout определяет индекс буфера в точке связывания GL_ATOMIC_COUTER_BUFFER (см. раздел «Подготовка» выше) и смещение относительно начала буфера с этим индексом. Так как в данном буфере имеется только одно значение, смещение равно 0, но вообще в одном буфере можно хранить несколько атомарных счетчиков. Третий – буфер linkedLists, предназначенный для хранения связанных списков. Это буферный объект хранилища шейдера. Организация данных внутри буфера определяется содержимым фигурных скобок. В данном случае мы имеем простой массив структур NodeType. Границы массива можно не указывать, так как размер его будет ограничиваться размером буферного объекта. Квалификатор layout определяет привязку и местоположение в памяти. Первый параметр, binding, указывает, что буфер связан с индексом 0 в точке привязки GL_SHADER_STORAGE_BUFFER. Второй, std430, указыва-

ет, как организована память внутри буфера. Этот параметр играет важную роль в случаях, когда необходимо осуществлять чтение данных из буфера со стороны основной программы. Как отмечалось выше, эта возможность определяется спецификацией OpenGL.

В первом проходе фрагментный шейдер сразу же увеличивает атомарный счетчик вызовом `atomicCounterIncrement`. Эта функция увеличит счетчик на единицу так, что эта операция не нарушит непротиворечивость данных в памяти, даже если в это же время другой экземпляр шейдера также попытается увеличить счетчик.

Атомарной называют операцию, изолированную от других потоков выполнения, которую можно рассматривать как единую, непрерываемую операцию. Другие потоки выполнения не смогут повлиять на результат атомарной операции. При работе с объектами, разделяемыми между экземплярами шейдера, всегда используйте атомарные операции.

Функция `atomicCounterIncrement` возвращает предыдущее значение счетчика – следующая свободная ячейка в буфере. Оно будет использоваться как индекс в массиве для сохранения информации о текущем фрагменте, поэтому тут же сохраняется в переменной `nodeIdx`. Это значение также будет играть роль ссылки на новое начало связанного списка, поэтому на следующем шаге производится обновление значения в изображении `headPointers`, в ячейке `gl_FragCoord.xy`, соответствующей текущему пикселю. Это изменение выполняется в рамках еще одной атомарной операции: `imageAtomicExchange`. Она записывает значение, указанное в третьем параметре, в ячейку с координатами, указанными во втором параметре, и возвращает прежнее значение, хранившееся в этой ячейке. Это – ссылка на прежнее начало связанного списка. Оно сохраняется в переменной `prevHead`, потому что еще потребуется связать новое начало списка с остальной его частью и обеспечить целостность связанного списка.

В заключение в узел с индексом `nodeIdx` записываются цвет и глубина, а также ссылка на предыдущее начало списка (`prevHead`) для данного фрагмента. На этом вставка фрагмента в начало связанного списка завершается.

По завершении первого прохода необходимо убедиться, что все экземпляры фрагментного шейдера закончили работу и изменения сохранены в буфер хранилища и объект изображения. Единственный способ сделать это – использовать барьер. Вызов `glMemoryBarrier` сделает все, что необходимо. Единственный параметр функции `glMemoryBarrier` определяет тип барьера. Управляя типом барьера, можно, в частности, определить, какого рода данные предполагается читать. Однако исключительно для пущей безопасности и простоты в данном примере используется значение `GL_ALL_BARRIER_BITS`, гарантирующее завершение записи всех возможных данных перед продолжением.

Во втором проходе шейдер сначала копирует содержимое связанного списка во временный массив. Для этого он извлекает индекс головы списка из изображения `headPointers` с помощью `imageLoad`. Затем в цикле `while` выполняет обход узлов списка и копирует данные в массив `frags`.

Далее массив сортируется по глубине в порядке убывания (от больших значений к меньшим). Алгоритм сортировки методом вставки, реализованный здесь

с успехом, можно использовать для небольших массивов, поэтому его выбор вполне оправдан.

В заключение все фрагменты объединяются с помощью функции `mix` и с учетом значения альфа-канала. Окончательный результат сохраняется в выходной переменной `FragColor`.

И еще...

Как отмечалось выше, из описания алгоритма работы шейдера я исключил все, что относится к отображению непрозрачных объектов. В общем случае сначала следует обработать все непрозрачные объекты с включенным буфером глубины и сохранить фрагменты в текстуре. Затем перед отображением прозрачных объектов нужно запретить запись в буфер глубины и построить связанные списки, как было описано выше. И наконец, использовать значения цвета из непрозрачной текстуры в качестве фона при выполнении смешивания значений из связанных списков.

Это был первый пример в данной книге, где использовалась возможность выполнения операций чтения/записи с произвольными ячейками разделяемого хранилища внутри шейдера. Данная возможность, появившаяся совсем недавно, дает огромную гибкость, но не бесплатно. Как отмечалось выше, необходимо проявлять особую осторожность, чтобы избежать нарушения целостности данных в памяти и других проблем синхронизации потоков выполнения. Для этого предусматриваются такие инструменты, как атомарные операции и барьеры, и данный пример показывает лишь вершину айсберга. Более подробно эта тема будет раскрыта в главе 10 «Вычислительные шейдеры», а кроме того, я советую прочитать главу, рассказывающую о памяти, в книге «OpenGL Programming Guide».

См. также

- Главу 10 «Вычислительные шейдеры».
- Книгу «OpenGL Development Cookbook» Мухаммада Мобин Мования (Muhammad Mobeen Movania), где в главе 6 «GPU-based Alpha Blending and Global Illumination» можно найти еще несколько рецептов.

Глава 6

Использование геометрических шейдеров и шейдеров тесселяции

В этой главе описываются следующие рецепты:

- отображение точечных спрайтов с помощью геометрического шейдера;
- наложение каркаса на освещенную поверхность;
- рисование линий силуэта с помощью геометрического шейдера;
- тесселяция кривой;
- тесселяция двухмерного прямоугольника;
- тесселяция трехмерной поверхности;
- тесселяция с учетом глубины.

Введение

Геометрические шейдеры и шейдеры тесселяции относительно недавно появились в конвейере OpenGL. Они дают программистам дополнительные возможности изменения геометрии по мере ее продвижения по конвейеру шейдеров. Геометрические шейдеры можно использовать для добавления, изменения или удаления объектов, а шейдеры тесселяции – с целью автоматической коррекции геометрии на различных уровнях детализации и упрощения интерполяции на основе произвольных входных данных («заплат»).

В этой главе рассматривается несколько примеров применения геометрических шейдеров и шейдеров тесселяции в разных контекстах. Однако, прежде чем приступить к исследованию рецептов, я предлагаю посмотреть, какое место они занимают в конвейере шейдеров.

Расширенный конвейер шейдеров

На рис. 6.1 приводится упрощенная схема конвейера шейдеров, где можно видеть, какое место занимают геометрический шейдер и шейдер тесселяции.

Часть конвейера, имеющая отношение к тесселяции, делится на два элемента: **шейдер управления тесселяцией (Tessellation Control Shader, TCS)** и **шейдер выполнения тесселяции (Tessellation Evaluation Shader, TES)**. Вслед за двумя

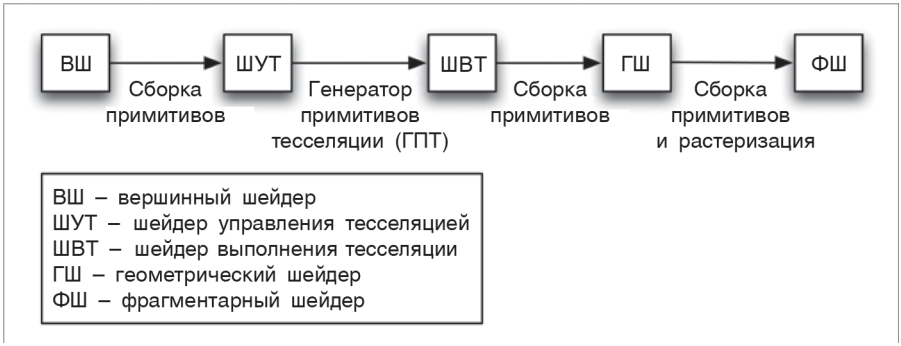

Рис. 6.1 ❖ Расширенный конвейер шейдеров

шейдерами тесселяции следует геометрический шейдер, и далее – фрагментный шейдер. Шейдеры тесселяции и геометрический шейдер являются необязательными; однако если шейдерная программа включает шейдер тесселяции или геометрический шейдер, она обязательно должна включать вершинный шейдер.

Кроме случая, упомянутого выше, все шейдеры являются необязательными. Однако если шейдерная программа не включает вершинного или фрагментного шейдера, результат ее работы не определен. Наличие геометрического шейдера не требует включать шейдер тесселяции, и наоборот. Очень редко можно встретить шейдерную программу, не включающую хотя бы вершинного и фрагментного шейдеров.

Геометрический шейдер

Геометрический шейдер (Geometry Shader, GS) вызывается один раз для каждого примитива. Ему доступны все вершины примитива, а также значения всех входных переменных, связанных с каждой вершиной. Иными словами, если на предыдущей стадии (например, в вершинном шейдере) определяются значения выходных переменных, геометрический шейдер получит доступ к значению этой переменной для всех вершин в примитиве. Как результат входные переменные в геометрическом шейдере всегда являются массивами.

Геометрический шейдер может выводить нуль, один или более примитивов. Выходные примитивы геометрического шейдера необязательно должны быть того же рода, что и входные примитивы. Но при этом геометрический шейдер может возвращать примитивы только одного типа. Например, геометрический шейдер может принимать треугольник и выводить несколько сегментов ломаной линии. Или он может принимать треугольник и выводить нуль или более треугольников в виде полосы из треугольников.

Это позволяет геометрическому шейдеру играть самые разные роли: отбраковывать (удалять) примитивы, опираясь на некоторые критерии; генерировать дополнительные примитивы для добавления деталей в форму отображаемого объекта; просто вычислять дополнительную информацию о примитиве и возвращать примитив в исходном виде; или создавать примитивы, полностью отличающиеся от входящих.

Функциональные возможности геометрического шейдера основаны на двух встроенных функциях, EmitVertex и EndPrimitive, позволяющих передавать множество вершин и примитивов дальше по конвейеру. Геометрический шейдер определяет выходные переменные для конкретной вершины и затем вызывает EmitVertex. Потом он может переопределить выходные переменные для следующей вершины и снова вызвать EmitVertex и т. д. После вывода всех вершин, составляющих примитив, графический шейдер может вызвать EndPrimitive, чтобы сообщить системе OpenGL, что все вершины примитива были выведены. Функция EndPrimitive неявно вызывается по завершении выполнения геометрического шейдера. Если геометрический шейдер вообще не вызовет EmitVertex, входной примитив фактически будет отброшен.

В рецептах, следующих ниже, мы познакомимся с несколькими примерами использования геометрического шейдера. В рецепте «Отображение точечных спрайтов с помощью геометрического шейдера» приводится пример, когда тип выходного примитива совершенно отличается от типа входного примитива. В рецепте «Наложение каркаса на освещенную поверхность» геометрический шейдер не изменяет входящих примитивов, но добавляет к ним некоторую дополнительную информацию, помогающую рисовать линии сетки. В рецепте «Рисование линий силуэта с помощью геометрического шейдера» демонстрируется пример, когда геометрический шейдер возвращает входящие примитивы, а также генерирует дополнительную геометрию.

Шейдеры тесселяции

При наличии шейдеров тесселяции можно отображать примитивы только одного типа: «заплаты» (GL_PATCHES). Попытки отобразить примитивы других типов (таких как треугольники или линии), при наличии активного шейдера тесселяции, будут приводить к ошибке. **Примитив «заплаты» (patch primitive)** – это произвольный «фрагмент» геометрии (или любой другой информации), полностью определяемой программистом. Он нигде больше не интерпретируется, кроме как в шейдерах управления тесселяцией (TCS) и выполнения тесселяции (TES). Число вершин в примитиве «заплаты» также может настраиваться. Максимальное число вершин на «заплату» зависит от реализации и может быть получено следующей командой:

```
glGetIntegerv(GL_MAX_PATCH_VERTICES, &maxVerts);
```

Задать число вершин на «заплату» можно с помощью такой функции:

```
glPatchParameteri( GL_PATCH_VERTICES, numPatchVerts );
```

Эта функция часто используется, когда примитив «заплаты» состоит из множества опорных точек, определяющих интерполируемую поверхность или кривую (например, поверхность или кривую Безье). Однако нет ничего, что препятствовало бы использованию информации в примитиве «заплаты» для других целей.

Фактически примитив «заплаты» никогда не отображается, а используется как средство передачи дополнительной информации в шейдеры управления тесселя-

цией и выполнения тесселяции (TCS и TES). Примитивы, которые передаются этими шейдерами дальше по конвейеру, создаются **генератором примитивов тесселяции (Tessellation Primitive Generator, TPG)**, располагающимся между шейдером управления тесселяцией и выполнения тесселяции. Генератор примитивов тесселяции можно рассматривать как настраиваемый механизм, который производит примитивы на основе набора стандартных алгоритмов тесселяции. Шейдеры управления и выполнения обладают доступом ко всей входящей «заплате», но области их ответственности отличаются кардинально. Шейдер управления тесселяцией отвечает за настройку генератора примитивов тесселяции – определяет, как должны генерироваться примитивы (какие алгоритмы должны использоваться), – и передачу выходных переменных-атрибутов для вершин. Шейдер выполнения тесселяции решает такие задачи, как определение координат (и любой другой информации) для каждой вершины в примитивах, которые создаются генератором TPG. Например, шейдер управления тесселяцией может потребовать от генератора примитивов создать ломаную линию, состоящую из 100 отрезков, а шейдер выполнения в этом случае будет отвечать за определение координат всех вершин для этих отрезков, при этом он наверняка будет использовать информацию из примитива «заплаты».

Шейдер управления выполняется один раз для каждой вершины в выходной «заплате». Он может вычислять дополнительную информацию о заплате и передавать ее дальше шейдеру выполнения, используя выходные переменные. Однако самое главное, что должен сделать шейдер управления, – сообщить генератору, сколько примитивов тесселяции он должен произвести. Делается это путем определения уровней тесселяции с помощью массивов `gl_TessLevelInner` и `gl_TessLevelOuter`. Эти массивы определяют детальность результатов, производимых генератором примитивов.

Генератор TPG создает примитивы, опираясь на определенные алгоритмы (прямоугольники, линии или треугольники). Все алгоритмы создания примитивов действуют немного по-разному, и в этой главе вы увидите примеры создания линий и прямоугольников. Каждая вершина, созданная генератором примитивов, имеет определенную позицию в пространстве параметрических координат `(u, v, w)`. Каждая координата – это число в диапазоне от `0.0` до `1.0`. Координаты могут использоваться для определения местоположения вершины, нередко путем интерполяции на основе вершин примитива «заплаты». Алгоритмы генератора, которые производят вершины (и сопутствующие параметрические координаты), действуют немного по-разному. Алгоритмы тесселяции, результатом работы которых являются прямоугольники и линии, вычисляют только первые две параметрические координаты: `u` и `v`. На рис. 6.2 изображен процесс обработки входящей «заплаты» и получения на ее основе выходящей «заплаты» с четырьмя вершинами в обеих. Здесь генератор использует алгоритм тесселяции прямоугольника с внутренним и внешним уровнями тесселяции, равными четырем.

Число вершин во входящей «заплате» необязательно должно совпадать с числом вершин в выходящей «заплате», несмотря на то что во всех примерах в этой главе они совпадают.

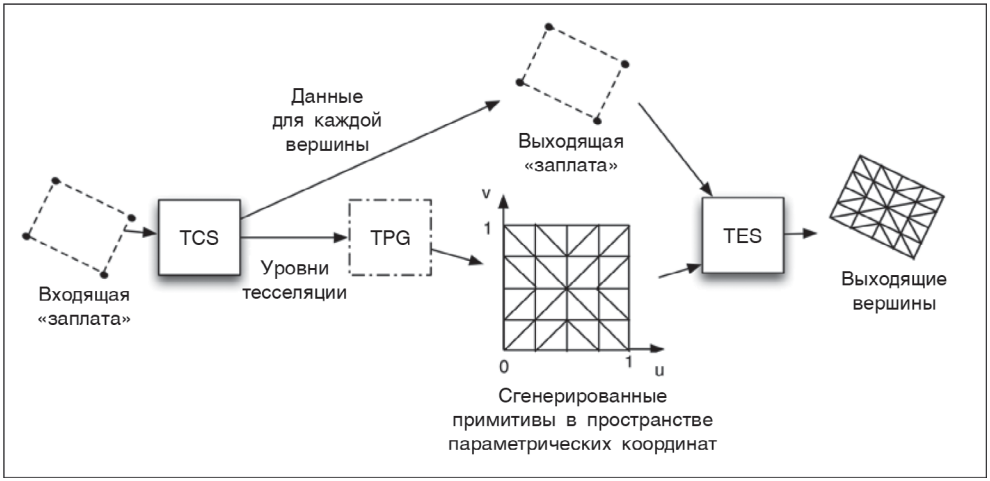

Рис. 6.2 ❖ Процесс обработки входящей «заплаты» и получения на ее основе выходящей «заплаты» с четырьмя вершинами

Шейдер выполнения тесселяции TES вызывается один раз для каждой вершины в параметрическом пространстве, созданной генератором примитивов TPG. Кажется немного странным, что шейдер выполнения тесселяции TES фактически определяет алгоритм, используемый генератором примитивов TPG. Это достигается посредством объявления входной переменной с квалификатором `layout`. Как отмечалось выше, главной задачей данного шейдера является определение позиций вершин (и любой другой информации, например векторов нормалей и координат текстуры). Обычно при этом шейдер выполнения тесселяции TES использует параметрические координаты `(u,v)`, полученные от генератора TPG, а также позиции всех вершин во входящей «заплате». Например, чтобы нарисовать кривую, «заплата» может содержать четыре вершины, являющиеся опорными точками кривой, а генератор TPG на их основе может создать 101 вершину (если уровень тесселяции установить равным 100), причем координаты `u` всех вершин будут равномерно располагаться в диапазоне от `0.0` до `1.0`. После этого шейдер выполнения тесселяции TES мог бы использовать координаты `u` вместе с координатами четырех вершин в исходной «заплате», чтобы определить позиции соответствующих вершин.

Если описание выше показалось вам непонятным, начните с рецепта «Тесселяция кривой» и двигайтесь дальше, к следующим рецептам.

В рецепте «Тесселяция кривой» представлен простой пример использования шейдеров тесселяции для рисования кривой Безье по четырем опорным точкам. В рецепте «Тесселяция двухмерного прямоугольника» мы попробуем разобраться с работой алгоритма тесселяции прямоугольников на примере отображения простого прямоугольника и треугольников, созданных генератором TPG. В рецепте «Тесселяция трехмерной поверхности» используется алгоритм тессляции прямо-

угольника для отображения трехмерной поверхности Безье. Наконец, в рецепте «Тесселяция с учетом глубины» мы увидим, как шейдеры тесселяции способны упростить реализацию алгоритмов отображения с изменяемыми уровнями детализации (Level-Of-Detail, LOD).

Отображение точечных спрайтов с помощью геометрического шейдера

Точечные спрайты (point sprites) – это простые квадраты (обычно с наложенной текстурой), ориентированные так, что лицевой стороной всегда смотрят в камеру. Их очень удобно использовать для реализации трехмерных систем частиц (см. главу 9 «Системы частиц и анимация») или двухмерных игр. Точечные спрайты определяются в приложениях OpenGL как простые точечные примитивы с режимом отображения `GL_POINTS`. Это упрощает процесс, потому что сам квадрат и координаты текстуры для квадрата определяются автоматически. В основном приложении точечные спрайты можно интерпретировать как простые точечные примитивы и тем самым избежать необходимости вычислений координат вершин квадратов.

На рис. 6.3 приводится скриншот с группой точечных спрайтов. Каждый спрайт отображается как точечный примитив. Квадрат и координаты текстуры определяются автоматически (внутри геометрического шейдера) и поворачиваются лицевой стороной к камере.

Рис. 6.3 ❖ Группа точечных спрайтов

OpenGL уже имеет встроенную поддержку точечных спрайтов в режиме отображения `GL_POINTS`. Когда отображение точечных примитивов выполняется в этом режиме, они рисуются как квадраты с длиной стороны, определяемой вызовом функции `glPointSize`. Кроме того, OpenGL автоматически будет генерировать координаты текстуры для фрагментов квадрата. Эти координаты изменяются в диапазоне значений от нуля до единицы в обоих направлениях (слева направо для `s` и снизу вверх для `t`) и доступны внутри фрагментного шейдера через встроенную переменную `gl_PointCoord`.

Существуют разные способы настройки отображения точечных спрайтов. Например, с помощью семейства функций `glPointParameter` можно определить начало системы координат для текстуры. С помощью того же семейства функций можно установить порядок определения значений альфа-канала для точек, когда включен режим множественной выборки.

Встроенная поддержка точечных спрайтов не позволяет программисту вращать квадраты или придавать им другую форму, например прямоугольную или треугольную. Однако подобные эффекты можно реализовать с помощью текстур и преобразований координат текстур. Например, координаты текстуры можно преобразовать с использованием матрицы поворота и придать спрайту вид повернутого объекта, даже при том что сама геометрия спрайта не изменится. Кроме того, размеры точечного спрайта необходимо регулировать в зависимости от значения глубины, если требуется получить эффект перспективы (когда спрайты уменьшаются в размерах с увеличением расстояния до них).

Если эти (и, возможно, другие) проблемы встроенной поддержки точечных спрайтов кажутся вам слишком ограничивающими, для их отображения можно воспользоваться геометрическим шейдером. Фактически данный прием является отличным примером, иллюстрирующим использование геометрического шейдера для создания разного рода примитивов, отличных от исходных. Основная идея состоит в том, чтобы с помощью геометрического шейдера изменить входящий примитив (в видимых координатах) и вывести квадрат с центром в заданной точке, повернутый лицевой стороной к камере. Геометрический шейдер может также автоматически вычислять координаты текстуры для квадрата.

При желании можно придавать спрайтам другую форму, например шестиугольника, или поворачивать квадраты перед их выводом из геометрического шейдера. Возможности бесконечны. Создание примитивов внутри геометрического шейдера дает нам непревзойденную гибкость, правда, ценой потери некоторой доли производительности. Встроенная поддержка точечных спрайтов сильно оптимизирована и в общем случае всегда обеспечивает наивысшую производительность.

Прежде чем перейти к программному коду, рассмотрим математический аппарат, лежащий в их основе. Внутри геометрического шейдера требуется сгенерировать вершины квадрата, с центром в заданной точке, в системе видимых координат. Пусть имеется точка *P*. Координаты углов квадрата можно вычислить путем простого переноса точки *P* в плоскости координатных осей X и Y, как показано на рис. 6.4.

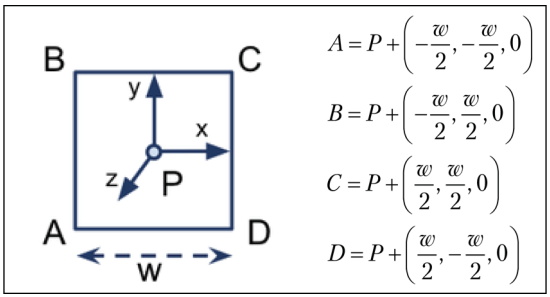

Рис. 6.4 ❖ Порядок вычисления координат углов квадрата

Геометрический шейдер в этом случае будет получать местоположение точки P в системе видимых координат и выводить квадрат как полосу из треугольников, с координатами текстуры. После этого фрагментный шейдер просто наложит текстуру на квадрат.

Подготовка

В этом примере мы попробуем отобразить множество точечных примитивов. Координаты будут передаваться через атрибут с индексом `0`. В данном случае не потребуется передавать векторы нормалей или координаты текстур.

Следующие uniform-переменные определяются внутри шейдеров и должны инициализироваться в основной программе:

- `Size2`: половина ширины квадрата спрайта;
- `SpriteTex`: текстурный слот, где хранится текстура для спрайта.

Как обычно, внутри шейдеров необходимо также определить uniform-переменные для стандартных матриц преобразований и инициализировать их в основной программе.

Как это делается...

Чтобы создать шейдерную программу для отображения точечных примитивов квадратной формы, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;

void main()
{
  gl_Position = ModelViewMatrix *
  vec4(VertexPosition,1.0);
}
```

2. Добавить геометрический шейдер:

```
layout( points ) in;
layout( triangle_strip, max_vertices = 4 ) out;

uniform float Size2; // Половина ширины квадрата
uniform mat4 ProjectionMatrix;

out vec2 TexCoord;

void main()
{
  mat4 m = ProjectionMatrix; // Чтобы сократить исходный код

  gl_Position = m * (vec4(-Size2,-Size2,0.0,0.0) +
                     gl_in[0].gl_Position);
  TexCoord = vec2(0.0,0.0);
  EmitVertex();

  gl_Position = m * (vec4(Size2,-Size2,0.0,0.0) +
                     gl_in[0].gl_Position);
  TexCoord = vec2(1.0,0.0);
  EmitVertex();

  gl_Position = m * (vec4(-Size2,Size2,0.0,0.0) +
                     gl_in[0].gl_Position);
  TexCoord = vec2(0.0,1.0);
  EmitVertex();

  gl_Position = m * (vec4(Size2,Size2,0.0,0.0) +
                     gl_in[0].gl_Position);
  TexCoord = vec2(1.0,1.0);
  EmitVertex();

  EndPrimitive();
}
```

3. Добавить фрагментный шейдер:

```
in vec2 TexCoord; // Поступает из геометрического шейдера

uniform sampler2D SpriteTex;

layout( location = 0 ) out vec4 FragColor;

void main()
{
  FragColor = texture(SpriteTex, TexCoord);
}
```

4. В функции отображения в основной программе вывести несколько точечных примитивов.

Как это работает...

Вершинный шейдер прост, насколько это вообще возможно. Он преобразует координаты точки в систему видимых координат умножением на матрицу моделивида и присваивает результат встроенной выходной переменной `gl_Position`.

Геометрический шейдер начинается с определения типа примитива, который он ожидает получить. Первая инструкция `layout` сообщает, что шейдер будут принимать точечные (`point`) примитивы.

```
layout( points ) in;
```

Следующая инструкция `layout` определяет тип примитивов, генерируемых геометрическим шейдером, и максимальное число вершин в примитиве.

```
layout( triangle_strip, max_vertices = 4 ) out;
```

В данном случае для каждой полученной точки требуется вывести единственный квадрат, поэтому здесь указывается, что выходным примитивом будет полоса треугольников с максимальным числом вершин, равным четырем.

Входящий примитив доступен внутри фрагментного шейдера через встроенную переменную `gl_in`. Имейте в виду, что это – массив структур. Возможно, кому-то покажется странным, что точечный примитив, представленный единственной точкой, передается в виде массива. Однако в этом нет ничего необычного, если вспомнить, что геометрический шейдер способен принимать не только точки (возможно, с сопутствующей информацией), но также треугольники и отрезки. Соответственно, число значений может быть больше одного. Если, к примеру, геометрическому шейдеру будут передаваться треугольники, он будет получать три входных значения (по одному для каждой вершины). Фактически при передаче примитивов типа `triangles_adjacency` он будет получать шесть значений (подробнее об этом рассказывается в последнем рецепте).

Переменная `gl_in` – это массив структур. Каждая структура содержит следующие поля: `gl_Position`, `gl_PointSize` и `gl_ClipDistance[]`. В данном примере нас интересует только поле `gl_Position`. Однако мы могли бы присваивать значения другим полям в вершинном шейдере для передачи дополнительной информации.

В функции `main` геометрического шейдера определяется квадрат (в виде полосы треугольников), как описывается ниже. Для каждой вершины в полосе треугольников выполняются следующие шаги:

1. Вычисляются атрибуты вершины (в данном случае координаты местоположения и координаты текстуры) и присваиваются соответствующим выходным переменным (`gl_Position` и `TexCoord`). Обратите внимание, что координаты также преобразуются с помощью матрицы проекции. Делается это потому, что переменная `gl_Position` должна содержать координаты в системе координат отсечения (clip coordinates), необходимые для последующих стадий в конвейере.
2. Производится вывод вершины (передача дальше по конвейеру) вызовом встроенной функции `EmitVertex()`.

После вывода всех вершин выходного примитива вызывается функция `EndPrimitive()`, завершающая вывод примитива и отправляющая его дальше.

В данном случае можно было бы и не вызывать функцию `EndPrimitive()`, потому что она неявно вызывается по завершении геометрического шейдера. Однако, как и при работе с файлами, никогда нелишним будет явно обозначить окончание работы с примитивом.

Фрагментный шейдер тоже очень прост. Он просто накладывает текстуру на фрагмент, используя (интерполированные) координаты текстуры, полученные от геометрического шейдера.

И еще...

Этот пример чрезвычайно прост и задумывался с целью облегчить знакомство с геометрическими шейдерами. Его можно было бы дополнить поддержкой поворота квадратов или ориентацией их в разных направлениях. Также можно было бы реализовать отбрасывание фрагментов с помощью текстуры (внутри фрагментного шейдера), чтобы получить возможность создавать точечные спрайты произвольной формы. Мощь геометрического шейдера открывает перед нами огромное число возможностей!

Наложение каркаса на освещенную поверхность

В предыдущем рецепте демонстрировалось использование геометрического шейдера для создания примитивов, отличных от исходных. Геометрические шейдеры способны также передавать дополнительную информацию последующим стадиям. Они отлично подходят для этого, потому что обладают доступом сразу ко всем вершинам примитива и могут производить вычисления для всего примитива, а не только для одной вершины.

Данный рецепт демонстрирует пример геометрического шейдера, который не изменяет входящих примитивов – он передает их дальше в неизменном виде, но при этом вычисляет дополнительную информацию о треугольнике для использования внутри фрагментного шейдера с целью выделения границ полигона. Идея примера состоит в том, чтобы нарисовать границы каждого полигона на освещенной поверхности.

На рис. 6.5 изображен результат применения описываемого приема. На поверхности объекта отображается ее каркас, для чего используется информация, сгенерированная внутри геометрического шейдера.

Существует множество приемов отображения каркаса на поверхности. Данный прием заимствован из статьи на сайте NVIDIA, опубликованной в 2007 году. Здесь используется геометрический шейдер, чтобы получить изображение каркаса за один проход. В данном примере также реализовано сглаживание линий каркаса, дабы результат выглядел более опрятным (см. рис. 6.5).

Чтобы отобразить каркас на поверхности, вычисляется расстояние от каждого фрагмента до ближайшей стороны треугольника. Когда фрагмент оказывается на

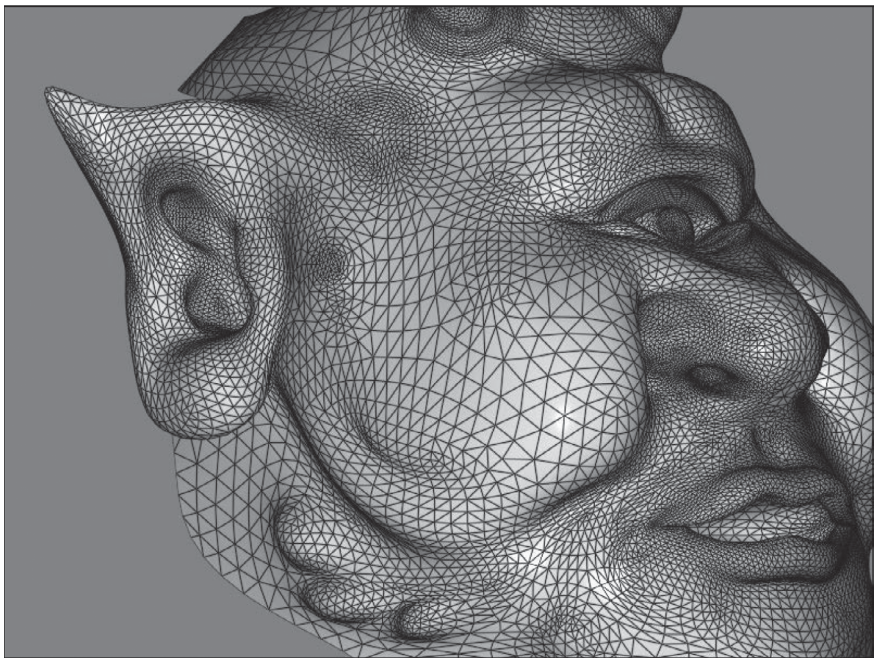

Рис. 6.5 ❖ Результат наложения каркаса на поверхность

определенном расстоянии, его цвет смешивается с цветом границы. В противном случае фрагмент отображается как обычно.

Вычисление расстояния от фрагмента до сторон производится следующим образом: в геометрическом шейдере вычисляется минимальное расстояние от каждой вершины до противоположной стороны (называется **высотой треугольника**). На рис. 6.6 эти расстояния показаны как **ha**, **hb** и **hc**.

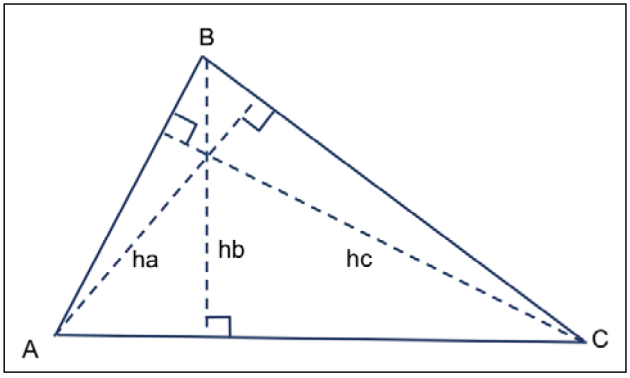

Рис. 6.6 ❖ Минимальные расстояния от вершин треугольника до противоположных сторон

Те же самые расстояния можно вычислить по трем сторонам треугольника, используя закон косинусов. Например, чтобы найти высоту **ha**, используется угол β при вершине *C*.

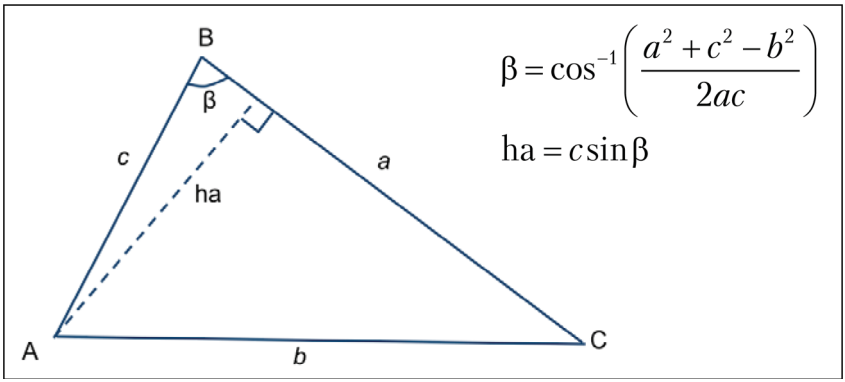

Рис. 6.7 ❖ Определение высоты ha по углу β при вершине *C*

Остальные высоты вычисляются аналогично. (Обратите внимание, что угол β может быть больше 90°, в этом случае могло бы появиться желание вычислить синус угла (180 – β). Однако синус угла (180 – β) равен синусу угла β.)

После получения высот треугольника внутри геометрического шейдера можно создать выходной вектор (вектор «расстояний до сторон»), который будет интерполироваться по всей площади треугольника. Компоненты этого вектора представляют расстояния от фрагмента до каждой из сторон треугольника. Компонент `x` представляет расстояние до стороны *a*, компонент `y` – расстояние до стороны *b*, и компонент `z` – до стороны *c*. Если присвоить этим компонентам правильные исходные значения, аппаратура графической карты автоматически будет интерполировать их и передавать соответствующие расстояния для каждого фрагмента. В вершине *A* этот вектор должен иметь значение (**ha**, 0, 0), потому что вершина *A* находится на расстоянии **ha** от стороны *a* и непосредственно на сторонах *b* и *c*. Аналогично для вершины *B* вектор должен иметь значение (0, **hb**, 0), и для вершины *C* – значение (0, 0, **hc**). При интерполяции этих трех значений по площади треугольника должны получаться расстояния от фрагмента до каждой из сторон.

Вычисления будут производиться в экранных координатах. То есть перед вычислением высот в геометрическом шейдере координаты вершин будут приводиться к экранным координатам. Так как все операции будут выполняться в системе экранных координат, нельзя интерполировать значения в перспективе (это было бы неправильно). Поэтому нужно сообщить аппаратуре, что интерполяция должна выполняться линейно.

Внутри фрагментного шейдера остается только найти минимальное из трех расстояний и, если это расстояние меньше толщины линии, смешать цвет фрагмента с цветом линий. Однако нам хотелось бы также выполнить сглаживание. Для это-

го воспроизводится эффект постепенного исчезновения края линии с помощью функции `smoothstep` в языке GLSL.

Интенсивность цвета линии масштабируется на расстоянии двух пикселей от ее центра. Пиксели, оказавшиеся на расстоянии меньше одного пикселя от истинного центра линии, получают 100%-ный цвет линии, а пиксели, оказавшиеся на расстоянии одного или более пикселей от края линии, получают 100%-ный цвет фрагмента. Цвет для пикселей, занимающих промежуточные позиции, вычисляется с помощью функции `smoothstep`, благодаря чему создается эффект плавного перехода цветов. Конечно же, толщина линий является настраиваемым параметром (`Line.Width`).

Подготовка

Для этого примера достаточно типовой конфигурации. Координаты вершин и векторы нормалей должны передаваться в атрибутах с индексами `0` и `1` соответственно, и необходимо также предусмотреть передачу параметров освещения в соответствии с выбранной моделью. Как обычно, стандартные матрицы должны инициализироваться в основном приложении и передаваться через uniform-переменные. Однако отметьте, что на этот раз необходимо также передать матрицу преобразования в экранные координаты (viewport matrix) через uniform-переменную `ViewportMatrix`, чтобы обеспечить преобразование в систему экранных координат.

Ниже перечислены uniform-переменные, определяющие параметры отображения линий, которые также необходимо инициализировать в основном приложении:

- `Line.Width`: хранит половинную толщину линий каркаса;
- `Line.Color`: цвет линий каркаса.

Как это делается...

Чтобы создать шейдерную программу для отображения каркаса на поверхности, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0 ) in vec3 VertexPosition;
layout (location = 1 ) in vec3 VertexNormal;

out vec3 VNormal;
out vec3 VPosition;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
  VNormal = normalize( NormalMatrix * VertexNormal);
```

```
  VPosition = vec3(ModelViewMatrix *
                   vec4(VertexPosition,1.0));
  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить геометрический шейдер:

```
layout( triangles ) in;
layout( triangle_strip, max_vertices = 3 ) out;

out vec3 GNormal;
out vec3 GPosition;
noperspective out vec3 GEdgeDistance;

in vec3 VNormal[];
in vec3 VPosition[];

uniform mat4 ViewportMatrix; // Матрица преобразования
                             // в экранные координаты
void main()
{
  // Преобразовать вершину в систему экранных координат
  vec3 p0 = vec3(ViewportMatrix * (gl_in[0].gl_Position /
                                   gl_in[0].gl_Position.w));
  vec3 p1 = vec3(ViewportMatrix * (gl_in[1].gl_Position /
                                   gl_in[1].gl_Position.w));
  vec3 p2 = vec3(ViewportMatrix * (gl_in[2].gl_Position /
                                   gl_in[2].gl_Position.w));

  // Вычислить атрибуты (ha, hb и hc)
  float a = length(p1 - p2);
  float b = length(p2 - p0);
  float c = length(p1 - p0);
  float alpha = acos( (b*b + c*c - a*a) / (2.0*b*c) );
  float beta = acos( (a*a + c*c - b*b) / (2.0*a*c) );
  float ha = abs( c * sin( beta ) );
  float hb = abs( c * sin( alpha ) );
  float hc = abs( b * sin( alpha ) );

  // Отправить треугольник с расстояниями до сторон
  GEdgeDistance = vec3( ha, 0, 0 );
  GNormal = VNormal[0];
  GPosition = VPosition[0];
  gl_Position = gl_in[0].gl_Position;
  EmitVertex();

  GEdgeDistance = vec3( 0, hb, 0 );
  GNormal = VNormal[1];
  GPosition = VPosition[1];
  gl_Position = gl_in[1].gl_Position;
  EmitVertex();

  GEdgeDistance = vec3( 0, 0, hc );
  GNormal = VNormal[2];
```

```
  GPosition = VPosition[2];
  gl_Position = gl_in[2].gl_Position;
  EmitVertex();

  EndPrimitive();
}
```

3. Добавить фрагментный шейдер:

```
// *** Добавить соответствующие объявления uniform-переменных
//     для модели затенения по Фонгу ***

// Параметры линий каркаса
uniform struct LineInfo {
  float Width;
  vec4 Color;
} Line;

in vec3 GPosition;

in vec3 GNormal;
noperspective in vec3 GEdgeDistance;

layout( location = 0 ) out vec4 FragColor;
vec3 phongModel( vec3 pos, vec3 norm )
{
  // *** Здесь выполняются вычисления в соответствии
  //     с моделью затенения по Фонгу ***
}

void main() {
  // Цвет поверхности.
  vec4 color=vec4(phongModel(GPosition, GNormal), 1.0);

  // Найти наименьшее расстояние
  float d = min( GEdgeDistance.x, GEdgeDistance.y );
  d = min( d, GEdgeDistance.z );

  // Определить пропорцию смешивания с цветом линии
  float mixVal = smoothstep( Line.Width - 1,
                             Line.Width + 1, d );

  // Смешать цвет поверхности с цветом линии
  FragColor = mix( Line.Color, color, mixVal );
}
```

Как это работает...

Вершинный шейдер весьма прост. Он переводит позицию вершины и вектор нормали в систему видимых координат (с центром в точке местоположения камеры) и передает их дальше геометрическому шейдеру. Встроенной переменной `gl_Position` присваивается местоположение в усеченной системе координат –

это значение будет использоваться геометрическим шейдером для определения экранных координат.

Геометрический шейдер начинается с определения типов входящих и выходящих примитивов.

```
layout( triangles ) in;
layout( triangle_strip, max_vertices = 3 ) out;
```

Фактически геометрия треугольников не изменяется в шейдере, поэтому входящие и выходящие примитивы имеют, по сути, один и тот же тип. На выходе шейдера будет получен тот же самый треугольник, что пришел в него.

Выходными переменными геометрического шейдера являются `GNormal`, `GPosition` и `GEdgeDistance`. Первые две хранят координаты вершины и вектор нормали в системе видимых координат и передаются дальше без изменений. Третья является вектором, в котором хранятся расстояния до сторон треугольника (как описывалось выше). Обратите внимание, что она определена с квалификатором `noperspective`.

```
noperspective out vec3 GEdgeDistance;
```

Квалификатор `noperspective` указывает, что значение этой переменной должно интерполироваться по линейному закону, то есть без учета перспективы видимой области. Как отмечалось прежде, эти расстояния измеряются в системе экранных координат, поэтому было бы неправильно интерпретировать их нелинейно.

Внутри функции `main` каждая из трех вершин треугольника сначала преобразуется из усеченной системы координат в экранную умножением на матрицу преобразования в экранные координаты (viewport matrix). (Обратите внимание, что координаты X, Y и Z необходимо разделить на координату `w`, потому что усеченные координаты являются однородными и их может потребоваться преобразовать обратно в истинные декартовы координаты.)

Далее вычисляются три высоты – `ha`, `hb` и `hc` – по формуле закона косинусов. После этого значения, соответствующие первой вершине, присваиваются переменным `GEdgeDistance`, `GNormal`, `GPosition` и `gl_Position` и передаются дальше вызовом `EmitVertex()`. На этом процесс обработки первой вершины заканчивается. После этого аналогичным образом обрабатываются две другие вершины треугольника и затем, вызовом `EndPrimitive()`, завершается обработка всего полигона.

Внутри фрагментного шейдера сначала выполняются вычисления в соответствии с моделью освещения, и полученный цвет сохраняется в переменной `color`. На этом этапе три компонента вектора `GEdgeDistance` должны содержать расстояния от данного фрагмента до сторон треугольника. Нас интересует минимальное расстояние, поэтому шейдер определяет минимальное значение из трех компонентов и сохраняет его в переменной `d`. Затем с помощью функции `smoothstep` определяется пропорция смешивания цвета линии с цветом фрагмента (`mixVal`).

```
float mixVal = smoothstep( Line.Width - 1,
                           Line.Width + 1, d );
```

Если расстояние оказывается меньше, чем `Line.Width - 1`, функция `smoothstep` вернет значение `0`, а если больше, чем `Line.Width + 1`, — она вернет `1`. Для любых других значений `d` мы получим плавный переход. То есть для фрагмента, лежащего на линии, функция `smoothstep` вернет `0`, для фрагмента, лежащего за пределами линии, она вернет `1`, и для фрагмента, лежащего в 2-пиксельной области, окружающей линию, она вернет значение в диапазоне от `0` до `1`, в зависимости от близости к линии. Таким образом, результат, возвращаемый функцией, можно использовать непосредственно для смешивания цвета фрагмента с цветом линии.

В заключение определяется цвет фрагмента путем смешивания его с цветом линии в пропорции `mixVal`.

И еще...

Этот прием дает визуально привлекательный результат и имеет относительно немного недостатков. Он является отличным примером использования геометрического шейдера для решения задач, не связанных с изменением исходной геометрии. В данном случае геометрический шейдер используется, только чтобы вычислить дополнительную информацию о примитиве, по мере его продвижения по конвейеру.

Данный шейдер можно применить к любой поверхности, не изменяя основного приложения. Его можно использовать для отладки или задействовать в программах графического моделирования.

Другие распространенные приемы, позволяющие получить подобный эффект, осуществляют наложение каркаса на поверхность в два прохода со смещением полигонов (с использованием функции `glPolygonOffset`), чтобы избежать «Z-конфликтов», которые могут возникать между каркасом и поверхностью под ним. Этот прием не всегда достаточно эффективен, потому что измененные значения глубины не всегда получаются правильными, и иногда бывает сложно найти «оптимальное» значение смещения полигона. Неплохой обзор приемов можно найти в разделе 11.4.2 в книге Томаса Акенина-Мюллера (Tomas Akenine-Möller), Эрика Хайнса (Eric Haines) и Натаниэля Хоффманна (Nathaniel Hoffmann) «Real Time Rendering. Third edition», AK Peters, 2008.

См. также...

- Впервые описание этого приема было опубликовано в статье на сайте NVIDIA в 2007 году («Solid Wireframe», NVIDIA Whitepaper WP-03014-001_v01: **http://developer.nvidia.com**). Статья позиционируется как описание реализации для Direct3D, но наша реализация, естественно, создана для OpenGL.
- Рецепт «Создание теней с использованием приема теневых объемов и геометрического шейдера» в главе 7 «Тени».
- Рецепт «Фоновый, рассеянный и отраженный свет» в главе 2 «Основы шейдеров GLSL».

Рисование линий силуэта с помощью геометрического шейдера

Когда требуется получить эффект карандашного рисунка, часто бывает желательно обвести черным контуры модели, а также выделить складки и выпуклости (то есть выделить силуэт). В данном рецепте мы обсудим один из приемов, позволяющих получить такой эффект, основанный на использовании геометрического шейдера. Геометрический шейдер будет воспроизводить линии силуэта, генерируя маленькие, узкие прямоугольники, расположенные по краям объекта.

На рис. 6.8 показано изображение лица огра с черными силуэтными линиями, полученными с помощью геометрического шейдера. Линии состоят из маленьких прямоугольников, расположенных по краям и складкам.

Рис. 6.8 ❖ Изображение лица огра с черными силуэтными линиями

Прием, демонстрируемый в этом рецепте, основан на методике, описанной в недавней статье Филиппа Ридеоута (Philip Rideout) (**http://prideout.net/blog/?p=54**). Его реализация использует два прохода (для базовой геометрии и силуэта) и включает множество оптимизаций, таких как сглаживание и нестандартный способ проверки глубины (с помощью g-буферов). Чтобы упростить пример, так как главной его целью является демонстрация возможностей геометрического шейдера, реализация этого приема будет выполнять один проход, без сглаживания и нестандартного способа проверки глубины. Интересующиеся этими оптимизациями могут заглянуть в статью Филиппа.

Одной из важнейших особенностей геометрического шейдера является возможность передачи дальше по конвейеру дополнительной информации. Вместе с геометрическим шейдером в OpenGL были введены также дополнительные режимы отображения примитивов. Эти «смежные» режимы позволяют включать в примитивы дополнительные вершины, обычно относящиеся сразу к группе смежных примитивов, но это совершенно необязательно (в действительности дополнительную информацию можно использовать для любых целей). В следующем списке перечислены смежные режимы отображения с краткими описаниями:

- `GL_LINES_ADJACENCY`: определяет отрезок, соединяющий смежные вершины (каждый отрезок задается четырьмя вершинами);
- `GL_LINE_STRIP_ADJACENCY`: определяет ломаную линию, соединяющую смежные вершины (для n отрезков нужно задать $(n + 3)$ вершин);
- `GL_TRIANGLES_ADJACENCY`: определяет треугольник, соединяющий вершины смежных треугольников (на каждый примитив требуется определить шесть вершин);
- `GL_TRIANGLE_STRIP_ADJACENCY`: определяет полосу треугольников, соединяющих вершины смежных треугольников (для n треугольников нужно задать $2(n + 2)$ вершин).

Полное описание этих режимов можно найти в официальной документации OpenGL. В данном рецепте используется режим `GL_TRIANGLES_ADJACENCY` с предоставлением информации о смежных треугольниках. В этом режиме для каждого примитива требуется задать шесть вершин. На рис. 6.9 показано, как должны располагаться эти вершины.

На рис. 6.9 сплошными линиями изображен генерируемый треугольник, а пунктирными линиями – смежные треугольники. Первая, третья и пятая вершины (**0**, **2** и **4**) образуют искомый треугольник. Вторая, четвертая и шестая вершины (**1**, **3** и **5**) принадлежат смежным треугольникам.

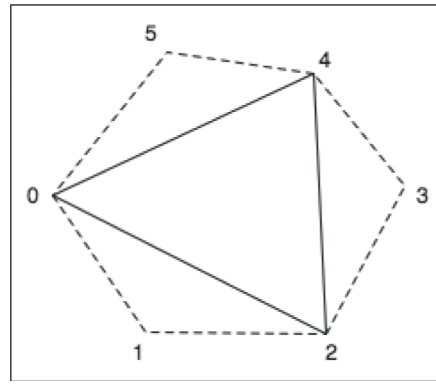

Рис. 6.9 ❖ Порядок расположения смежных треугольников для создания примитива в режиме GL_TRIANGLES_ADJACENCY

Информация о вершинах обычно не передается в таком виде, поэтому нужно предварительно обработать отображаемую поверхность и включить в нее информацию о дополнительных вершинах. Часто для этого требуется в два раза увеличить массив индексов. Массивы с вершинами, векторами нормалей и координатами текстуры при этом не изменяются.

Когда выполняется отображение в одном из смежных режимов, геометрический шейдер получает доступ ко всем шести вершинам, связанным с данным треугольником. В результате с помощью смежного треугольника можно определить, лежит ли сторона треугольника на границе силуэта объекта: сторона треугольника считается лежащей на границе силуэта, если треугольник участвует в образовании лицевой поверхности, а соответствующий смежный треугольник – нет.

Определить, участвует ли треугольник в образовании лицевой поверхности, можно, вычислив вектор нормали треугольника (как векторное произведение). При работе в системе видимых (или усеченных) координат координата Z вектора нормали для лицевых треугольников будет иметь положительное значение. То есть достаточно вычислить лишь координату Z вектора нормали, благодаря чему можно сэкономить несколько тактов процессорного времени. Для треугольника с вершинами A, B и C значение координаты Z вектора нормали определяется следующим уравнением:

$$n_z = (A_xB_y - B_xA_y) + (B_xC_y - C_xB_y) + (C_xA_y - A_xC_y).$$

Выяснив, какие стороны лежат на границах силуэта, геометрический шейдер будет генерировать дополнительные узкие прямоугольники. Все вместе эти прямоугольники будут образовывать линии силуэта (см. рис. 6.8). После создания и вывода прямоугольников геометрический шейдер будет выводить оригинальный треугольник.

Чтобы выполнить отображение поверхности в один проход, без наложения эффекта освещения/затенения на линии силуэта, будет использоваться дополнительная выходная переменная. С ее помощью фрагментный шейдер сможет определить, когда отображается сама поверхность, а когда линия силуэта.

Подготовка

Подготовьте исходные данные о поверхности, включив в них информацию о смежных вершинах. Как только что говорилось, для этого нужно увеличить массив индексов и сохранить в нем дополнительную информацию. Сделать это можно, выполнив обход треугольников и определив общие стороны. Для экономии места в книге я не буду описывать эту процедуру во всех тонкостях – всю необходимую информацию можно найти в статье Филиппа Ридеоута, упомянутой выше. Отметьте также, что исходный код данного примера реализует простейший (и не самый эффективный) прием.

Ниже перечислены некоторые важные uniform-переменные, используемые в данном примере:

- `EdgeWidth`: толщина линий силуэта в усеченных (нормализованных) координатах;

- `PctExtend`: процент, определяющий, насколько нужно удлинить прямоугольники линии силуэта;
- `LineColor`: цвет линий силуэта.

Как обычно, необходимо также определить uniform-переменные для передачи параметров освещения и стандартных матриц.

Как это делается...

Чтобы создать шейдерную программу для отображения линий силуэта с помощью геометрического шейдера, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0 ) in vec3 VertexPosition;
layout (location = 1 ) in vec3 VertexNormal;

out vec3 VNormal;
out vec3 VPosition;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;
void main()
{
  VNormal = normalize( NormalMatrix * VertexNormal);
  VPosition = vec3(ModelViewMatrix *
                   vec4(VertexPosition,1.0));
  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Добавить геометрический шейдер:

```
layout( triangles_adjacency ) in;
layout( triangle_strip, max_vertices = 15 ) out;

out vec3 GNormal;
out vec3 GPosition;

// Признак принадлежности примитива линии силуэта
flat out bool GIsEdge;

in vec3 VNormal[];       // Нормаль в видимых координатах
in vec3 VPosition[];     // Позиция в видимых координатах
uniform float EdgeWidth; // Толщина линий силуэта в усеч. коорд.
uniform float PctExtend; // Процент превышения длины

bool isFrontFacing( vec3 a, vec3 b, vec3 c )
{
  return ((a.x * b.y - b.x * a.y) +
          (b.x * c.y - c.x * b.y) +
          (c.x * a.y - a.x * c.y)) > 0;
}

void emitEdgeQuad( vec3 e0, vec3 e1 )
```

```
{
  vec2 ext = PctExtend * (e1.xy - e0.xy);
  vec2 v = normalize(e1.xy - e0.xy);
  vec2 n = vec2(-v.y, v.x) * EdgeWidth;

  // Вывести прямоугольник
  GIsEdge = true; // Принадлежит линии силуэта

  gl_Position = vec4( e0.xy - ext, e0.z, 1.0 );
  EmitVertex();
  gl_Position = vec4( e0.xy - n - ext, e0.z, 1.0 );
  EmitVertex();
  gl_Position = vec4( e1.xy + ext, e1.z, 1.0 );
  EmitVertex();
  gl_Position = vec4( e1.xy - n + ext, e1.z, 1.0 );
  EmitVertex();

  EndPrimitive();
}

void main()
{
  vec3 p0 = gl_in[0].gl_Position.xyz /
            gl_in[0].gl_Position.w;
  vec3 p1 = gl_in[1].gl_Position.xyz /
            gl_in[1].gl_Position.w;
  vec3 p2 = gl_in[2].gl_Position.xyz /
            gl_in[2].gl_Position.w;
  vec3 p3 = gl_in[3].gl_Position.xyz /
            gl_in[3].gl_Position.w;
  vec3 p4 = gl_in[4].gl_Position.xyz /
            gl_in[4].gl_Position.w;
  vec3 p5 = gl_in[5].gl_Position.xyz /
            gl_in[5].gl_Position.w;

  if( isFrontFacing(p0, p2, p4) ) {
    if( ! isFrontFacing(p0,p1,p2) )
                emitEdgeQuad(p0,p2);
    if( ! isFrontFacing(p2,p3,p4) )
                emitEdgeQuad(p2,p4);
    if( ! isFrontFacing(p4,p5,p0) )
                emitEdgeQuad(p4,p0);
  }

  // Вывести оригинальный треугольник
  GIsEdge = false; // Треугольник не принадлежит линии силуэта

  GNormal = VNormal[0];
  GPosition = VPosition[0];
  gl_Position = gl_in[0].gl_Position;
  EmitVertex();

  GNormal = VNormal[2];
  GPosition = VPosition[2];
  gl_Position = gl_in[2].gl_Position;
```

```
  EmitVertex();

  GNormal = VNormal[4];
  GPosition = VPosition[4];
  gl_Position = gl_in[4].gl_Position;
  EmitVertex();

  EndPrimitive();
}
```

3. Добавить фрагментный шейдер:

```
// *** Объявить здесь переменные с параметрами освещения и
//     свойствами материала ****

uniform vec4 LineColor; // Цвет линий силуэта

in vec3 GPosition;      // Позиция в видимых координатах
in vec3 GNormal;        // Нормаль в видимых координатах

flat in bool GIsEdge;   // Признак принадлежности линии силуэта

layout( location = 0 ) out vec4 FragColor;

vec3 toonShade( )
{
  // *** реализация алгоритма создания
  //     "мультяшного" эффекта из главы 3 ***
}

void main()
{
  // При рисовании линии силуэта использовать постоянный цвет,
  // иначе - вычислить цвет, исходя из освещенности
  if( GIsEdge ) {
    FragColor = LineColor;
  } else {
    FragColor = vec4( toonShade(), 1.0 );
  }
}
```

Как это работает...

Вершинный шейдер играет роль простого «передаточного звена». Он преобразует координаты вершины и вектор нормали в систему видимых координат и отправляет их дальше, через переменные `VPosition` и `VNormal`. Они будут использоваться для вычисления освещенности внутри фрагментного шейдера и обрабатываться (или игнорироваться) геометрическим шейдером. Позиция вершины также преобразуется в усеченные координаты (или нормализованные координаты устройства) умножением на матрицу модели вида проекции и затем присваивается встроенной переменной `gl_Position`.

Геометрический шейдер начинается с определения типов входящего и исходящего примитивов:

```
layout( triangles_adjacency ) in;
layout( triangle_strip, max_vertices = 15 ) out;
```

Эти инструкции сообщают, что геометрический шейдер получает треугольники с информацией о смежных вершинах и выводит полосы из треугольников. Данный геометрический шейдер будет производить единственный треугольник (оригинальный) и не более одного прямоугольника для каждой его стороны. То есть на выходе может получаться до 15 вершин, о чем и сообщает квалификатор `layout`, описывающий выходной примитив.

Выходная переменная `GIsEdge` используется как флаг, сообщающий фрагментному шейдеру, является ли текущий полигон прямоугольником, лежащим на линии силуэта. Анализируя этот флаг, фрагментный шейдер будет принимать решение о необходимости вычисления освещенности фрагмента. Значение флага не должно интерполироваться (бессмысленно интерполировать логическое значение), поэтому в объявлении переменной использован квалификатор `flat`.

Первые несколько строк в функции `main` извлекают координаты всех шести вершин (в усеченных координатах) и делят их на четвертую координату, чтобы перейти от гомогенного представления к истинным декартовым координатам. Это необходимо на тот случай, если используется перспективная проекция, – в ортогональной проекции деление выполнять не требуется.

Далее проверяется, лежит ли оригинальный треугольник (определяется вершинами `0`, `2` и `4`) на лицевой стороне. Эту проверку выполняет функция `isFrontFacing`, используя уравнение, описанное выше. Если треугольник лежит на лицевой стороне, выводится прямоугольник линии силуэта, но только если смежный треугольник не лежит на лицевой стороне.

Функция `emitEdgeQuad` выводит прямоугольник, одной стороной которого является отрезок, определяемый точками `e0` и `e1`. Она начинается с вычисления `ext` – вектора от `e0` до `e1`, масштабированного значением `PctExtend` (чтобы немного удлинить прямоугольник). Удлинение необходимо, чтобы устранить пропуски, которые могут появиться между прямоугольниками (эта проблема обсуждается в разделе «И еще...» ниже).

Отметьте также, что здесь не учитывается координата Z. Так как точки определены в усеченных координатах и наша цель – нарисовать прямоугольник в плоскости X–Y, достаточно будет вычислить координаты X и Y вершин простым перемещением в плоскости X–Y. Поэтому можно смело игнорировать координату Z – ее значение останется неизменным в получившихся вершинах.

Далее переменной `v` присваивается нормализованный вектор из точки `e0` в точку `e1`. Переменной `n` присваивается вектор, перпендикулярный вектору `v` (в двухмерной проекции это достигается путем обмена значениями координат X и Y и изменения знака новой координаты X). Это самая обычная операция поворота двухмерной плоскости на 90° по часовой стрелке. Затем вектор `n` масштабируется значением `EdgeWidth`, потому что его длина должна совпадать с шириной прямоугольника. Векторы `ext` и `n` будут использоваться для определения вершин прямоугольника, как показано на рис. 6.10.

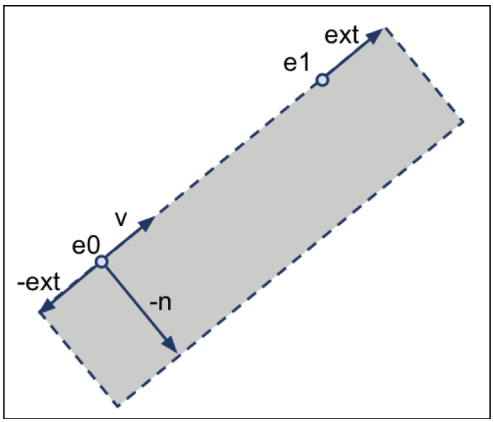

Рис. 6.10 ❖ Определение вершин прямоугольника

Координаты четырех вершин прямоугольника определяются как **e0 – ext**, **e0 – – n – ext**, **e1 + ext** и **e1 – n + ext**. Координата Z двух нижних вершин равна координате Z вершины **e0**, а координата Z двух верхних вершин равна координате Z вершины **e1**.

Заканчивается функция `emitEdgeQuad` присваиванием переменной `GIsEdge` значения `true`, чтобы фрагментный шейдер интерпретировал прямоугольник как элемент линии силуэта, и отправкой четырех вершин прямоугольника. В заключение вызывается функция `EndPrimitive`, завершающая обработку треугольников, составляющих прямоугольник.

После вывода прямоугольника линии силуэта функция `main` выводит оригинальный треугольник в неизменном виде. Значения `VNormal`, `VPosition` и `gl_Position` для вершин `0`, `2` и `4` передаются фрагментному шейдеру без каких-либо изменений. Каждая вершина выводится вызовом `EmitVertex`, и вывод всего примитива завершается вызовом `EndPrimitive`.

Для определения цвета фрагмента внутри фрагментного шейдера используется либо модель освещенности, воспроизводящая «мультяшный» эффект, либо константа. Выбор определяется значением входной переменной `GIsEdge`. Если `GIsEdge` имеет значение `true`, для фрагмента выбирается цвет линии силуэта. В противном случае считается, что отображается фрагмент поверхности, и цвет для него вычисляется в соответствии с моделью, описанной в главе 3 «Освещение, затенение и оптимизация».

И еще...

Один из недостатков только что описанного приема заключается в возможности появления «разрывов» между соседними прямоугольниками.

На рис. 6.11 показан эффект появления разрывов в линии силуэта. Разрывы между прямоугольниками можно было бы заполнить треугольниками, но в дан-

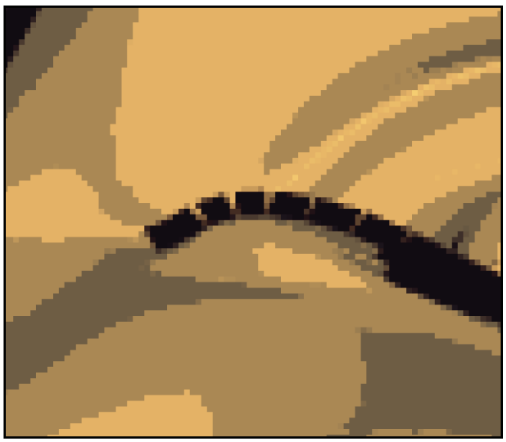

Рис. 6.11 ❖ Разрывы между соседними прямоугольниками, составляющими линию силуэта

ном примере было решено просто удлинить каждый прямоугольник. Конечно, если удлинить прямоугольники слишком сильно, появится другой эффект – «зубцы», но, как показывает практика, они заметны куда меньше и не мешают восприятию изображения.

Второй недостаток связан с механизмом проверки глубины. Если край полигона «наезжает» на соседнюю область, он может быть обрезан механизмом проверки глубины. Этот эффект демонстрируется на рис. 6.12.

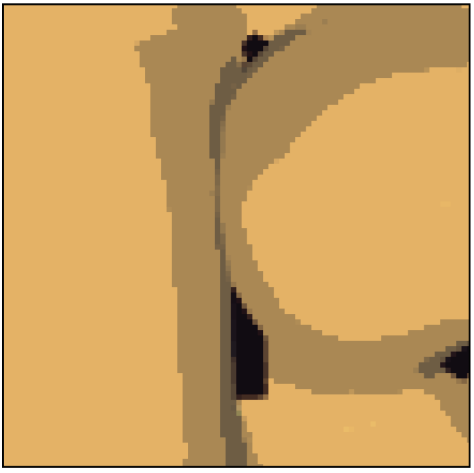

Рис. 6.12 ❖ Эффект обрезания края полигона механизмом проверки глубины

Здесь черная линия силуэта должна продолжаться вверх, пересекая центральную часть изображения, но была обрезана из-за того, что оказалась дальше выпуклости справа. Эту проблему можно решить, используя собственный механизм проверки глубины при отображении линий силуэта. Подробное описание этого приема можно найти в статье, упоминавшейся выше. Также можно вообще отключить проверку глубины при выводе линий силуэта, но при этом придется побеспокоиться о том, чтобы не нарисовать линии, находящиеся позади и потому действительно невидимые.

См. также

- Статью с описанием реализации эффекта ворсистой поверхности на основе геометрического шейдера: **http://developer.download.nvidia.com/whitepapers/2007/SDK10/FurShellsAndFins.pdf**[1].
- Рецепт «Создание теней с использованием приема теневых объемов и геометрического шейдера» в главе 7 «Тени».
- Рецепт «Придание изображению «мультяшного» вида» в главе 3 «Освещение, затенение и оптимизация».

Тесселяция кривой

В данном рецепте рассматриваются основы применения шейдеров тесселяции на примере рисования **кубической кривой Безье**. Кривая Безье – это параметрическая кривая, задаваемая четырьмя опорными точками, определяющими общую форму кривой. Первая и последняя точки соответствуют началу и концу кривой, а средние точки задают ее форму, при этом они не обязательно должны находиться на самой кривой. Кривая определяется путем интерполяции по четырем опорным точкам с использованием множества **функций сопряжения (blending functions)**. Функции сопряжения определяют вклад каждой опорной точки в формирование кривой в данную точку на кривой. Часто функции сопряжения называют **полиномами Бернштейна**:

$$B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i.$$

Первый член этого полинома – биномиальный коэффициент (ниже показано, как он определяется), n – степень полинома, i – номер полинома, t – параметрический параметр:

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}.$$

В общем параметрическом виде кривая Безье определяется как сумма произведений полиномов Бернштейна на опорные точки (P_i):

---

[1] Похожую статью на русском языке можно найти по адресу: **http://habrahabr.ru/post/228753/**. – *Прим. перев.*

$$P(t) = \sum_{i=0}^{n} B_i^n(t) P_i.$$

В данном примере реализовано рисование кубической кривой Безье по четырем опорным точкам ($n = 3$):

$$P(t) = B_0^3(t)P_0 + B_1^3(t)P_1 + B_2^3(t)P_2 + B_3^3(t)P_3,$$

где кубические полиномы Бернштейна определяются как:

$$B_0^3(t) = (1-t)^3;$$

$$B_1^3(t) = 3(1-t)^2 t;$$

$$B_2^3(t) = 3(1-t)t^2;$$

$$B_3^3(t) = t^3.$$

Как отмечалось во введении к этой главе, тесселяция в OpenGL выполняется в два этапа, двумя разными шейдерами: шейдером управления тесселяцией (Tessellation Control Shader, TCS) и шейдером выполнения тесселяции (Tessellation Evaluation Shader, TES). В этом примере шейдер управления будет получать число отрезков для кривой Безье и определять уровень тесселяции, а шейдер выполнения будет вычислять координаты точки на кривой Безье для каждой полученной вершины. На рис. 6.13 показан результат работы данного примера с тремя разными уровнями тесселяции. Изображение слева было создано с уровнем тесселяции 3 (три отрезка), в середине – с уровнем тесселяции 5, справа – с уровнем тесселяции 30. Маленькими квадратиками изображены опорные точки.

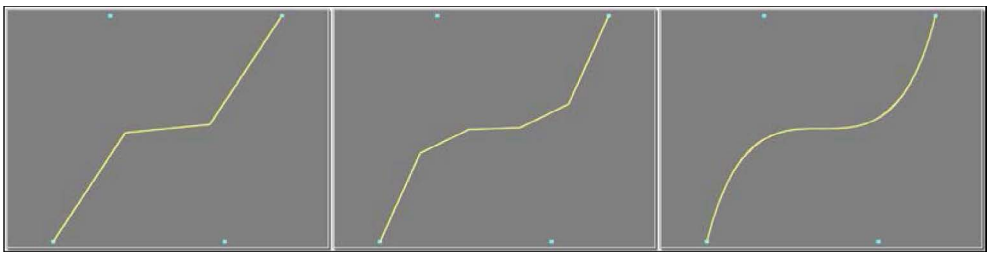

Рис. 6.13 ❖ Кривая Безье, полученная с тремя разными уровнями тесселяции

Опорные точки кривой Безье передаются вниз по конвейеру в виде примитива «заплаты», содержащего четыре вершины. Примитив «заплаты» – это тип примитива, определяемый программистом. В общем случае это группа вершин, которые программист может использовать по своему усмотрению. Шейдер управления TCS вызывается один раз для каждой вершины в «заплате», а шейдер выполнения TES – переменное число раз, в зависимости от количества вершин, произведенных генератором примитивов TPG. Результатом стадии тесселяции является множество примитивов. В данном случае таким множеством будет ломаная линия.

Одной из задач шейдера управления TCS является определение уровня тесселяции. В грубом приближении уровень тесселяции связан с числом вершин, которые должны быть сгенерированы. В данном случае шейдер управления TCS будет генерировать ломаную линию, соответственно, уровень тесселяции будет определяться числом отрезков в ней. Каждая вершина этой ломаной будет связана с координатой тесселяции, изменяющейся в диапазоне от нуля до единицы. Она соответствует параметру t в уравнении кривой Безье выше, и мы будем называть ее «координатой u».

Описание выше не отражает полную картину происходящего. В действительности шейдер управления TCS инициирует создание множества ломаных линий, называемых изолиниями. Каждая вершина в этом множестве имеет координаты `u` и `v`. Координата `u` изменяется в диапазоне от нуля до единицы вдоль заданной изолинии, а координата `v` является константой для каждой отдельной изолинии. Число уникальных значений `u` и `v` определяется двумя разными уровнями тесселяции, которые иногда называют «внешними» уровнями. Однако в данном примере генерируется только одна ломаная, поэтому второй уровень тесселяции (для `v`) всегда равен единице.

Основной задачей шейдера выполнения TES является определение координат вершины. Шейдер имеет доступ к координатам `u` и `v` вершины, а также (только для чтения) ко всем вершинам в «заплате». Определить координаты вершины можно с помощью параметрического уравнения, описанного выше, где роль параметра t будет играть координата `u`.

Подготовка

Ниже перечислены наиболее важные uniform-переменные в данном примере:

- `NumSegments`: число отрезков ломаной линии, которое нужно найти;
- `NumStrips`: число генерируемых изолиний. В данном примере этой переменной должно быть присвоено число `1`;
- `LineColor`: цвет ломаной линии.

Присвойте исходные значения uniform-переменным в основном приложении. Всего потребуется скомпилировать и скомпоновать четыре шейдера: вершинный, фрагментный, управления тесселяцией и выполнения тесселяции.

Как это делается...

Чтобы создать шейдерную программу, генерирующую кривую Безье на основе «заплаты» с четырьмя опорными точками, нужно выполнить следующие шаги:

1. Добавить простой вершинный шейдер:

```
layout (location = 0 ) in vec2 VertexPosition;
void main()
{
  gl_Position = vec4(VertexPosition, 0.0, 1.0);
}
```

2. Добавить шейдер управления тесселяцией:

```
layout( vertices=4 ) out;

uniform int NumSegments;
```

```
uniform int NumStrips;

void main()
{
  // Передать координаты вершины без изменений
  gl_out[gl_InvocationID].gl_Position =
           gl_in[gl_InvocationID].gl_Position;

  // Определить уровни тесселяции
  gl_TessLevelOuter[0] = float(NumStrips);
  gl_TessLevelOuter[1] = float(NumSegments);
}
```

3. Добавить шейдер выполнения тесселяции:

```
layout( isolines ) in;
uniform mat4 MVP; // матрица проекции вида модели

void main()
{
  // Координата u тесселяции
  float u = gl_TessCoord.x;

  // Вершины из "заплаты" (опорные точки)
  vec3 p0 = gl_in[0].gl_Position.xyz;
  vec3 p1 = gl_in[1].gl_Position.xyz;
  vec3 p2 = gl_in[2].gl_Position.xyz;
  vec3 p3 = gl_in[3].gl_Position.xyz;

  float u1 = (1.0 - u);
  float u2 = u * u;

  // Вычисление полинома Бернштейна для u
  float b3 = u2 * u;
  float b2 = 3.0 * u2 * u1;
  float b1 = 3.0 * u * u1 * u1;
  float b0 = u1 * u1 * u1;

  // Интерполяция кубической кривой Безье
  vec3 p = p0 * b0 + p1 * b1 + p2 * b2 + p3 * b3;

  gl_Position = MVP * vec4(p, 1.0);
}
```

4. Добавить фрагментный шейдер:

```
uniform vec4 LineColor;

layout ( location = 0 ) out vec4 FragColor;

void main()
{
  FragColor = LineColor;
}
```

5. Определить число вершин в «заплате» в основном приложении. Сделать это можно с помощью функции `glPatchParameter`:

```
glPatchParameteri( GL_PATCH_VERTICES, 4);
```

6. Отобразить четыре опорные точки в основном приложении:

```
glDrawArrays(GL_PATCHES, 0, 4);
```

Как это работает...

Вершинный шейдер играет роль «простого передаточного звена». Он передает вершину на следующую стадию без каких-либо изменений.

Шейдер управления тесселяцией начинается с определения числа вершин в «заплате»:

```
layout (vertices = 4) out;
```

Обратите внимание, что это число никак не связано с числом вершин, которые будут сгенерированы в процессе тесселяции. В данном случае «заплата» хранит четыре опорные точки, поэтому указано значение `4`.

Функция `main` шейдера управления переписывает координаты исходной вершины (из «заплаты») без изменений. Массивы `gl_out` и `gl_in` содержат входную и выходную информацию для каждой вершины в «заплате». Обратите внимание, что операции чтения/записи выполняются с элементами массивов, имеющими индекс `gl_InvocationID`. Переменная `gl_InvocationID` определяет номер вершины в выходной «заплате», для обработки которой был вызван шейдер. Шейдер управления может обращаться к любым элементам массива `gl_in`, но сохранять результаты должен только в элементе `gl_InvocationID` массива `gl_out`.

Далее шейдер управления устанавливает значения уровней тесселяции, присваивая значения элементам массива `gl_TessLevelOuter`. Отметьте, что элементами этого массива являются вещественные значения, а не целые. Они автоматически будут округляться до ближайшего целого самой системой OpenGL.

Первый элемент массива определяет число изолиний. Каждая изолиния получает свое постоянное значение координаты `v`. В данном примере элементу `gl_TessLevelOuter[0]` всегда присваивается единица. Второй определяет число отрезков, составляющих ломаную линию. Каждая вершина ломаной получит свое значение параметрической координаты `u` в диапазоне от нуля до единицы.

Шейдер выполнения тесселяции начинается с определения типа входящего примитива:

```
layout (isolines) in;
```

Здесь указывается тип примитива, который должен быть создан генератором примитивов тесселяции. Таких типов всего три, двумя другими являются `quads` и `triangles`.

В функции `main` шейдера выполнения тесселяции переменная `gl_TessCoord` хранит координаты `u` и `v` для данного вызова. Для тесселяции в одном измерении (как в данном случае) нужна только координата `u`, соответствующая координате `x` в `gl_TessCoord`.

Далее в локальные переменные переписываются координаты четырех опорных точек из входящего примитива «заплаты» – массива `gl_in`.

Затем для координаты u вычисляются значения кубических полиномов Бернштейна, и результаты сохраняются в b0, b1, b2 и b3. После этого на основе уравнения кривой Безье (описанного выше) определяются координаты вершины, преобразуются в усеченную систему координат и присваиваются выходной переменной gl_Position.

Фрагментный шейдер просто применяет цвет LineColor к фрагменту.

И еще...

Шейдеры тесселяции способны решать гораздо более сложные задачи, но этот пример задумывался как простое введение, поэтому оставим обсуждение дополнительных возможностей для других рецептов. Далее мы рассмотрим пример тесселяции поверхностей в двух измерениях.

Тесселяция двухмерного прямоугольника

Один из лучших способов разобраться в аппаратной тесселяции – попробовать выполнить тесселяцию двухмерного прямоугольника. При использовании линейной интерполяции получающиеся треугольники непосредственно связаны с координатами тесселяции (u,v), которые вычисляются генератором примитивов тесселяции. Чрезвычайно полезной будет попытка нарисовать несколько прямоугольников с разными уровнями тесселяции и попутно исследовать получающиеся треугольники. Именно этим мы и займемся в данном рецепте.

Когда осуществляется тесселяция прямоугольника, генератор примитивов делит пространство параметров (u,v) на множество фрагментов, исходя из шести параметров. К этим параметрам относятся уровни внутренней тесселяции для u и v (внутренние уровни 0 и 1) и уровни внешней тесселяции для u и v и обоих краев (внешние уровни 0, 1, 2 и 3). Данные параметры определяют число делений вдоль краев и внутри. Рассмотрим их по отдельности:

- **внешний уровень 0 (OL0)**: число делений вдоль v для u = 0;
- **внешний уровень 1 (OL1)**: число делений вдоль u для v = 0;
- **внешний уровень 2 (OL2)**: число делений вдоль v для u = 1;
- **внешний уровень 3 (OL3)**: число делений вдоль u для v = 1;
- **внутренний уровень 0 (IL0)**: число делений вдоль u для всех внутренних значений v;
- **внутренний уровень 1 (IL1)**: число делений вдоль v для всех внутренних значений u.

На рис. 6.14 показаны отношения между уровнями тесселяции и области пространства параметров, где они действуют. Внешние уровни определяют число делений вдоль краев, а внутренние уровни – число делений внутри.

Шесть уровней тесселяции, описанных выше, настраиваются через массивы gl_TessLevelOuter и gl_TessLevelInner. Например, элемент gl_TessLevelInner[0] соответствует уровню **IL0**, gl_TessLevelOuter[2] – уровню OL2 и т. д.

Если нарисовать примитив «заплаты», состоящий из единственного прямоугольника (четыре вершины), используя линейную интерполяцию, получившие-

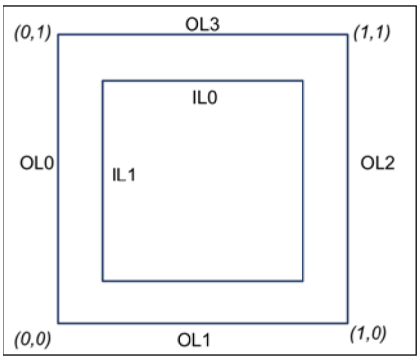

Рис. 6.14 ❖ Уровни тесселяции и области пространства параметров, где они действуют

ся треугольники помогут понять, как OpenGL осуществляет тесселяцию прямоугольника. На рис. 6.15 показаны результаты отображения примитива «заплаты» с разными уровнями тесселяции.

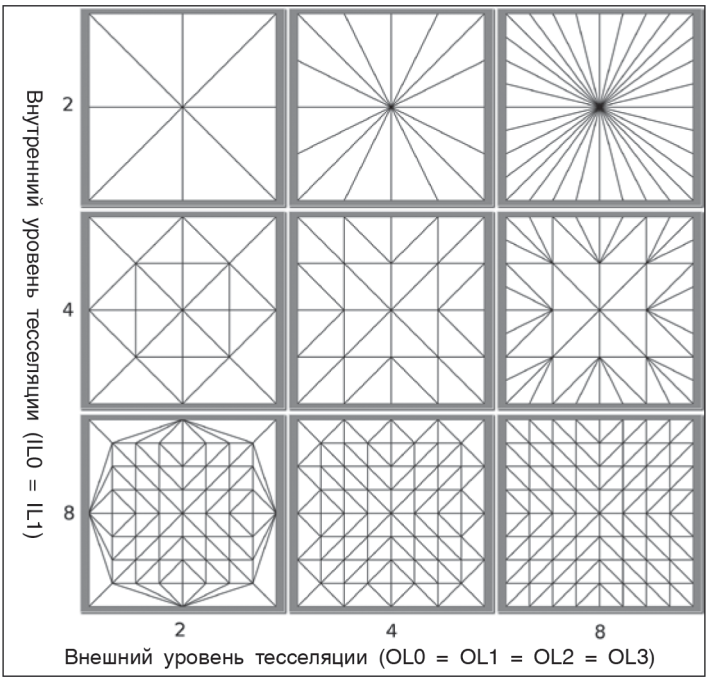

Рис. 6.15 ❖ Результаты отображения примитива «заплаты» с разными уровнями тесселяции

При использовании линейной интерполяции получающиеся треугольники являются визуальным отображением пространства параметров (u, v). Ось X на рис. 6.15 соответствует координате u, а ось Y – координате v. Вершины треугольников имеют координаты (u,v), вычисленные генератором примитивов тесселяции. Число делений ясно видно в получившейся сетке треугольников. Например, если всем внешним уровням присвоить значение 2 и всем внутренним – значение 8, внешние края будут поделены пополам, но внутри прямоугольника координаты u и v будут поделены на 8 интервалов.

Прежде чем перейти к программному коду, обсудим еще линейную интерполяцию. Если предположить, что имеется прямоугольник с четырьмя вершинами, как показано на рис. 6.16, тогда координаты любой точки внутри прямоугольника можно определить путем линейной интерполяции по четырем вершинам, с учетом параметров u и v.

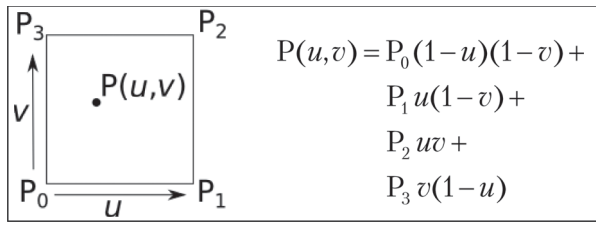

Рис. 6.16 ❖ Координаты любой точки внутри прямоугольника можно определить путем линейной интерполяции по четырем вершинам

Если позволить генератору примитивов создать множество вершин с соответствующими параметрическими координатами, мы легко сможем определить их местоположения, интерполируя по четырем углам прямоугольника в соответствии с формулой на рис. 6.16.

Подготовка

Значения внешних и внутренних уровней тесселяции будут определяться через uniform-переменные Inner и Outer. Для отображения треугольников нам потребуется геометрический шейдер, о котором рассказывалось в первой части главы.

Подготовьте приложение OpenGL для отображения примитива «заплаты», состоящего из четырех вершин, как показано на рис. 6.16.

Как это делается...

Чтобы создать шейдерную программу, которая сгенерирует множество треугольников с использованием механизма тесселяции на основе «заплаты» с четырьмя вершинами, нужно выполнить следующие шаги:

1. Добавить простой вершинный шейдер:

```
layout (location = 0 ) in vec2 VertexPosition;

void main()
{
  gl_Position = vec4(VertexPosition, 0.0, 1.0);
}
```

2. Добавить шейдер управления тесселяцией:

```
layout( vertices=4 ) out;

uniform int Outer;
uniform int Inner;

void main()
{
  // Передать координаты вершины без изменений
  gl_out[gl_InvocationID].gl_Position =
            gl_in[gl_InvocationID].gl_Position;

  gl_TessLevelOuter[0] = float(Outer);
  gl_TessLevelOuter[1] = float(Outer);
  gl_TessLevelOuter[2] = float(Outer);
  gl_TessLevelOuter[3] = float(Outer);

  gl_TessLevelInner[0] = float(Inner);
  gl_TessLevelInner[1] = float(Inner);
}
```

3. Добавить шейдер выполнения тесселяции:

```
layout( quads, equal_spacing, ccw ) in;

uniform mat4 MVP;

void main()
{
  float u = gl_TessCoord.x;
  float v = gl_TessCoord.y;

  vec4 p0 = gl_in[0].gl_Position;
  vec4 p1 = gl_in[1].gl_Position;
  vec4 p2 = gl_in[2].gl_Position;
  vec4 p3 = gl_in[3].gl_Position;

  // Линейная интерполяция
  gl_Position =
      p0 * (1-u) * (1-v) +
      p1 * u * (1-v) +
      p3 * v * (1-u) +
      p2 * u * v;

  // Преобразовать в систему усеченных координат
  gl_Position = MVP * gl_Position;
}
```

4. Добавить геометрический шейдер из рецепта «Наложение каркаса на освещенную поверхность».
5. Добавить фрагментный шейдер:

```
uniform float LineWidth;
uniform vec4 LineColor;
uniform vec4 QuadColor;

noperspective in vec3 EdgeDistance; // Из геом. шейдера

layout ( location = 0 ) out vec4 FragColor;

float edgeMix()
{
  // ** вставить сюда код, определяющий пропорцию смешивания
  //    с цветом линий (см. рецепт "Наложение каркаса на
  //    освещенную поверхность" **
}

void main()
{
  float mixVal = edgeMix();

  FragColor = mix( QuadColor, LineColor, mixVal );
}
```

6. В функции отображения, в основной программе, определить число вершин в «заплате»:

```
glPatchParameteri(GL_PATCH_VERTICES, 4);
```

7. Отобразить «заплату» как четыре двухмерные вершины в порядке следования по часовой стрелке.

Как это работает...

Вершинный шейдер передает координаты шейдеру управления тесселяцией без каких-либо изменений.

Шейдер управления определяет число вершин в «заплате» с помощью квалификатора `layout`:

```
layout (vertices=4) out;
```

В функции `main` он передает вершину дальше, без изменений, и устанавливает внешние и внутренние уровни тесселяции. Все четыре внешних уровня тесселяции устанавливаются в значение `Outer`, а два внутренних уровня – в значение `Inner`.

В шейдере выполнения тесселяции, с помощью квалификатора `layout`, задаются режим и другие параметры тесселяции:

```
layout ( quads, equal_spacing, ccw ) in;
```

Параметр `quads` указывает, что генератор примитивов тесселяции должен делить пространство параметров с использованием прямоугольной тесселяции, как опи-

сывалось выше. Параметр `equal_spacing` сообщает, что тесселяция должна выполняться так, чтобы деление выполнялось на равные отрезки. Последний параметр `ccw` требует, чтобы примитивы генерировались в направлении по часовой стрелке.

Функция `main` в шейдере выполнения тесселяции начинается с извлечения параметрических координат вершины, обращением к переменной `gl_TessCoord`. Затем во временные переменные копируются координаты четырех вершин из массива `gl_in` `array` «заплаты», которые потом используются для интерполяции.

После этого результат интерполяции присваивается встроенной выходной переменной `gl_Position`. В заключение полученные координаты преобразуются в систему усеченных координат путем умножения на матрицу проекции вида модели.

Внутри фрагментного шейдера всем фрагментам присваивается цвет, который может смешиваться с цветом линий.

См. также

- Рецепт «Наложение каркаса на освещенную поверхность».

Тесселяция трехмерной поверхности

Чтобы продемонстрировать пример тесселяции трехмерной поверхности, мы попробуем отобразить многогранник в форме чайника. Как оказывается, в действительности чайник определяется набором «заплат» 4×4 с опорными точками для кубической интерполяции Безье. То есть отображение чайника фактически сводится к рисованию набора кубических поверхностей Безье.

Похоже, что эта задача как раз для шейдеров тесселяции! Мы будем отображать каждую «заплату» с 16 вершинами как примитив, используя прямоугольную тесселяцию для деления пространства параметров, и реализуем интерполяцию Безье внутри шейдера выполнения тесселяции.

На рис. 6.17 представлены желаемые результаты. Слева изображен чайник, полученный с уровнями внешней и внутренней тесселяции, равными `2`, в середине – с уровнем `4`, справа – с уровнем `16`. Шейдер выполнения тесселяции использует интерполяцию Безье.

Рис. 6.17 ❖ Изображение чайника с разными уровнями тесселяции

Для начала посмотрим, как выполняется кубическая интерполяция Безье. Если предположить, что поверхность определена 16 опорными точками (в виде матрицы 4×4) P_{ij}, где индексы i и j изменяются в диапазоне от 0 до 3, тогда параметрическая поверхность Безье задается следующим уравнением:

$$P(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_i^3(u)B_j^3(v)P_{ij}.$$

Члены B в предыдущем уравнении – это кубические полиномы Бернштейна (см. предыдущий рецепт «Тесселяция двухмерного прямоугольника»).

Для каждой точки, полученной путем интерполяции, также необходимо вычислить вектор нормали как векторное произведение частных производных предыдущего уравнения:

$$n(u,v) = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v}.$$

Вычисление частных производных поверхности Безье сводится к вычислению частных производных полиномов Бернштейна:

$$\frac{\partial P}{\partial u} = \sum_{i=0}^{3}\sum_{j=0}^{3} \frac{\partial B_i^3(u)}{\partial u} B_j^3(v)P_{ij};$$

$$\frac{\partial P}{\partial v} = \sum_{i=0}^{3}\sum_{j=0}^{3} B_i^3(u)\frac{\partial B_j^3(v)}{\partial v} P_{ij}.$$

Поиск частных производных и вычисление векторного произведения будут производиться в шейдере выполнения тесселяции для каждой генерируемой вершины.

Подготовка

Подготовьте вершинный шейдер, который будет просто передавать координаты, не изменяя их (можно воспользоваться вершинным шейдером из рецепта «Тесселяция двухмерного прямоугольника»). Добавьте фрагментный шейдер, реализующий модель освещенности/затенения по своему выбору. Он должен принимать входные переменные `TENormal` и `TEPosition` с вектором нормали и вершиной в системе видимых координат.

Переменной `TessLevel` должно быть присвоено значение желаемого уровня тесселяции. Это значение будет присваиваться всем внутренним и внешним уровням тесселяции.

Как это делается...

Чтобы создать шейдерную программу, которая сгенерирует «заплаты» Безье на основе входящих «заплат» с 16 опорными точками, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер из рецепта «Тесселяция двухмерного прямоугольника».

2. Добавить шейдер управления тесселяцией:

```
layout( vertices=16 ) out;

uniform int TessLevel;

void main()
{
  // Передать координаты вершины без изменений
  gl_out[gl_InvocationID].gl_Position =
              gl_in[gl_InvocationID].gl_Position;

  gl_TessLevelOuter[0] = float(TessLevel);
  gl_TessLevelOuter[1] = float(TessLevel);
  gl_TessLevelOuter[2] = float(TessLevel);
  gl_TessLevelOuter[3] = float(TessLevel);

  gl_TessLevelInner[0] = float(TessLevel);
  gl_TessLevelInner[1] = float(TessLevel);
}
```

3. Добавить шейдер выполнения тесселяции:

```
layout( quads ) in;
out vec3 TENormal;   // Нормаль в видимых координатах
out vec4 TEPosition; // Позиция вершины в видимых координатах

uniform mat4 MVP;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;

void basisFunctions(out float[4] b, out float[4] db, float t)
{
  float t1 = (1.0 - t);
  float t12 = t1 * t1;

  // Полиномы Бернштейна
  b[0] = t12 * t1;
  b[1] = 3.0 * t12 * t;
  b[2] = 3.0 * t1 * t * t;
  b[3] = t * t * t;

  // Производные
  db[0] = -3.0 * t1 * t1;
  db[1] = -6.0 * t * t1 + 3.0 * t12;
  db[2] = -3.0 * t * t + 6.0 * t * t1;
  db[3] = 3.0 * t * t;
}

void main()
{
  float u = gl_TessCoord.x;
  float v = gl_TessCoord.y;

  // Шестнадцать опорных точек
```

```
vec4 p00 = gl_in[0].gl_Position;
vec4 p01 = gl_in[1].gl_Position;
vec4 p02 = gl_in[2].gl_Position;
vec4 p03 = gl_in[3].gl_Position;
vec4 p10 = gl_in[4].gl_Position;
vec4 p11 = gl_in[5].gl_Position;
vec4 p12 = gl_in[6].gl_Position;
vec4 p13 = gl_in[7].gl_Position;
vec4 p20 = gl_in[8].gl_Position;
vec4 p21 = gl_in[9].gl_Position;
vec4 p22 = gl_in[10].gl_Position;
vec4 p23 = gl_in[11].gl_Position;
vec4 p30 = gl_in[12].gl_Position;
vec4 p31 = gl_in[13].gl_Position;
vec4 p32 = gl_in[14].gl_Position;
vec4 p33 = gl_in[15].gl_Position;

// Вычислить базисные функции
float bu[4], bv[4]; // Базисные функции для u и v
float dbu[4], dbv[4]; // Производные для u и v

basisFunctions(bu, dbu, u);
basisFunctions(bv, dbv, v);

// Интерполяция Безье
TEPosition =
  p00*bu[0]*bv[0] + p01*bu[0]*bv[1] + p02*bu[0]*bv[2] +
  p03*bu[0]*bv[3] +
  p10*bu[1]*bv[0] + p11*bu[1]*bv[1] + p12*bu[1]*bv[2] +
  p13*bu[1]*bv[3] +
  p20*bu[2]*bv[0] + p21*bu[2]*bv[1] + p22*bu[2]*bv[2] +
  p23*bu[2]*bv[3] +
  p30*bu[3]*bv[0] + p31*bu[3]*bv[1] + p32*bu[3]*bv[2] +
  p33*bu[3]*bv[3];

// Частные производные
vec4 du =
  p00*dbu[0]*bv[0]+p01*dbu[0]*bv[1]+p02*dbu[0]*bv[2]+
  p03*dbu[0]*bv[3]+
  p10*dbu[1]*bv[0]+p11*dbu[1]*bv[1]+p12*dbu[1]*bv[2]+
  p13*dbu[1]*bv[3]+
  p20*dbu[2]*bv[0]+p21*dbu[2]*bv[1]+p22*dbu[2]*bv[2]+
  p23*dbu[2]*bv[3]+
  p30*dbu[3]*bv[0]+p31*dbu[3]*bv[1]+p32*dbu[3]*bv[2]+
  p33*dbu[3]*bv[3];

vec4 dv =
  p00*bu[0]*dbv[0]+p01*bu[0]*dbv[1]+p02*bu[0]*dbv[2]+
  p03*bu[0]*dbv[3]+
  p10*bu[1]*dbv[0]+p11*bu[1]*dbv[1]+p12*bu[1]*dbv[2]+
  p13*bu[1]*dbv[3]+
  p20*bu[2]*dbv[0]+p21*bu[2]*dbv[1]+p22*bu[2]*dbv[2]+
  p23*bu[2]*dbv[3]+
  p30*bu[3]*dbv[0]+p31*bu[3]*dbv[1]+p32*bu[3]*dbv[2]+
```

```
    p33*bu[3]*dbv[3];

  // Нормаль как векторное произведение частных производных
  vec3 n = normalize( cross(du.xyz, dv.xyz) );

  // Преобразовать в усеченные координаты
  gl_Position = MVP * TEPosition;

  // Преобразовать в видимые координаты
  TEPosition = ModelViewMatrix * TEPosition;
  TENormal = normalize(NormalMatrix * n);
}
```

4. Реализовать во фрагментном шейдере модель освещенности/затенения по своему выбору, используя выходные переменные из шейдера TES.
5. Отобразить опорные точки Безье как примитив «заплаты» с 16 вершинами. Не забудьте указать число вершин в «заплате» внутри основного приложения:

```
glPatchParameteri(GL_PATCH_VERTICES, 16);
```

Как это работает...

Шейдер управления тесселяцией начинается с квалификатора `layout`, определяющего число вершин в «заплате»:

```
layout( vertices=16 ) out;
```

Затем он просто устанавливает уровни тесселяции в значение `TessLevel` и отправляет координаты вершины дальше без каких-либо изменений.

Шейдер выполнения тесселяции начинается с квалификатора `layout`, определяющего тип тесселяции. Так как тесселяции подвергается поверхность Безье размером 4×4, наиболее уместным выглядит тип `quads`.

Функция `basisFunctions` вычисляет полиномы Бернштейна и их производные для заданного значения параметра `t`. Результаты возвращаются в выходных параметрах `b` и `db`.

Функция `main` сначала присваивает координаты тесселяции переменным `u` и `v`, а затем переписывает все 16 вершин из «заплаты» во временные переменные с более короткими именами (чтобы сократить код, который следует ниже).

Затем, чтобы вычислить полиномы Бернштейна и их производные `u` и `v`, вызывается функция `basisFunctions`. Результаты сохраняются в переменных `bu`, `dbu`, `bv` и `dbv`.

Следующий шаг – вычисление сумм на основе формул, описанных выше, для позиции `TEPosition`, частной производной для `u` (`du`) и частной производной для `v` (`dv`).

Вектор нормали вычисляется как векторное произведение `du` и `dv`.

В заключение позиция `TEPosition` преобразуется в систему усеченных координат, и результат присваивается переменной `gl_Position`. Эта позиция преобразуется также в видимые координаты перед передачей фрагментному шейдеру.

Преобразование вектора нормали в видимые координаты выполняется умножением на `NormalMatrix` с последующей нормализацией. Результат передается фрагментному шейдеру через переменную `TENormal`.

См. также

- Рецепт «Тесселяция двухмерного прямоугольника».

Тесселяция с учетом глубины

Одной из замечательных особенностей шейдеров тесселяции является простота реализации с их помощью алгоритмов изменения **уровня детализации (Level-Of-Detail, LOD)**. Термин «LOD» часто используется в компьютерной графике для обозначения процесса увеличения или уменьшения сложности геометрии объекта в зависимости от расстояния до камеры (или других факторов). Чем дальше от камеры находится объект, тем ниже детальность его отображения, потому что сам объект выглядит меньше. Однако по мере приближения объекта к камере он заполняет все большую и большую часть экрана, и тем больше деталей нужно отобразить (сделать поверхность более гладкой или убрать какие-либо геометрические артефакты).

На рис. 6.18 показаны изображения нескольких чайников, полученных с разными уровнями тесселяции в зависимости от расстояния до камеры. Изображения всех чайников получены с помощью одного и того же кода в основной программе. А отличия обусловлены автоматическим выбором уровня тесселяции в зависимости от глубины в шейдере управления тесселяцией.

При использовании шейдеров тесселяции степень сложности геометрической формы объекта определяется уровнем тесселяции. А так как уровни тесселяции

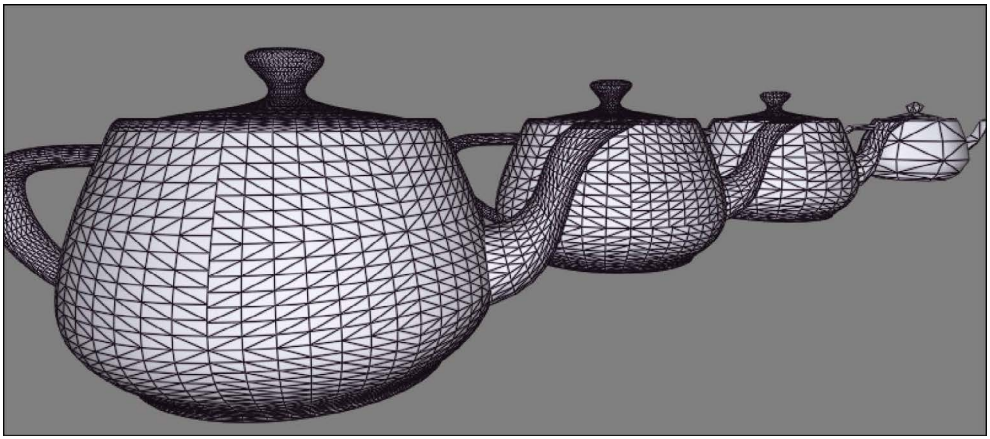

Рис. 6.18 ❖ Изображения нескольких чайников, полученных с разными уровнями тесселяции

можно задавать внутри шейдера управления тесселяцией, не составляет никакого труда менять их в зависимости от расстояния до камеры.

Данный пример реализует линейное изменение уровней тесселяции от минимума до максимума в зависимости от расстояния до камеры. «Расстояние до камеры» вычисляется как абсолютное значение координаты Z в системе видимых координат (разумеется, это не истинное расстояние, но для целей данного примера это вполне приемлемое допущение). Уровни тесселяции определяются, исходя из этого значения. В примере также определяются два дополнительных значения (как uniform-переменные) – `MinDepth` и `MaxDepth`. Для отображения объектов, находящихся относительно камеры ближе, чем `MinDepth`, используется максимальный уровень тесселяции, а для отображения объектов, находящихся дальше, чем `MaxDepth`, используется минимальный уровень тесселяции. Для объектов, находящихся между этими крайними точками, уровень тесселяции вычисляется по формуле линейной интерполяции.

Подготовка

Программа для данного рецепта почти идентична программе из рецепта «Тесселяция трехмерной поверхности». Единственное отличие заключено в шейдере управления тесселяцией. Из программы нужно удалить uniform-переменную `TessLevel` и добавить несколько новых:

- `MinTessLevel`: минимальный уровень тесселяции;
- `MaxTessLevel`: максимальный уровень тесселяции;
- `MinDepth`: минимальное «расстояние» до камеры, до достижения которого используется максимальный уровень тесселяции;
- `MaxDepth`: максимальное «расстояние» до камеры, за которым используется минимальный уровень тесселяции.

Объекты отображаются в виде примитивов «заплат» с 16 вершинами, как было показано в рецепте «Тесселяция трехмерной поверхности».

Как это делается...

Чтобы создать шейдерную программу, изменяющую уровень тесселяции в зависимости от глубины, нужно сделать следующие шаги:

1. Добавить вершинный шейдер и шейдер выполнения тесселяции из рецепта «Тесселяция трехмерной поверхности».
2. Добавить шейдер управления тесселяцией:

```
layout( vertices=16 ) out;

uniform int MinTessLevel;
uniform int MaxTessLevel;
uniform float MaxDepth;
uniform float MinDepth;

uniform mat4 ModelViewMatrix;

void main()
```

```
{
  // Позиция в видимых координатах
  vec4 p = ModelViewMatrix *
                  gl_in[gl_InvocationID].gl_Position;

  // "Расстояние" до камеры преобразуется в диапазон от 0 до 1
  float depth = clamp( (abs(p.z) - MinDepth) /
                       (MaxDepth - MinDepth), 0.0, 1.0 );

  // Интерполировать между минимальным и
  // максимальным уровнями тесселяции
  float tessLevel =
          mix(MaxTessLevel, MinTessLevel, depth);

  gl_TessLevelOuter[0] = float(tessLevel);
  gl_TessLevelOuter[1] = float(tessLevel);
  gl_TessLevelOuter[2] = float(tessLevel);
  gl_TessLevelOuter[3] = float(tessLevel);

  gl_TessLevelInner[0] = float(tessLevel);
  gl_TessLevelInner[1] = float(tessLevel);

  gl_out[gl_InvocationID].gl_Position =
                  gl_in[gl_InvocationID].gl_Position;
}
```

3. Реализовать внутри фрагментного шейдера модель освещенности/затенения по своему выбору, как в предыдущем рецепте.

Как это работает...

Шейдер управления принимает координаты вершины, преобразует их в систему видимых координат и сохраняет результат в переменной `p`. Затем абсолютное значение координаты Z масштабируется и усекается так, чтобы в результате получилось значение в диапазоне от нуля до единицы. Если координата Z равна `MaxDepth`, переменная `depth` получит значение `1.0`; если она равна `MinDepth`, получит значение `0.0`. Если координата Z имеет значение между `MinDepth` и `MaxDepth`, переменная `depth` получит значение от нуля до единицы. Если значение координаты Z выходит за этот диапазон, значение результата будет усечено функцией `clamp` до `0.0` или `1.0`.

Далее значение `depth` используется для линейной интерполяции между `MaxTessLevel` и `MinTessLevel` с помощью функции `mix`. Результат `tessLevel` используется для установки внешних и внутренних уровней тесселяции.

И еще...

В этом примере есть один тонкий момент. Вспомните, что шейдер управления тесселяцией вызывается один раз для каждой вершины в выходящем примитиве «заплаты». То есть при отображении кубических поверхностей Безье этот шейдер будет выполнен 16 раз для каждой «заплаты». В каждом вызове значение `depth` будет получаться немного другим, из-за того что вычисления выполняются на основе координаты Z вершины. В этот момент у многих наверняка возник вопрос:

какой из 16 возможных уровней тесселяции будет использоваться? Уровни тесселяции не интерполируются по пространству параметров. Так какое же значение будет выбрано?

Выходные массивы `gl_TessLevelInner` и `gl_TessLevelOuter` являются входными переменными для всей «заплаты». То есть для «заплаты» будет использоваться только одно значение, подобно тому как действует квалификатор `flat`. Спецификация OpenGL, похоже, указывает, что в конечном итоге может использоваться любое из значений, полученных в вызовах шейдера управления тесселяцией.

См. также

- Статью «DirectX 11 Terrain Tessellation» по адресу: **http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/TerrainTessellation_WhitePaper.pdf**.
- Рецепт «Тесселяция трехмерной поверхности».

Глава 7

Тени

В этой главе описываются следующие рецепты:

- отображение теней с помощью карты теней;
- сглаживание границ теней методом PCF;
- смягчение границ теней методом случайной выборки;
- создание теней с использованием приема теневых объемов и геометрического шейдера.

Введение

Тени играют важную роль в придании реалистичности. Без теней легко можно ошибиться в оценке взаимного расположения объектов, а освещение может выглядеть нереалистичным из-за того, что создается ощущение, будто лучи проходят сквозь объекты.

Тени – важный фактор, увеличивающий реалистичность сцен, но иногда бывает сложно выбрать наиболее эффективные алгоритмы, подходящие для интерактивных приложений. Для создания теней в графике реального времени часто используется алгоритм на основе **карты теней**. В этой главе мы рассмотрим несколько рецептов, основанных на алгоритме использования карты теней. В первом рецепте я познакомлю вас с основным алгоритмом и подробно расскажу о нем. Затем я представлю два приема, улучшающих внешний вид теней, создаваемых с помощью основного алгоритма.

В конце главы я расскажу об альтернативном приеме создания теней, который называют «теневыми объемами» (shadow volumes). Этот прием создает почти идеальные тени с резкими границами, но он плохо подходит для создания мягких, размытых теней.

Отображение теней с помощью карты теней

Один из наиболее часто используемых и популярных приемов отображения теней называется «карта теней». В его каноническом виде этот алгоритм выполняет обработку в два прохода. В первом проходе сцена отображается, как она видится с позиции источника света. Информация о глубинах, полученная в этом проходе, сохраняется в текстуре, которую называют картой теней. Эта карта помогает получить информацию о видимости объектов с позиции источника света. Иными словами, карта теней хранит расстояние (в действительности псевдоглубину) от источника света до всего, что он «видит». Все, что оказывается ближе к источнику

света, чем соответствующая глубина в карте, освещается; все остальное оказывается в тени.

Во втором проходе сцена отображается как обычно, но сначала глубина фрагмента (с позиции источника света) сравнивается со значением в карте теней, чтобы определить, находится ли фрагмент в тени. Затем, в зависимости от результата сравнения, цвет фрагмента вычисляется одним из двух способов. Если фрагмент находится в тени, при вычислении его цвета учитывается только фоновое (ambient) освещение; в противном случае цвет вычисляется как обычно.

На рис. 7.1 показан пример отображения теней с использованием простого приема на основе карт теней.

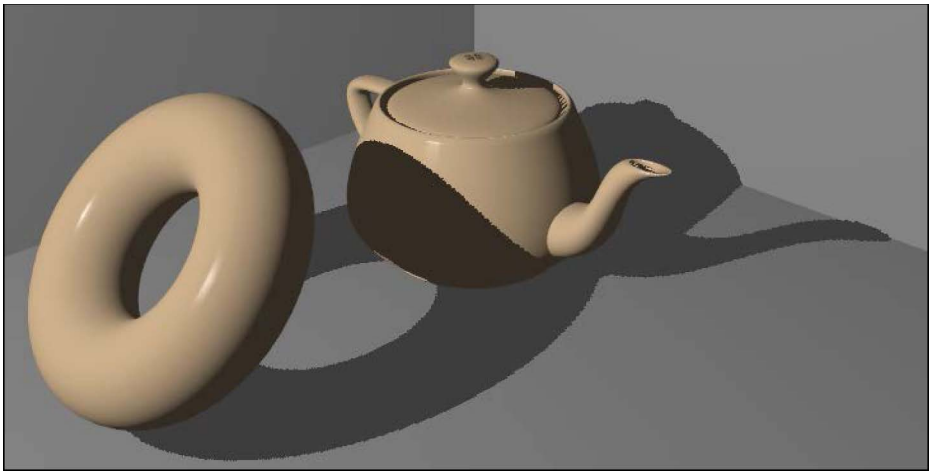

Рис. 7.1 ❖ Отображение теней с использованием карт теней

А теперь подробно рассмотрим работу алгоритма.

Первый шаг – создание карты теней. Предварительно нужно определить матрицу вида – такую, чтобы сцена отображалась, как если бы камера находилась в позиции источника света и смотрела в сторону объектов, отбрасывающих тень; матрицу проекции – такую, чтобы видимый объем включал все объекты, которые могут отбрасывать тени, а также область, куда падают эти тени. Затем следует отобразить сцену обычным способом и сохранить информацию из буфера глубины в текстуре. Эта текстура называется картой теней (или картой глубин). Ее можно считать множеством расстояний от источника света до различных поверхностей.

Технически значения глубины не являются расстояниями. Глубина – это не настоящее расстояние (от начала координат), но в грубом приближении ее можно считать таковой для целей данного примера.

На рис. 7.2 показано, как осуществляется составление карты теней. Слева показаны местоположение источника света и соответствующий ему видимый объ-

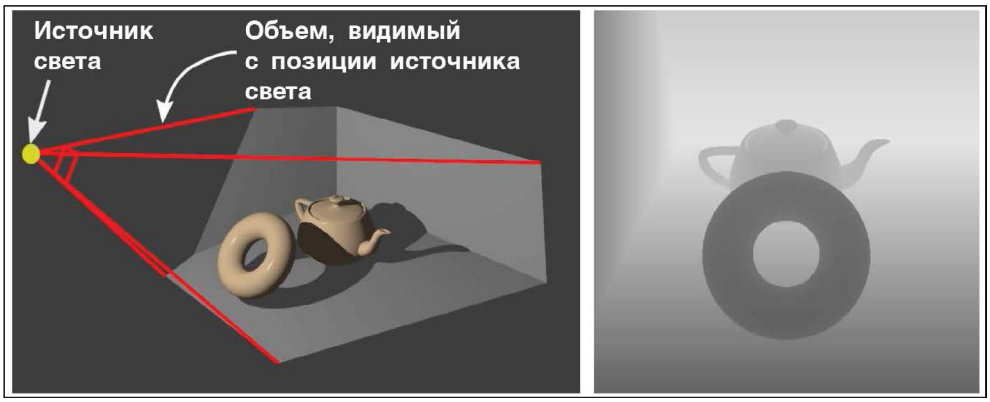

Рис. 7.2 ❖ Слева: источник света и его видимый объем; справа: соответствующая карта теней

ем. Справа показана карта теней для данной сцены. Интенсивность серого цвета в карте теней соответствует значениям глубины (чем темнее, тем ближе).

После создания карты теней и сохранения ее в текстуре выполняется повторное отображение сцены, но уже с позиции камеры. На этот раз используется фрагментный шейдер, определяющий затенение каждого фрагмента на основе результатов сравнения глубины со значением из карты теней. Сначала позиция фрагмента преобразуется в систему координат с началом в точке местоположения источника света и с применением матрицы проекции для источника света. Полученный результат используется для получения координат в текстуре и сравнения со значением в карте теней. Если значение глубины для фрагмента оказывается больше, чем соответствующее значение в карте теней, следовательно, между фрагментом и источником света находится какой-то объект, данный фрагмент находится в тени, и цвет его должен вычисляться только с учетом фонового освещения. В противном случае фрагмент освещается источником света, и его цвет должен вычисляться, как обычно.

Ключевым аспектом здесь является преобразование трехмерных координат фрагмента в систему координат, соответствующую карте теней. Так как карта теней является обычной двухмерной текстурой, нужно получить координаты со значениями от нуля до единицы для точки, находящейся внутри объема, видимого с позиции источника света. Преобразование позиции точки из мировых координат в видимые координаты, с началом в позиции источника света, выполняется с помощью матрицы вида источника света. А преобразование видимых координат в **однородные усеченные координаты (homogeneous clip coordinates)** выполняется с помощью матрицы проекции источника света.

Система координат называется усеченной, потому что при преобразовании в эту систему координат вызывается встроенный механизм усечения. Координаты точек внутри видимого объема (перспективного или ортографического) преобразуются матрицей проекции в (однородное, homogeneous) пространство в форме куба с центром в начале координат

и длиной стороны, равной двум. Это пространство называют **каноническим видимым объемом** (**canonical viewing volume**). Под термином «однородные» («homogeneous») подразумевается, что эти координаты могут считаться истинными декартовыми координатами только после деления компонентов `x`, `y` и `z` на четвертую координату. За более полным определением гомогенных координат обращайтесь к специализированным книгам по компьютерной графике.

Компоненты `x` и `y` позиции в усеченной системе координат – именно то, что нужно для доступа к карте теней. Компонент `z` содержит значение глубины, которое нужно сравнить со значением из карты теней. Однако прежде чем выполнить сравнение, нужно сделать две вещи. Во-первых, координаты следует привести к диапазону `[0..1]` (из диапазона `[-1..1]`) и, во-вторых, применить **перспективное деление (perspective division)**.

Чтобы привести усеченные координаты X, Y и Z точки (в видимом объеме источника света) к виду, пригодному для использования с картой теней, их необходимо преобразовать в диапазон от нуля до единицы. Значения, хранящиеся в буфере глубины OpenGL (а также в карте теней), – это простые вещественные значения в диапазоне от нуля до единицы (обычно). Нулевому значению соответствует точка на ближней плоскости отсечения в перспективном видимом объеме, а единичному значению соответствует точка на дальней плоскости отсечения. Поэтому, прежде чем сравнивать координату Z со значением из буфера глубины, ее необходимо масштабировать и сместить в соответствии с параметрами видимого объема.

Значение Z в усеченной системе координат (после перспективного деления) изменяется в диапазоне от `-1` до `1`. Описываемое преобразование видимого объема (viewport transformation), помимо всего прочего, приводит значение глубины к диапазону от нуля до единицы. Однако в случае необходимости вызовом функции `glDepthRange` преобразование видимого объема можно настроить для приведения значений глубины к другому диапазону (например, от `0` до `100`).

Конечно, компоненты `x` и `y` также нужно привести к диапазону от нуля до единицы, чтобы иметь возможность извлекать данные из текстуры.

Для преобразования усеченной системы координат можно использовать следующую матрицу «приведения»:

$$B = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Эта матрица предусматривает масштабирование и перенос координат X, Y и Z точек в объеме, видимом с позиции источника света, в диапазон от `0` до `1` (после перспективного деления). Теперь, объединив матрицу приведения с матрицами вида (V_l) и проекции (P_l) для источника света, можно получить уравнение преобразования мировых координат (W) в гомогенные координаты (Q), пригодные для доступа к карте теней:

$$Q = BP_lV_lW.$$

Наконец, перед использованием вектора координат Q его компоненты `x`, `y` и `z` необходимо разделить на четвертый компонент `w`. Этот шаг иногда называют «перспективным делением» («perspective division»). Эта операция преобразует однородные координаты в истинные декартовы координаты, и ее всегда необходимо выполнять при использовании матрицы перспективной проекции.

Следующее уравнение содержит матрицу тени (S), включающую также матрицу модели (M), благодаря чему с ее помощью можно выполнить непосредственное преобразование координат модели (C). (Имейте в виду, что $W = MC$, потому что матрица модели приводит координаты модели к мировым координатам.)

$Q = SC$.

Здесь S – матрица тени – произведение матрицы модели на все предшествующие матрицы.

$S = BP_lV_lM$.

Для большей простоты и ясности в этом рецепте будет использоваться упрощенный алгоритм наложения теней, без каких-либо усовершенствований. Этот алгоритм также будет лежать в основе последующих рецептов. Прежде чем перейти к программному коду, необходимо заметить, что, скорее всего, результаты получатся неудовлетворительные из-за отсутствия сглаживания краев теней в этом алгоритме. Однако его легко можно дополнить любым из множества приемов сглаживания. Некоторые из таких приемов будут представлены в следующих рецептах.

Подготовка

Координаты вершин должны передаваться в вершинный шейдер через атрибут с индексом `0`, а вектор нормали – через атрибут с индексом `1`. Также следует объявить и инициализировать uniform-переменные для передачи параметров модели затенения и стандартных матриц преобразований. Переменную `ShadowMatrix` следует инициализировать матрицей преобразования координат модели в координаты карты теней (S в предыдущем уравнении).

Необходимо также объявить и инициализировать uniform-переменную `ShadowMap` – дескриптор текстуры с картой теней, связанной с текстурным слотом `0`.

Как это делается...

Чтобы создать приложение OpenGL, отображающее тени с использованием карты теней, нужно выполнить следующие шаги. Сначала я покажу, как создать и настроить объект буфера кадра (Framebuffer Object, FBO), где будет храниться текстура с картой теней, а затем перейду к шейдерам:

1. Создать в основной программе объект буфера кадра с буфером глубины. Объявить переменную типа `GLuint` с именем `shadowFBO` для хранения дескриптора этого объекта. Создать объект текстуры, который будет играть роль буфера глубины. Сделать все это можно, как показано ниже:

```
GLfloat border[]={1.0f,0.0f,0.0f,0.0f};

// Текстура с картой теней
GLuint depthTex;
glGenTextures(1,&depthTex);
glBindTexture(GL_TEXTURE_2D,depthTex);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_DEPTH_COMPONENT24,
               shadowMapWidth, shadowMapHeight);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,
                GL_CLAMP_TO_BORDER);
glTexParameterfv(GL_TEXTURE_2D,GL_TEXTURE_BORDER_COLOR,
                 border);

glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_MODE,
                GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_FUNC,
                GL_LESS);

// Связать карту теней с текстурным слотом 0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D,depthTex);

// Создать и настроить FBO
glGenFramebuffers(1,&shadowFBO);
glBindFramebuffer(GL_FRAMEBUFFER,shadowFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER,GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D,depthTex,0);
GLenum drawBuffers[]={GL_NONE};
glDrawBuffers(1,drawBuffers);

// Сделать созданный буфер кадра буфером по умолчанию
glBindFramebuffer(GL_FRAMEBUFFER,0);
```

2. Добавить вершинный шейдер:

```
layout (location=0) in vec3 VertexPosition;
layout (location=1) in vec3 VertexNormal;

out vec3 Normal;
out vec3 Position;

// Координаты для выборки значения из карты теней
out vec4 ShadowCoord;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;
uniform mat4 ShadowMatrix;

void main()
```

```
{
  Position = (ModelViewMatrix *
              vec4(VertexPosition,1.0)).xyz;
  Normal = normalize( NormalMatrix * VertexNormal );

  // ShadowMatrix преобразует координаты модели
  // в координаты карты теней.
  ShadowCoord =ShadowMatrix * vec4(VertexPosition,1.0);

  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

3. Добавить фрагментный шейдер:

```
// Объявить переменные для параметров модели затенения
uniform sampler2DShadow ShadowMap;

in vec3 Position;
in vec3 Normal;
in vec4 ShadowCoord;

layout (location = 0) out vec4 FragColor;

vec3 diffAndSpec()
{
  // Вычислить только цвет рассеянного и
  // отраженного света.
}

subroutine void RenderPassType();
subroutine uniform RenderPassType RenderPass;

subroutine (RenderPassType)
void shadeWithShadow()
{
  vec3 ambient = ...;// вычислить цвет фонового освещения
  vec3 diffSpec = diffAndSpec();

  // Найти значение в карте теней
  float shadow = textureProj(ShadowMap, ShadowCoord);

  // Если фрагмент находится в тени,
  // использовать только цвет фонового освещения.
  FragColor = vec4(diffSpec * shadow + ambient, 1.0);
}

subroutine (RenderPassType)
void recordDepth()
{
  // Ничего не делать, глубина сохраняется автоматически
}

void main() {
  // Вызовет shadeWithShadow или recordDepth
  RenderPass();
}
```

В функции отображения в основной программе нужно выполнить следующие шаги.

Проход 1

1. Использовать матрицы вида, проекции и преобразования видимого объема, соответствующие источнику света.
2. Связать буфер кадра, содержащий карту теней (`shadowFBO`).
3. Очистить буфер глубины.
4. Выбрать подпрограмму `recordDepth`.
5. Включить удаление лицевых поверхностей.
6. Нарисовать сцену.

Проход 2

1. Использовать матрицы вида, проекции и преобразования видимого объема, соответствующие сцене.
2. Вернуться к использованию буфера кадра по умолчанию.
3. Выключить удаление лицевых поверхностей (или включить удаление обратных поверхностей).
4. Выбрать подпрограмму `shadeWithShadow`.
5. Нарисовать сцену.

Как это работает...

Первый блок кода в предыдущем списке создает объект буфера кадра для хранения текстуры с картой теней. Этот объект содержит только текстуру, играющую роль буфера глубины. Первые несколько строк кода создают текстуру для хранения карты теней. Текстура создается с помощью функции `glTexStorage2D` и с внутренним форматом `GL_DEPTH_COMPONENT24`.

Для фильтров `GL_TEXTURE_MAG_FILTER` и `GL_TEXTURE_MIN_FILTER` выбирается алгоритм интерполяции `GL_NEAREST`. Здесь можно было бы использовать алгоритм `GL_LINEAR` и получить более привлекательный результат, но я использовал `GL_NEAREST`, чтобы сделать более отчетливыми артефакты, вызванные отсутствием сглаживания, и чтобы немного улучшить производительность.

Далее для режимов `GL_TEXTURE_WRAP_*` устанавливается алгоритм `GL_CLAMP_TO_BORDER`. Если попадется фрагмент, находящийся за границами карты теней (за пределами объема, видимого с позиции источника света), координаты текстуры для такого фрагмента получатся больше единицы или меньше нуля. Подобные фрагменты не должны затеняться. При использовании алгоритма `GL_CLAMP_TO_BORDER` операция поиска в текстуре (для координат за пределами диапазона `[0..1]`) вернет значение, находящееся на границе. По умолчанию значение границы равно `(0,0,0,0)`. Когда текстура используется как карта теней, первый компонент каждого значения в ней хранит глубину. Нулевое значение глубины не позволит получить желаемый результат, потому что нулевая глубина соответствует точке на ближней плоскости отсечения. Соответственно, все точки поверхностей объектов, находящиеся за пределами объема, видимого с позиции источника света, будут

интерпретироваться как затененные! Чтобы этого не произошло, цвет на границе устанавливается в значение (1,0,0,0) вызовом функции glTexParameterfv, что соответствует максимально возможной глубине.

Следующие два вызова glTexParameteri определяют настройки, характерные для текстур с картами глубин. Первый вызов выбирает алгоритм GL_COMPARE_REF_TO_TEXTURE для режима GL_TEXTURE_COMPARE_MODE. Когда используется этот алгоритм, при обращении к текстуре возвращается результат сравнения, а не цвет: третий компонент координат текстуры (компонент p) сравнивается со значением, хранящимся в позиции (s,t). Результат возвращается в виде единственного вещественного значения. Функция сравнения определяется значением GL_TEXTURE_COMPARE_FUNC, которое устанавливается в следующей строке. В данном случае настраивается использование функции GL_LESS, то есть результатом сравнения будет значение 1.0, если значение p координат текстуры окажется меньше значения, хранимого в ячейке с координатами (s,t). (Имеются также другие функции: GL_LEQUAL, GL_ALWAYS, GL_GEQUAL и т. д.)

В следующих нескольких строках создается и настраивается объект буфера кадра. Вызовом функции glFramebufferTexture2D текстура с картой теней подключается к объекту, к точке подключения буфера глубины. Подробнее об объектах буфера кадра рассказывается в рецепте «Отображение в текстуру» в главе 4 «Текстуры».

Вершинный шейдер имеет очень простую реализацию. Он преобразует позицию вершины и нормаль в систему видимых координат (с началом в позиции камеры) и передает их фрагментному шейдеру через выходные переменные Position и Normal. Также с помощью матрицы ShadowMatrix позиция вершины преобразуется в координаты карты теней. Это та самая матрица *S*, о которой рассказывалось в предыдущем разделе. Она описывает преобразование позиции из координат модели в координаты тени. Результат передается фрагментному шейдеру через выходную переменную ShadowCoord.

Кроме того, как обычно, позиция преобразуется в усеченные координаты, и результат присваивается встроенной выходной переменной gl_Position.

Фрагментный шейдер действует по-разному в разных проходах. Функция main вызывает RenderPass – подпрограмму, которая, в свою очередь, вызывает recordDepth или shadeWithShadow. В первом проходе (когда создается карта теней) при вызове подпрограммы выполняется функция recordDepth. Эта функция вообще ничего не делает! Она существует только потому, что нужно обеспечить сохранение глубин в буфере глубины. Реализация OpenGL делает это автоматически (при условии что переменная gl_Position получила верное значение в вершинном шейдере), поэтому на долю фрагментного шейдера ничего не остается.

Во втором проходе выполняется функция shadeWithShadow. Она вычисляет цвет, обусловленный фоновым освещением, и сохраняет результат в переменной ambient. Затем вычисляется цвет рассеянного и отраженного света и сохраняется в переменной diffuseAndSpec.

Следующий шаг является ключевым в алгоритме наложения теней. Он выполняется вызовом встроенной функции textureProj доступа к текстуре ShadowMap

с картой теней. Прежде чем воспользоваться координатами для доступа к текстуре, функция textureProj разделит первые три компонента координат на четвертый. Напомню, что это именно та операция, которую требуется выполнить для преобразования однородных координат (ShadowCoord) в истинные декартовы координаты.

После перспективного деления функция textureProj использует результат для доступа к текстуре. Так как текстура имеет тип sampler2DShadow, она интерпретируется как текстура, хранящая значения глубины, поэтому при обращении к ней возвращается результат сравнения, а не фактическое значение из текстуры. Первые два компонента из переменной ShadowCoord используются для извлечения значения глубины из текстуры. Затем это значение сравнивается со значением третьего компонента в переменной ShadowCoord. В режиме интерполяции GL_NEAREST (как в данном случае) результатом будет значение 1.0 или 0.0. Так как была настроена функция сравнения GL_LESS, значение 1.0 будет возвращаться, если третий компонент ShadowCoord окажется меньше значения глубины в текстуре. Этот результат сохраняется в переменной shadow. В заключение выполняется присваивание значения выходной переменной FragColor. Результат сравнения с картой теней (shadow) умножается на цвет отраженного и рассеянного света, и к произведению прибавляется цвет фонового света. Когда переменная shadow получает значение 0.0 (сравнение потерпело неудачу), это означает, что между данным фрагментом и источником света имеется преграда. Поэтому фрагмент получает только цвет, обусловленный фоновым освещением. В противном случае в формировании цвета фрагмента принимают участие все три компонента освещения.

Обратите внимание, что перед созданием карты теней был включен режим удаления лицевых поверхностей. Сделано это с целью избежать Z-конфликтов, которые могут возникать при включении лицевых поверхностей в карту теней. Имейте в виду, что этот прием работает, только если отображаемые объекты полностью закрыты. Если предусматривается отображение обратных поверхностей, может потребоваться использовать другой прием (например, с применением функции glPolygonOffset). Я расскажу подробнее об этом в следующем разделе.

И еще...

Прием наложения теней, описанный в этом рецепте, имеет множество сопутствующих проблем. Давайте рассмотрим только самые бросающиеся в глаза.

Искажение границ

Как отмечалось выше, данный алгоритм страдает проблемой появления артефактов на границах теней. Это обусловлено тем, что карта теней, по сути, проецируется на сцену, когда выполняется сравнение со значением в глубины. Если проецирование карты сопряжено с увеличением размеров теней, появляются артефакты в виде зубчатых границ.

На рис. 7.3 показан эффект искажения границ теней.

Простейшее решение состоит в том, чтобы просто увеличить размеры текстуры с картой теней. Однако это может оказаться невозможно из-за ограниченного объ-

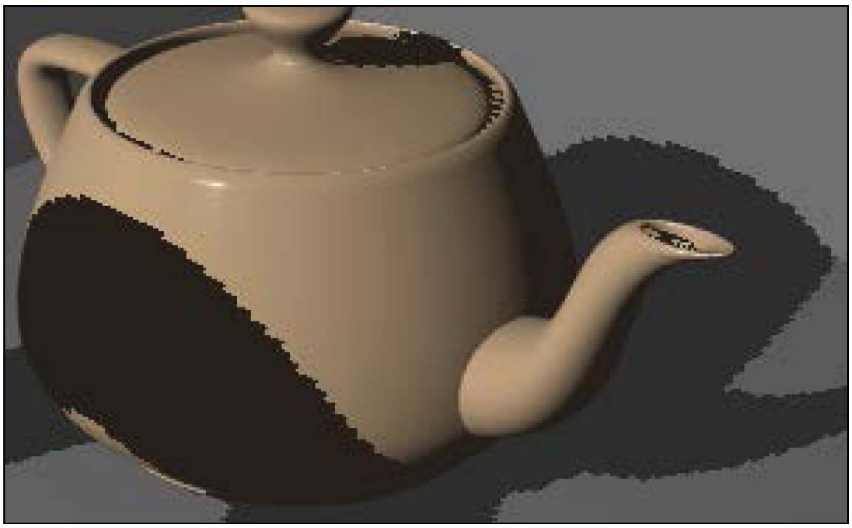

Рис. 7.3 ❖ Эффект искажения границ теней

ема памяти, недостаточного быстродействия CPU или по каким-то другим причинам. Существует огромное число приемов повышения качества изображений, производимых алгоритмом на основе карт теней, таких как карты теней с соответствующими разрешениями (resolution-matched shadow maps), каскадные карты теней (cascaded shadow maps), дисперсные карты теней (variance shadow maps), перспективные карты теней (perspective shadow maps) и многие другие. В следующем рецепте мы познакомимся с некоторыми приемами, помогающими смягчить и сгладить границы теней.

Создание карт теней за счет отображения только обратных поверхностей

Создавая карту теней, рецепт отображал только обратные поверхности. Сделано так потому, что при отображении лицевых поверхностей некоторые точки могут иметь глубину, незначительно отличающуюся от глубины или совпадающую с ней в карте теней, что может вызывать ошибочное затенение в действительности освещенных точек. На рис. 7.4 показан пример такого эффекта.

Так как такому неприятному эффекту подвержено большинство поверхностей, обращенных к источнику света, отображая только обратные поверхности в проходе создания карты теней, нам практически удалось избежать проблемы. Однако подобный подход можно использовать, только когда объекты полностью закрыты. Если это не так, исправить ситуацию можно с помощью функции `glPolygonOffset`, смещая глубину поверхностей, чтобы увеличить разность с картой теней. Фактически, даже когда для создания карты теней отображаются только обратные поверхности, подобные артефакты могут возникать на поверхностях, повернутых

Рис. 7.4 ❖ Эффект ошибочного затенения в действительности освещенных точек

в сторону от источника света (обратных по отношению к источнику света, но лицевых по отношению к камере). Поэтому на практике весьма часто используется комбинация двух приемов – на основе отключения отображения лицевых поверхностей и использования функции `glPolygonOffset`.

См. также

- Рецепт «Отображение в текстуру» в главе 4 «Текстуры».
- Рецепт «Сглаживание границ теней методом PCF».
- Рецепт «Смягчение границ теней методом случайной выборки».

Сглаживание границ теней методом PCF

Чаще других для сглаживания границ теней используется прием **фильтрации процентом более близких (Percentage-Closer Filtering, PCF)**. Такое название было выбрано потому, что суть приема заключается в том, чтобы выбрать фрагменты в области, окружающей текущий, определить процент фрагментов, глубина которых (относительно источника света) меньше глубины текущего фрагмента, и использовать этот процент для вычисления цвета (рассеянного и отраженного компонентов). В результате получается эффект размытия границы тени.

Впервые описание этого приема было опубликовано Ривесом (Reeves) с соавторами в 1987 году («SIGGRAPH Proceedings, Volume 21, Number 4, July 1987»). Суть заключается в преобразовании области, окружающей фрагмент, в теневое пространство, выборке нескольких фрагментов из этой области и вычислении процента фрагментов, более близких к источнику света, чем фрагмент. Результат затем используется для уменьшения затенения. С увеличением размера области фильтрации возникает эффект размытия границ теней.

На практике часто используется разновидность алгоритма PCF, основанного на выборке постоянного числа ближайших текселей из карты теней. Процент тексе-

лей, оказавшихся ближе к источнику света, чем текущий, используется для ослабления затенения. Получаемый эффект, возможно, не точно соответствует тому, что наблюдается в действительности, но он не вызывает отторжения.

На рис. 7.5 показаны тени, полученные с применением приема PCF (справа) и без него (слева). Отметьте, что тени на изображении справа имеют размытые границы, благодаря чему зубчатость не так бросается в глава.

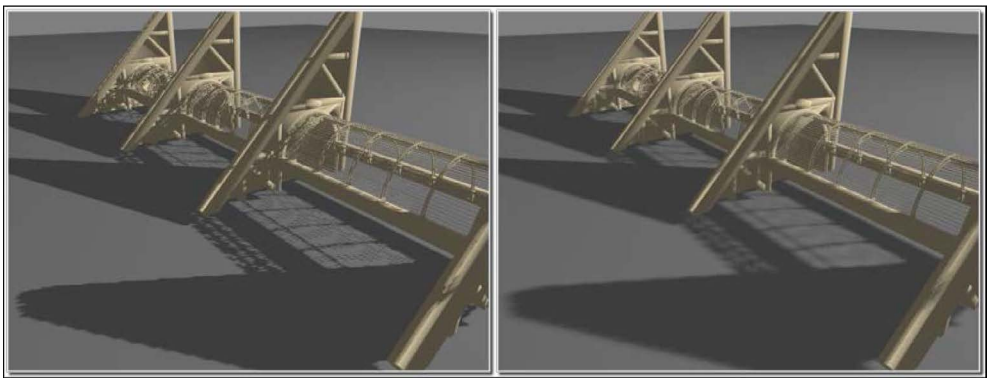

Рис. 7.5 ❖ Тени, полученные с применением приема PCF (справа) и без него (слева)

В данном рецепте будет использоваться прием PCF с выборкой постоянного числа текселей из карты теней, ближайших к текущему. Мы будем вычислять среднее из результатов сравнения и использовать это значение для масштабирования рассеянного и отраженного компонентов освещения.

Мы воспользуемся встроенной в OpenGL поддержкой PCF, задействовав линейную фильтрацию по текстуре с глубинами. Когда линейная фильтрация применяется к текстурам подобного типа, аппаратура способна автоматически отбирать четыре соседних текселя (выполняя четыре сравнения глубин) и усреднять результаты (конкретные детали работы алгоритма зависят от реализации). То есть когда включена линейная фильтрация, результатом функции `textureProj` может быть любое значение между `0.0` и `1.0`.

Мы также воспользуемся встроенными функциями для доступа к текстуре со смещениями. В OpenGL имеется функция `textureProjOffset`, третий параметр которой (смещение) прибавляется к координатам текселя перед выборкой/сравнением.

Подготовка

За основу можно взять программу из предыдущего рецепта «Отображение теней с помощью карты теней». В нее потребуется внести лишь несколько небольших изменений.

Как это делается...

Чтобы добавить сглаживание методом PCF к алгоритму наложения теней, нужно выполнить следующие шаги:

1. При подготовке объекта буфера кадра с картой теней необходимо включить линейную фильтрацию для текстуры. Замените соответствующие строки следующим кодом:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR);
```

2. Измените код функции `shadeWithShadow` внутри фрагментного шейдера, как показано ниже:

```
subroutine (RenderPassType)
void shadeWithShadow()
{
  vec3 ambient = vec3(0.2);
  vec3 diffSpec = diffAndSpec();

  // Сумма сравнений с ближайшими текселями
  float sum = 0;

  // Сумма вкладов текселей, окружающих ShadowCoord
  sum += textureProjOffset(ShadowMap, ShadowCoord,
                           ivec2(-1,-1));
  sum += textureProjOffset(ShadowMap, ShadowCoord,
                           ivec2(-1,1));
  sum += textureProjOffset(ShadowMap, ShadowCoord,
                           ivec2(1,1));
  sum += textureProjOffset(ShadowMap, ShadowCoord,
                           ivec2(1,-1));
  float shadow = sum * 0.25;

  FragColor = vec4(ambient + diffSpec * shadow,1.0);
}
```

Как это работает...

Первый шаг включает линейную фильтрацию текстуры с картой теней. Когда она включена, драйвер OpenGL выполнит сравнение глубин четырех соседних текселей. Результат четырех сравнений будет усреднен и возвращен вызывающему коду.

Для выборки четырех соседних для `ShadowCoord` текселей (по диагонали) внутри фрагментного шейдера используется функция `textureProjOffset`. Третий аргумент этой функции определяет смещение. Оно добавляется к координатам текселя (не к координатам текстуры) перед выборкой.

Так как включена линейная фильтрация, каждая операция чтения из текстуры приведет к выборке четырех текселей. То есть всего будет выбрано 16 текселей. Результаты усредняются и сохраняются в переменной `shadow`.

Как и прежде, значение `shadow` используется для ослабления рассеянного и отраженного компонентов в модели освещения.

И еще...

Фабио Пеллачини (Fabio Pellacini), сотрудник компании Pixar, написал отличный обзор приема PCF, который можно найти в главе 11 «Shadow Map Anti-aliasing» книги «GPU Gems», вышедшей под редакцией Рандима Фернандо (Randima Fernando) (Addison-Wesley Professional, 2004). Если вы хотите получить более подробные сведения о данном приеме, я настоятельно рекомендую прочитать эту короткую, но весьма информативную главу.

Благодаря простоте и эффективности прием PCF чрезвычайно широко используется для сглаживания границ теней, воспроизводимых с помощью карт теней. Так как он создает эффект размытия границ теней, его можно также использовать для имитации «мягких теней». Но для этого необходимо увеличить размер выборки в соответствии с размером размытого края (полутени), чтобы избежать появления некоторых артефактов. Из-за этого может ухудшиться общая производительность. В следующем рецепте я покажу прием создания мягких теней методом случайной выборки из большой области.

Полутенью называют область тени, где освещенность ослабляется лишь частично.

См. также

- Рецепт «Отображение теней с помощью карты теней».

Смягчение границ теней методом случайной выборки

Упрощенный алгоритм наложения теней в сочетании с приемом фильтрации PCF может воспроизводить тени с размытыми границами. Однако если потребуется получить размытый край значительной ширины (чтобы сымитировать по-настоящему мягкие тени), понадобится существенно увеличить площадь выборки. Кроме того, такой подход влечет ненужную трату вычислительных ресурсов на затенение фрагментов, находящихся в центре теней с большой площадью, или за пределами теней. Для таких фрагментов процедура определения значений текселей в карте теней будет возвращать один и тот же результат. Именно поэтому вычислительные ресурсы, направленные на извлечение этих текселей и усреднение результатов сравнения, по сути, будут расходоваться впустую.

Прием, который будет представлен в этом рецепте, основан на главе из книги «GPU Gems 2», вышедшей под редакцией Мэтта Фарра (Matt Pharr) и Рандима Фернандо (Randima Fernando) в издательстве Addison-Wesley Professional, 2005 (глава 17, написанная Юрием Уральским (Yury Uralsky)). В ней описывается метод отображения теней с размытыми краями произвольной ширины, решающий

обе проблемы, упомянутые выше: он не требует обращаться к текстуре внутри и за пределами тени.

Ниже перечислены основные идеи метода:

- вместо выборки текселей, окружающих данную точку (в карте теней), с использованием множества постоянных смещений предлагается использовать случайный, круговой шаблон;
- сначала осуществляется выборка текселей только во внешнем кольце, чтобы определить, не находится ли фрагмент за пределами тени или внутри нее.

На рис. 7.6 показан один из возможных шаблонов выборки текселей из карты теней. В центре находится испытуемый фрагмент, а крестиками (×) обозначены тексели, выбираемые для усреднения. Они распределены случайно в ячейках круговой сетки, окружающих фрагмент (не более одного текселя на одну ячейку сетки).

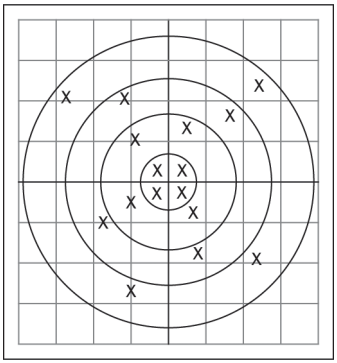

Рис. 7.6 ❖ Один из возможных шаблонов выборки текселей из карты теней

Изменение смещений выбираемых текселей организуется за счет предварительного создания множественного шаблона выборки: перед отображением вычисляется набор случайных смещений и сохраняется в текстуре. Затем внутри фрагментного шейдера, сначала из текстуры шаблона, извлекается множество смещений, после чего они применяются для выборки текселей из карты теней. Результаты усредняются, как это делается в алгоритме PCF.

На рис. 7.7 показаны различия в отображении теней с использованием алгоритма PCF (слева) и метода случайной выборки (справа), описываемого в данном рецепте.

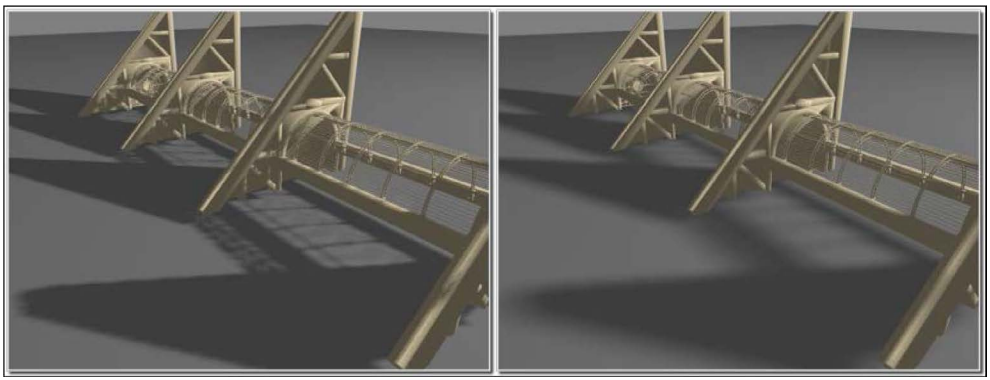

Рис. 7.7 ❖ Различия в отображении теней с использованием алгоритма PCF (слева) и метода случайной выборки (справа)

Смещения в этом рецепте будут храниться в трехмерной текстуре ($n \times n \times d$). Первые два измерения имеют произвольный размер, а третье определяет число смещений. Каждой позиции (s,t) соответствует список (с длиной d) случайных смещений, упакованных в структуру RGBA цвета. Каждое значение RGBA цвета в текстуре хранит два двухмерных смещения. Каналы R и G хранят смещение первого выбираемого текселя выборки, а каналы B и A – второго. То есть каждой позиции (s,t) соответствует 2*d смещений. Например, позиции с координатами (1, 1, 3) соответствуют шестое и седьмое смещения в позиции (1,1). Все множество значений для заданной позиции (s,t) включает полный набор смещений.

Выборка смещений из текстуры будет осуществляться по экранным координатам фрагмента. Координаты в текстуре смещений определяются как остаток от деления экранных координат на размеры текстуры. Например, для фрагмента с координатами (10.0,10.0) и текстуры с размерами (4,4) смещения будут выбираться из позиции (2,2).

Подготовка

Возьмите за основу приложение из рецепта «Отображение теней с помощью карты теней».

Добавьте три дополнительные uniform-переменные:

- `OffsetTexSize`: определяет ширину, высоту и глубину текстуры со смещениями. Обратите внимание, что значение глубины соответствует числу текселей, выбираемых для анализируемого фрагмента, деленному на два;
- `OffsetTex`: дескриптор текстурного слота, где хранится текстура со смещениями;
- `Radius`: радиус области размытия в пикселях, деленный на размер текстуры с картой теней (предполагается, что используется квадратная карта теней). Это значение может служить своеобразным коэффициентом мягкости тени.

Как это делается...

Чтобы изменить алгоритм наложения теней и задействовать метод случайной выборки, нужно выполнить следующие шаги. Текстура со смещениями будет создаваться в основном приложении и использоваться внутри фрагментного шейдера:

1. Добавить следующий код в основное приложение, чтобы обеспечить создание текстуры со смещениями. Он должен выполняться только один раз, на этапе инициализации программы:

```
void buildOffsetTex(int size, int samplesU, int samplesV)
{
  int samples = samplesU * samplesV;
  int bufSize = size * size * samples * 2;
  float *data = new float[bufSize];

  for( int i = 0; i< size; i++ ) {
    for(int j = 0; j < size; j++ ) {
      for( int k = 0; k < samples; k += 2 ) {
```

```
        int x1,y1,x2,y2;
        x1 = k % (samplesU);
        y1 = (samples - 1 - k) / samplesU;
        x2 = (k+1) % samplesU;
        y2 = (samples - 1 - k - 1) / samplesU;

        vec4 v;
        // Центр сетки и случайное смещение
        v.x = (x1 + 0.5f) + jitter();
        v.y = (y1 + 0.5f) + jitter();
        v.z = (x2 + 0.5f) + jitter();
        v.w = (y2 + 0.5f) + jitter();

        // Преобразовать в диапазон от 0 до 1
        v.x /= samplesU;
        v.y /= samplesV;
        v.z /= samplesU;
        v.w /= samplesV;

        // "Завернуть" сетку в круг
        int cell = ((k/2) * size * size + j *
                    size + i) * 4;
        data[cell+0] = sqrtf(v.y) * cosf(TWOPI*v.x);
        data[cell+1] = sqrtf(v.y) * sinf(TWOPI*v.x);
        data[cell+2] = sqrtf(v.w) * cosf(TWOPI*v.z);
        data[cell+3] = sqrtf(v.w) * sinf(TWOPI*v.z);

      }
    }
  }

  glActiveTexture(GL_TEXTURE1);
  GLuint texID;
  glGenTextures(1, &texID);

  glBindTexture(GL_TEXTURE_3D, texID);
  glTexStorage3D(GL_TEXTURE_3D, 1, GL_RGBA32F, size, size,
                 samples/2);
  glTexSubImage3D(GL_TEXTURE_3D, 0, 0, 0, 0, size, size,
                  samples/2, GL_RGBA, GL_FLOAT, data);
  glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER,
                  GL_NEAREST);
  glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER,
                  GL_NEAREST);

  delete [] data;
}

// Возвращает случайное вещественное число
// в диапазоне от -0.5 до 0.5
float jitter() {
  return ((float)rand() / RAND_MAX) - 0.5f;
}
```

2. Добавить объявления uniform-переменных во фрагментный шейдер:

```
uniform sampler3D OffsetTex;
uniform vec3 OffsetTexSize; // (ширина, высота, глубина)
uniform float Radius;
```

3. Изменить реализацию функции shadeWithShadow внутри фрагментного шейдера:

```
subroutine (RenderPassType)
void shadeWithShadow()
{
  vec3 ambient = vec3(0.2);
  vec3 diffSpec = diffAndSpec();

  ivec3 offsetCoord;
  offsetCoord.xy = ivec2( mod( gl_FragCoord.xy,
                              OffsetTexSize.xy ) );

  float sum = 0.0;
  int samplesDiv2 = int(OffsetTexSize.z);
  vec4 sc = ShadowCoord;

  for( int i = 0 ; i < 4; i++ ) {
    offsetCoord.z = i;
    vec4 offsets = texelFetch(OffsetTex,offsetCoord,0) *
                     Radius * ShadowCoord.w;
    sc.xy = ShadowCoord.xy + offsets.xy;
    sum += textureProj(ShadowMap, sc);
    sc.xy = ShadowCoord.xy + offsets.zw;
    sum += textureProj(ShadowMap, sc);
  }
  float shadow = sum / 8.0;

  if( shadow != 1.0 && shadow != 0.0 ) {
    for( int i = 4; i < samplesDiv2; i++ ) {
      offsetCoord.z = i;
      vec4 offsets =
        texelFetch(OffsetTex, offsetCoord,0) *
                   Radius * ShadowCoord.w;

      sc.xy = ShadowCoord.xy + offsets.xy;
      sum += textureProj(ShadowMap, sc);
      sc.xy = ShadowCoord.xy + offsets.zw;
      sum += textureProj(ShadowMap, sc);
    }
    shadow = sum / float(samplesDiv2 * 2.0);
  }
  FragColor = vec4(diffSpec * shadow + ambient, 1.0);
}
```

Как это работает...

Функция buildOffsetTex создает трехмерную текстуру со случайными смещениями. Первый параметр texSize определяет ширину и высоту текстуры. Чтобы соз-

дать изображения на рис. 7.7, я использовал значение 8. Второй и третий параметры, samplesU и samplesV, определяют число текселей, выбираемых из текстуры, в направлениях u и v. Я использовал значения 4 и 8 соответственно, то есть общий объем выборки составляет 32 текселя. Направления **u** и **v** – это произвольные оси, определяющие сетку смещений. Чтобы проще было понять суть, взгляните на рис. 7.8.

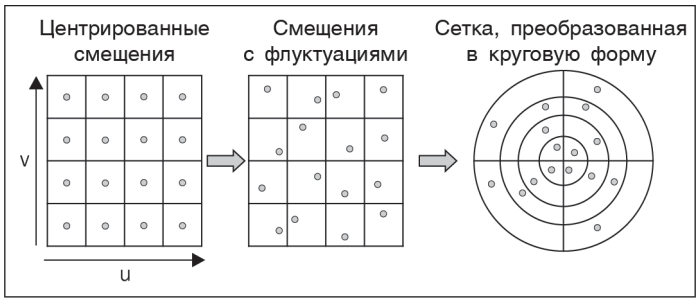

Рис. 7.8 ❖ Сетка смещений

Сначала в сетке с размерами samplesU × samplesV (4×4, см. рис. 7.8) определяются центрированные смещения. Координаты смещений масштабируются так, что вся сетка умещается в единичный квадрат (со стороной 1) с началом координат в левом нижнем углу. Затем каждая точка случайно смещается в пределах своей ячейки. В заключение сетка преобразуется в круговую форму («заворачивается») так, чтобы начало координат оказалось в центре, как показано на рис. 7.8.

На последнем шаге координата v интерпретируется как расстояние от начала координат, а координата u – как угол в диапазоне от 0 до 360 градусов. Вычисления выполняются по следующим формулам:

$$w_x = \sqrt{v}\cos(2\pi u);$$
$$w_y = \sqrt{v}\sin(2\pi u),$$

где w – координата в круговой сетке. В результате получается множество смещений, окружающих начало координат и находящихся от него на расстоянии не более 1.0. Кроме того, данные генерируются так, чтобы первыми следовали точки, находящиеся ближе к внешнему краю круга, а последними – ближние к центру. Это поможет избежать оценки слишком большого числа текселей, когда исследуемый фрагмент находится далеко от края тени, по ту или иную сторону.

Конечно же, смещения упаковываются так, что единственный тексель в текстуре со смещениями содержит две точки. Это не является обязательным условием, но помогает экономить память, хотя и за счет усложнения программного кода.

Внутри фрагментного шейдера сначала вычисляется цвет фонового света, отдельно от рассеянного и отраженного. Доступ к текстуре со смещениями осуществляется по экранным координатам фрагмента (gl_FragCoord). Для этого находится остаток от деления координат фрагмента на размеры текстуры со смещениями.

Результат сохраняется в первых двух компонентах `offsetCoord`. В результате для соседних пикселей получаются разные наборы смещений. Третий компонент `offsetCoord` используется для доступа к паре смещений. Число обращений к текстуре, которые нужно выполнить, чтобы извлечь всю выборку, равно значению глубины этой текстуры, деленному на два, и хранится в переменной `samplesDiv2`. Извлечение смещений выполняется с помощью функции `texelFetch`. Она позволяет получить тексель, используя целочисленные координаты вместо нормализованных координат текстуры в диапазоне `0...1`.

Извлеченное смещение умножается на `Radius` и компонент `w` вектора `ShadowCoord`. Умножение на `Radius` просто масштабирует смещения, приводя их к диапазону от `0.0` до `Radius`. Умножение на компонент `w` необходимо потому, что смещения применяются с целью переноса координат `ShadowCoord`, которые являются однородными, и чтобы перенос был выполнен правильно, смещения нужно умножить на компонент `w`. Влияние компонента `w` будет устранено позднее, операцией перспективного деления.

Далее смещения применяются к `ShadowCoord`, и по полученным координатам выполняется доступ к карте теней для сравнения глубин с помощью `textureProj`. Эта операция выполняется для обеих пар смещений, хранящихся в текселе, одна из которых хранится в двух первых компонентах вектора `offsets`, а вторая – в двух последних. Результаты добавляются в `sum`.

Первый цикл выбирает первые восемь смещений. Если для всех смещений получилось значение `0.0` или `1.0`, предполагается, что все точки в выборке находятся по одну сторону границы тени. В этом случае проверка других точек пропускается. В противном случае вычисляются значения в других точках, после чего они усредняются.

В заключение найденное среднее (`shadow`) используется для ослабления рассеянного и отраженного света.

И еще...

Использование небольшой текстуры со множеством случайно выбранных смещений помогает размыть границу тени лучше, чем с применением стандартного приема PCF со множеством постоянных смещений. Однако иногда на границах все еще можно наблюдать повторяющиеся артефакты из-за того, что текстура имеет конечные размеры и шаблоны смещений повторяются с шагом в несколько пикселей. Ситуацию можно улучшить, добавив поворот сетки со смещениями на случайный угол или просто вычисляя случайные смещения в самом шейдере.

Следует также отметить, что такое размытие границ теней может быть желательно не во всех случаях. Например, границы, которые совпадают с границами преград, то есть с границами поверхностей, отбрасывающих тень, должны быть резкими и неразмытыми. Такие границы могут быть видимы не всегда, но иногда, когда преграды достаточно узкие, подобные границы могут наблюдаться. Суть в том, чтобы заставить объект выглядеть так, как если бы он парил над поверхностью. К сожалению, эта задача не имеет простых решений.

См. также

- Рецепт «Отображение теней с помощью карты теней».

Создание теней с использованием приема теневых объемов и геометрического шейдера

Как отмечалось в предыдущих рецептах, одной из основных проблем, свойственных приему использования карт теней, является отсутствие сглаживания. Эта проблема обусловлена в основном несовпадением разрешения карт(ы) теней и сцены. Чтобы уменьшить искажения, можно попробовать реализовать сглаживание границ теней (как в предыдущих рецептах) или обеспечить совпадение разрешения карты теней с разрешением экранного пространства. Существует множество методик, позволяющих сделать это; познакомиться с которыми вы сможете, например, в книге «Real Time Shadows».

Альтернативный прием отображения теней называют **теневыми объемами (shadow volumes)**. Этот метод совершенно не подвержен проблеме появления артефактов, от которой страдает метод карты теней. Он воспроизводит точные и резкие тени, без каких-либо артефактов. На рис. 7.9 изображена сцена с тенями, полученными методом теневых объемов.

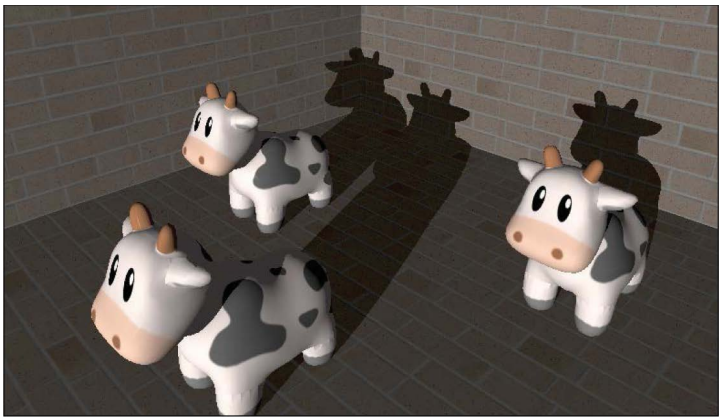

Рис. 7.9 ❖ Сцена с тенями, полученными методом теневых объемов

Метод теневых объемов основан на использовании трафаретного буфера для маскировки областей, попадающих в тень. Мы будем делать это путем рисования границ фактических теневых объемов (подробнее об этом рассказывается ниже). Теневой объем – это область пространства, доступ света в которую закрывает объект. Например, на рис. 7.10 показаны теневые объемы от треугольника (слева) и от шара (справа).

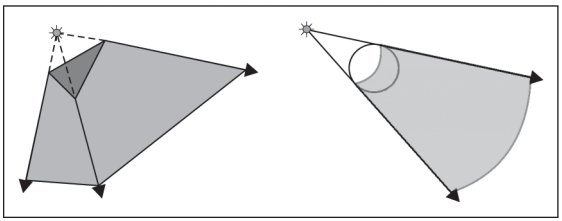

Рис. 7.10 ❖ Теневые объемы от треугольника (слева) и от шара (справа)

Боковые границы теневого объема образуются четырехугольниками, простирающимися от границ объекта, в направлении от источника света. Для единственного треугольника теневой объем формируется тремя четырехугольниками, простирающимися от сторон треугольника, и имеет вид трехгранной усеченной пирамиды. Одним основанием усеченной пирамиды является сам треугольник, а другое основание находится на некотором расстоянии в направлении от источника света. Для объектов, состоящих из множества треугольников, таких как шар на рис. 7.10, теневой объем определяется так называемыми границами силуэта. Эти границы образуются сторонами треугольников, находящимися на границе затененной и освещенной областей объекта или рядом с ней. В общем случае граница силуэта пролегает по границе между треугольниками, один из которых обращен к источнику света, а другой – в обратную сторону. Чтобы получить теневой объем, нужно найти все границы силуэта и нарисовать четырехугольники для каждой из них. Ближнее основание теневого объема можно определить как замкнутый полигон (или веер из треугольников – группа треугольников с одной общей для всех вершиной), включающий все точки границ силуэта. Аналогично конструируется дальнее основание объема.

Прием, основанный на создании теневых объемов, действует следующим образом. Представьте луч, простирающийся от точки местоположения камеры до пикселя на ближайшей поверхности. Допустим, что мы движемся по этому лучу и подсчитываем, сколько раз он войдет в теневой объем и выйдет из него, соответственно увеличивая и уменьшая счетчик. Подсчет прекращается по достижении точки поверхности. Мы можем с уверенностью сказать, что точка находится в тени, если счетчик имеет ненулевое значение, в противном случае точку можно считать освещенной. Эту идею иллюстрирует рис. 7.11.

Почти горизонтальная линия на рис. 7.11 представляет поверхность, на которую ложится тень. Числа под ней представляют значения счетчика для каждого луча, исходящего из камеры. Например, для второго справа луча значение счетчика равно +1, потому что луч дважды вошел в теневые объемы и один раз вышел, пока достиг поверхности: `1 + 1 - 1 = 1`. Для самого правого луча счетчик получил нулевое значение, потому что луч дважды вошел и вышел из обоих теневых объемов: `1 + 1 - 1 - 1 = 0`.

В теории все выглядит просто, но как на практике проследить движение луча? Самое интересное, что нам это не нужно! Все необходимое за нас сделает трафарет-

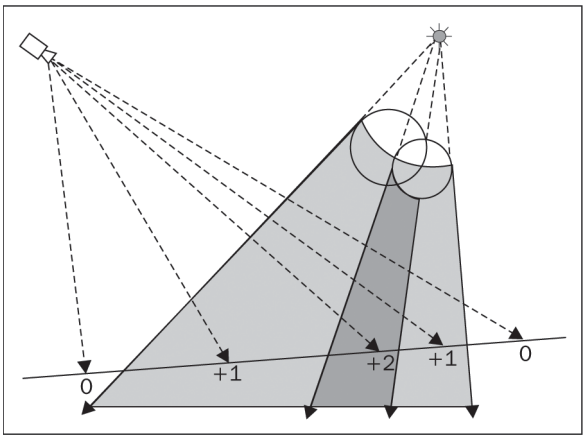

Рис. 7.11 ❖ Если счетчик больше нуля, точка поверхности находится в тени, иначе она считается освещенной

ный буфер. С его помощью можно увеличивать/уменьшать счетчики для каждого пикселя, исходя из того, находится ли он на лицевой или обратной поверхности. То есть мы можем нарисовать границы всех теневых объемов и затем увеличивать счетчик в буфере трафарета при отображении каждого пикселя лицевой поверхности и уменьшать при отображении пикселя обратной поверхности.

Здесь важно понять, что каждый пиксель в отображаемой фигуре представляет луч из камеры (как на рис. 7.11). То есть каждому данному пикселю в буфере трафарета соответствует значение, которое было бы получено, если бы мы фактически подсчитывали пересечения с границами объемов на пути к этому пикселю. Проверка глубины помогает остановить подсчет по достижении поверхности.

Имейте в виду, что это лишь краткое введение в прием на основе теневых объемов. Полное его обсуждение выходит далеко за рамки данной книги. Желающим получить более полное представление об этой методике я рекомендую обратиться к книге «Real Time Shadows» Эйсмана (Eisemann) и других авторов.

В этом рецепте будет показано, как нарисовать теневые объемы с помощью геометрического шейдера. Вместо вычисления теневых объемов на CPU мы отобразим геометрию сцены, как обычно, и в геометрическом шейдере воссоздадим теневые объемы. В рецепте «Рисование линий силуэта с помощью геометрического шейдера» в главе 6 «Использование геометрических шейдеров и шейдеров тесселяции» было показано, что вместе с каждым треугольником геометрический шейдер может получать информацию о смежных вершинах. Опираясь на эту информацию, можно определить, граничит ли данный треугольник с линией силуэта. Если треугольник обращен лицевой стороной в направлении от источника света, следовательно, одна из его сторон может считаться линией силуэта и использоваться для построения полигона основания теневого объема.

Все вычисления выполняются в три этапа:

- выполняется отображение сцены как обычно, но цвет фрагментов сохраняется в двух отдельных буферах. В одном сохраняется фоновый компонент, а в другом – сумма рассеянного и отраженного компонентов;
- буфер трафарета настраивается так, чтобы проверка трафарета всегда завершалась успехом, встреча с лицевой поверхностью вызывала увеличение счетчика, а встреча с обратной поверхностью – уменьшение счетчика. Буфер глубины делается доступным только для чтения, и отображаются лишь объекты, отбрасывающие тень. На этом этапе геометрический шейдер создает теневые объемы, фрагментный шейдер отображает только теневые объемы;
- буфер трафарета настраивается так, чтобы проверка трафарета завершалась успехом только для значений, равных нулю. Выполняется рисование прямоугольника, покрывающего всю плоскость экрана, и когда проверка трафарета завершается успехом, производится объединение значений цвета из двух буферов, созданных на первом этапе.

Так выглядит алгоритм в общих чертах, но в нем имеется множество нюансов. Давайте рассмотрим их поближе.

Подготовка

Сначала нужно создать буферы. С этой целью мы создадим объект буфера кадра с одним буфером глубины и двумя буферами цвета. Цвет фонового освещения можно сохранять сразу в буфер отображения (без использования текстуры), потому что это обеспечит более быстрое копирование в буфер кадра по умолчанию благодаря отсутствию операций чтения из текстуры. Сумма компонентов рассеянного и отраженного света будет сохраняться в текстуру. Итак, создайте буфер для хранения фонового компонента освещения (`ambBuf`), буфер глубины (`depthBuf`) и текстуру (`diffSpecTex`), затем настройте объект буфера кадра.

```
glGenFramebuffers(1, &colorDepthFBO);
glBindFramebuffer(GL_FRAMEBUFFER, colorDepthFBO);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER, depthBuf);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                          GL_RENDERBUFFER, ambBuf);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
                       GL_TEXTURE_2D, diffSpecTex, 0);
```

Свяжите буферы рисования с буферами цвета:

```
GLenum drawBuffers[] = {GL_COLOR_ATTACHMENT0,
                        GL_COLOR_ATTACHMENT1};
glDrawBuffers(2, drawBuffers);
```

Как это делается...

Перед первым проходом следует активировать объект буфера кадра, подготовленный выше, и отобразить сцену, как обычно. Внутри фрагментного шейдера сохра-

нить фоновый компонент в одну выходную переменную, а сумму рассеянного и отраженного компонентов – в другую.

```
layout( location = 0 ) out vec4 Ambient;
layout( location = 1 ) out vec4 DiffSpec;

void shade( )
{
  // Вычислить компоненты освещенности в соответствии с выбраной моделью
  // и сохранить фоновый компонент отдельно.
  Ambient = ...;  // Фоновый компонент
  DiffSpec = ...; // Рассеянный + отраженный
}
void main() { shade(); }
```

Во втором проходе будут отображаться теневые объемы. Для этого нужно настроить буфер трафарета так, чтобы проверка всегда завершалась успехом, встреча с лицевой поверхностью вызывала увеличение, а с обратной поверхностью – уменьшение счетчика.

```
glClear(GL_STENCIL_BUFFER_BIT);
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, 0, 0xffff);
glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_KEEP, GL_INCR_WRAP);
glStencilOpSeparate(GL_BACK, GL_KEEP, GL_KEEP, GL_DECR_WRAP);
```

Кроме того, в этом проходе необходимо задействовать буфер глубины, полученный в первом проходе, но при этом использовать буфер кадра по умолчанию, поэтому придется скопировать буфер глубины из объекта буфера кадра первого прохода. Также нужно скопировать информацию о цвете, содержащую значение фонового компонента.

```
glBindFramebuffer(GL_READ_FRAMEBUFFER, colorDepthFBO);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER,0);
glBlitFramebuffer(0,0,width-1,height-1,0,0,width-1,height-1,
           GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

В этом проходе следует предотвратить запись в буферы глубины и цвета, сделав их доступными только для чтения, потому что цель данного прохода заключается в изменении буфера трафарета:

```
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);
```

Далее нужно отобразить объекты, отбрасывающие тень с передачей информации о смежных вершинах. В геометрическом шейдере определяются границы силуэта и выводятся только четырехугольники, определяющие границы теневых объемов.

```
layout( triangles_adjacency ) in;
layout( triangle_strip, max_vertices = 18 ) out;

in vec3 VPosition[];
```

```
in vec3 VNormal[];

uniform vec4 LightPosition; // Позиция источника света (в видимых коорд.)
uniform mat4 ProjMatrix;    // Матрица проекции (с дальней плоскостью
                            // в бесконечности)

bool facesLight( vec3 a, vec3 b, vec3 c )
{
  vec3 n = cross( b - a, c - a );
  vec3 da = LightPosition.xyz - a;
  vec3 db = LightPosition.xyz - b;
  vec3 dc = LightPosition.xyz - c;
  return dot(n, da) > 0 || dot(n, db) > 0 || dot(n, dc) > 0;
}

void emitEdgeQuad( vec3 a, vec3 b ) {
  gl_Position = ProjMatrix * vec4(a, 1);
  EmitVertex();
  gl_Position = ProjMatrix * vec4(a - LightPosition.xyz, 0);
  EmitVertex();
  gl_Position = ProjMatrix * vec4(b, 1);
  EmitVertex();
  gl_Position = ProjMatrix * vec4(b - LightPosition.xyz, 0);
  EmitVertex();
  EndPrimitive();
}

void main()
{
  if( facesLight(VPosition[0], VPosition[2], VPosition[4]) ) {
    if( ! facesLight(VPosition[0],VPosition[1],VPosition[2]) )
      emitEdgeQuad(VPosition[0],VPosition[2]);
    if( ! facesLight(VPosition[2],VPosition[3],VPosition[4]) )
      emitEdgeQuad(VPosition[2],VPosition[4]);
    if( ! facesLight(VPosition[4],VPosition[5],VPosition[0]) )
      emitEdgeQuad(VPosition[4],VPosition[0]);
  }
}
```

В третьем проходе нужно настроить буфер трафарета так, чтобы проверка оканчивалась успехом только для нулевых значений в буфере.

```
glStencilFunc(GL_EQUAL, 0, 0xffff);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

Необходимо включить смешивание, чтобы обеспечить объединение фонового компонента освещения с рассеянным и отраженным компонентами, когда проверка завершается успехом.

```
glEnable(GL_BLEND);
glBlendFunc(GL_ONE,GL_ONE);
```

В этом проходе просто рисуется квадрат, покрывающий плоскость экрана. Если проверка значения в буфере трафарета завершается успехом, цвет пикселя ком-

бинируется из фонового компонента, который уже присутствует в буфере (был скопирован выше вызовом функции glBlitFramebuffer), и суммы рассеянного и отраженного компонентов.

```
layout(binding = 0) uniform sampler2D DiffSpecTex;
layout(location = 0) out vec4 FragColor;

void main() {
  vec4 diffSpec = texelFetch(DiffSpecTex, ivec2(gl_FragCoord), 0);
  FragColor = vec4(diffSpec.xyz, 1);
}
```

Как это работает...

Первый проход очень прост. В нем выполняется рисование сцены, как обычно, за исключением того, что цвет фонового света и сумма цветов рассеянного и отраженного света сохраняются в разных буферах.

Второй проход является ядром алгоритма. В нем отображаются объекты, отбрасывающие тень, и разрешается геометрическому шейдеру создать теневые объемы. Благодаря геометрическому шейдеру нам вообще не требуется отображать объекты – только их тени. Однако перед проходом нужно выполнить некоторые подготовительные операции. Вызовом glStencilOpSeparate мы настроили буфер трафарета так, чтобы при отображении лицевой поверхности значение в буфере увеличивалось, а при отображении обратной поверхности – уменьшалось, и вызовом glStencilFunc определили, что проверки трафарета всегда должны завершаться успехом. Мы также скопировали буфер глубины и буфер со значениями фонового цвета из объекта буфера кадра, задействованного в первом проходе, воспользовавшись функцией glBlitFramebuffer. Так как во втором проходе требуется отобразить только теневые объемы, не осложненные дополнительными геометрическими построениями, вызовом glDepthMask буфер глубины был сделан доступным только для чтения. Наконец, мы запретили запись в буфер цвета вызовом glColorMask, потому что в этом проходе не предусмотрена запись какой-либо информации в буфер цвета.

Геометрический шейдер конструирует теневые объемы по силуэтам. Так как отображение выполняется с использованием информации о смежных вершинах (см. рецепт «Рисование линий силуэта с помощью геометрического шейдера» в главе 6 «Использование геометрических шейдеров и шейдеров тесселяции»), геометрический шейдер получает доступ к шести вершинам, определяющим текущий отображаемый треугольник, и три соседних треугольника. Вершины нумеруются числами от 0 до 5 и в данном рецепте доступны через входной массив с именем VPosition. Вершины 0, 2 и 4 определяют текущий треугольник, а другие – смежные треугольники, как показано на рис. 7.12.

Сначала геометрический шейдер проверяет, повернут ли основной треугольник (0, 2, 4) лицевой стороной к источнику света. Для этого вычисляются вектор нормали к плоскости треугольника (n) и векторы из каждой вершины в сторону источника света, а затем скалярное произведение вектора нормали на каждый из

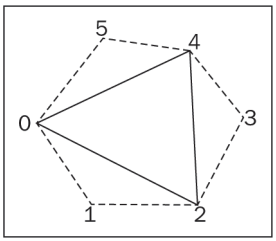

Рис. 7.12 ❖ Вершины 0, 2 и 4 определяют текущий треугольник, а другие – смежные треугольники

трех векторов к источнику света (`da`, `db` и `dc`). Если хотя бы один из трех результатов окажется положительным, следовательно, треугольник повернут лицевой стороной к источнику света, и в этом случае выполняется аналогичная проверка для соседних треугольников. Если окажется, что соседний треугольник повернут в направлении от источника света, границу между ними можно считать линией силуэта и использовать как основание боковой грани теневого объема.

Создание боковой грани теневого объема осуществляется функцией `emitEdgeQuad`. Точки `a` и `b` определяют линию силуэта – одну из сторон боковой грани теневого объема. Две другие вершины боковой грани определяются путем прокладывания прямых от источника света через точки `a` и `b`. Здесь используется математический трюк, возможный благодаря использованию однородных координат. Мы продолжаем грань в бесконечность за счет использования нулевого значения в компоненте `w` координат удаленных вершин. Таким способом определяется однородный вектор, который иногда называют точкой в бесконечности: координаты X, Y и Z определяют направление вектора от источника света, а компонент `w` устанавливается в значение `0`. В итоге получается четырехугольник, простирающийся в бесконечность в направлении от источника света.

Имейте в виду, что такой прием допустим только при использовании модифицированной матрицы проекции, учитывающей точки, определяемые описанным способом. По сути, матрица проекции должна определять дальнюю плоскость отсечения как находящуюся в бесконечности. Получить такую матрицу можно с помощью функции `infinitePerspective` из библиотеки GLM.

При использовании описываемого приема не приходится беспокоиться о рисовании вершин граней теневого объема – рисовать дальние вершины грани вообще не требуется благодаря использованию трюка с однородными координатами, описанного выше, а ближние вершины будут нарисованы автоматически, при отображении самого объекта.

В третьем и последнем проходе вызовом функции `glStencilFunc` буфер трафарета настраивается так, что проверка будет считаться успешной только для нулевых значений в буфере. В случае успеха проверки нужно объединить цвет фонового света с суммой рассеянного и отраженного света, поэтому разрешается поддержка смешивания и в качестве коэффициентов исходного и целевого цветов устанавли-

вается GL_ONE. Далее отображается единственный прямоугольник, покрывающий весь экран, и выводятся значения из текстуры, содержащей суммы цветов рассеянного и отраженного света. Проверка трафарета обеспечит пропуск фрагментов, находящихся в тени, а поддержка смешивания в OpenGL обеспечит объединение цветов фонового, рассеянного и отраженного света для фрагментов, прошедших проверку. (Напомню, что цвет фонового освещения был скопирован перед этим вызовом glBlitFramebuffer.)

И еще...

Методику, описанную выше, часто называют методикой **z-pass**. Она имеет один фатальный недостаток. Если камера оказывается внутри теневого объема, данный прием начинает давать ошибочные результаты, потому что счетчики в трафаретном буфере получат завышенные или заниженные значения, как минимум на единицу. На практике этот недостаток часто устраняется решением обратной задачи – трассировкой лучей из бесконечности к точке местоположения камеры. Эту методику называют методикой z-fail, или обращением Кармака (Carmack's reverse).

Слова «fail» и «pass» в названиях этих двух методик указывают, какой результат проверки глубины учитывается: «неудачный» или «удачный».

Применение приема z-fail требует особого внимания, потому что предполагает рисование всех вершин граней теневых объемов. Но в остальном он очень похож на прием z-pass, только значение счетчика увеличивается/уменьшается, когда проверка глубины терпит неудачу, а не когда завершается успехом, и он обеспечивает «трассировку» лучей из бесконечности к наблюдателю.

Следует также отметить, что предыдущий код не особенно устойчив к вырожденным треугольникам (треугольникам, имеющим почти параллельные стороны) или к незамкнутым поверхностям. Об этих ситуациях придется позаботиться отдельно. Например, чтобы обеспечить надежную обработку вырожденных треугольников, можно использовать какой-нибудь другой прием определения нормали к плоскости треугольника. А для обработки незамкнутых поверхностей добавить дополнительный код, обрабатывающий их края, или просто всегда использовать замкнутые поверхности.

См. также

- Рецепт «Рисование линий силуэта с помощью геометрического шейдера» в главе 6 «Использование геометрических шейдеров и шейдеров тесселяции».

Глава 8

Использование шума в шейдерах

В этой главе описываются следующие рецепты:

- создание текстуры шума с использованием GLM;
- создание бесшовной текстуры шума;
- создание эффекта облаков;
- создание эффекта текстуры древесины;
- создание эффекта разрушения;
- создание эффекта брызг краски;
- создание эффекта изображения в приборе ночного видения.

Введение

С помощью шейдеров легко можно воспроизводить гладкие поверхности, но это не всегда желательно. Иногда, для придания большей реалистичности, бывает необходимо смоделировать какие-либо недостатки, так присущие поверхностям в реальном мире. К таким недостаткам, например, можно отнести царапины, пятна ржавчины, вмятины и следы разрушения. Порой кажется удивительным, насколько сложно бывает заставить поверхность выглядеть так, как будто она подвергалась действию различных естественных процессов. Точно так же иногда бывает нужно отображать поверхности, напоминающие естественные материалы, такие как древесина, или воспроизводить природные явления, такие как облака, чтобы сцена не выглядела синтетической из-за присутствия повторяющихся фрагментов.

В большинстве случаев природные явления или поверхности демонстрируют некоторую степень случайности и нелинейности. Поэтому, как вы уже наверняка догадались, для их воспроизведения требуется использовать случайные данные. Однако данные, получаемые с помощью генератора псевдослучайных чисел, не особенно полезны в компьютерной графике по двум основным причинам:

- во-первых, последовательности данных должны повторяться, чтобы в процессе анимации сохранялся внешний вид объекта (добиться этого можно было бы за счет использования постоянного начального значения в каждом новом кадре, но это решает проблему лишь наполовину);
- во-вторых, для моделирования большинства природных явлений нужны длинные последовательности согласованных данных, которые тем не менее

выглядят достаточно случайными. Длинные и согласованные последовательности позволяют получить более реалистичное изображение многих естественных материалов и природных явлений. Просто случайные данные не обладают этим свойством. Каждое последующее случайное значение не зависит от предыдущего.

Благодаря работе Кена Перлина (Ken Perlin) в нашем распоряжении имеется понятие **шума** (или, точнее, «визуального шума», используемого в компьютерной графике). В его работе шум определяется как функция, обладающая определенными качествами, перечисленными ниже:

- она воспроизводит последовательности согласованных данных;
- она воспроизводит повторяемые последовательности (для одного и того же входного значения генерируется одна и та же последовательность);
- она может быть определена для любого числа измерений;
- она не имеет регулярно повторяющихся фрагментов и создает эффект случайности.

Такая функция шума – ценный инструмент в компьютерной графике, который можно использовать для создания бесконечного множества интересных эффектов. Например, в этой главе я покажу, как с помощью шума создавать облака, воспроизводить текстуру древесины, отображать эффект разрушения (рассыпания) и т. д.

Шум Перлина (Perlin noise) – это функция, которая впервые была определена Кевином Перлином (Ken Perlin) (см. **http://mrl.nyu.edu/~perlin/doc/oscar.html**). Однако мы не будем обсуждать ее во всех деталях, так как это выходит далеко за рамки данной книги.

На выбор имеются три варианта использования шума Перлина внутри шейдера:

1) задействовать встроенные функции, имеющиеся в языке GLSL;
2) написать собственные функции на языке GLSL;
3) использовать текстуру с предварительно вычисленными данными.

На момент написания этих строк в некоторых коммерческих драйверах OpenGL еще отсутствовали встроенные функции шума, поэтому я решил не использовать их в этой главе. Так как создание собственных функций выходит далеко за рамки этой книги, а также потому, что вариант 3 из списка выше обеспечивает наилучшую производительность, все рецепты в этой главе будут основаны на третьем подходе (с использованием текстур, вычисленных предварительно).

Во многих книгах по компьютерной графике демонстрируется использование трехмерных текстур шума вместо двухмерных, чтобы обеспечить воспроизведение трехмерных эффектов и явлений. Однако, чтобы упростить рецепты и сосредоточить ваше внимание на особенностях использования координат текстуры, я решил использовать в этой главе только двухмерные текстуры. При необходимости вы без труда сможете расширить эти рецепты на использование трехмерных текстур.

Для начала будут показаны два рецепта, демонстрирующих создание текстуры шума с помощью библиотеки GLM. Затем мы перейдем к рецептам, использующим подобные текстуры для создания таких эффектов, как текстура древесины, облака, статические разряды, брызги краски и эрозия.

Рецепты в этой главе создавались с целью служить отправной точкой для последующих экспериментов. Их не следует рассматривать как единственно правильное решение для любых из воспроизводимых ими эффектов. Самое замечательное в компьютерной графике – элемент творчества. Попробуйте изменять шейдеры в рецептах, чтобы добиться похожих результатов, а затем попробуйте воспроизвести собственные эффекты. И почаще развлекайтесь!

См. также...

- Книгу «Texturing and Modeling: A Procedural Approach» Кена Масгрейва (Ken Musgrave) с соавторами.

Создание текстуры шума с использованием GLM

Чтобы создать текстуру для использования в качестве источника шума, нужно каким-то способом сгенерировать данные. Реализация собственного генератора шума с подходящими характеристиками может оказаться весьма непростой задачей. К счастью, в библиотеке GLM имеется несколько простых и удобных функций, генерирующих шум.

В этом рецепте демонстрируется применение библиотеки GLM для создания двухмерной текстуры с использованием генератора **шума Перлина**. С помощью функции `glm::perlin` из библиотеки GLM можно создавать двух-, трех- и четырехмерный шум Перлина.

На практике часто используется прием суммирования данных, возвращаемых генератором шума Перлина, действующим с разными частотами и амплитудами. Интервалы частот генератора принято называть **октавами** (на интервале одной октавы частота удваивается). Например, на рис. 8.1 показаны результаты воспроизведения двухмерного шума Перлина для четырех разных октав (частота генератора увеличивается слева направо). На изображении слева используется базовая частота, а на каждом последующем – слева направо – частота удваивается.

Выражаясь языком математики, если функцию когерентного шума Перлина обозначить как $P(x, y)$, тогда каждое изображение на рис. 8.1 можно представить формулой

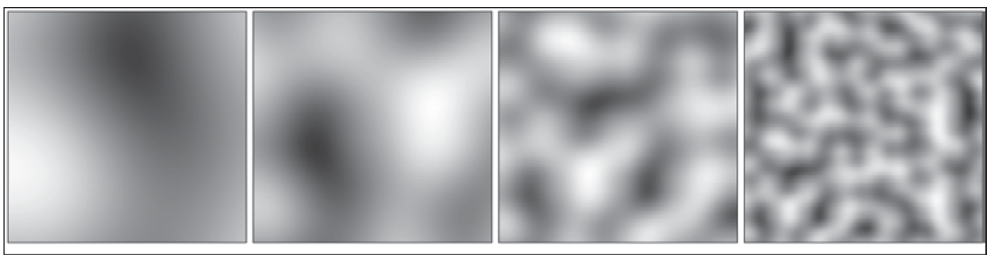

Рис. 8.1 ❖ Результаты воспроизведения двухмерного шума Перлина для четырех разных октав

$$n_i(x, y) = P(2^i x, 2^i y),$$

где $i = 0$, 1, 2 и 3 слева направо.

Как отмечалось выше, на практике окончательный результат часто получается путем сложения нескольких октав. Каждая последующая октава воспроизводится с уменьшенной амплитудой и добавляется к предыдущей. То есть для N октав получается следующая сумма:

$$n(x,y) = \sum_{i=0}^{N-1} \frac{P(2^i ax,\ 2^i ay)}{b^i},$$

где a и b – настраиваемые константы. На рис. 8.2 показаны изображения шума, полученные сложением 2, 3 и 4 октав (слева направо) с константами $a = 1$ и $b = 2$.

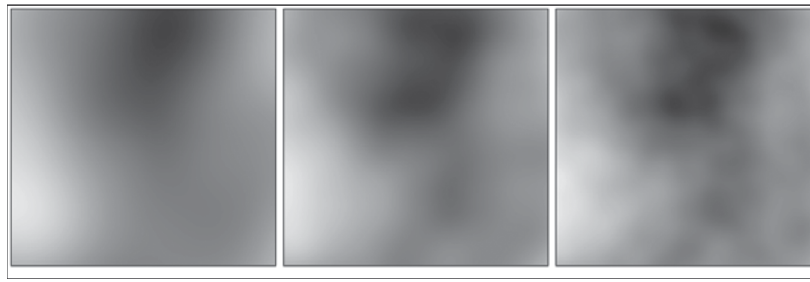

Рис. 8.2 ❖ Изображения шума, полученные сложением 2, 3 и 4 октав (слева направо)

При суммировании с более высокими октавами получается более высокочастотный шум, чем при суммировании с более низкими. Однако, увеличивая частоту, легко можно выйти за рамки разрешающей способности буфера, используемого для хранения шума, поэтому нужно проявлять особое внимание к этому, чтобы избежать напрасных вычислений. Умение определять границы – это и искусство, и наука. Предыдущую формулу можно использовать в качестве основы; попробуйте изменять ее параметры для достижения желаемого эффекта.

Мы будем хранить значения шума в единой двухмерной текстуре. Значения из одной октавы шума Перлина будут храниться в первом компоненте (канал красного цвета), из двух октав – во втором компоненте (канал зеленого цвета), из трех октав – в третьем (канал синего цвета), и из четырех – в четвертом (альфа-канал).

Подготовка

Проверьте, установлена ли библиотека GLM и доступны ли ее заголовочные файлы.

Как это делается...

Чтобы создать двухмерную текстуру шума с помощью библиотеки GLM, нужно выполнить следующие шаги:

1. Подключить заголовочный файл библиотеки, содержащий определения функций шума.

```
#include <glm/gtc/noise.hpp>
```

2. Сгенерировать данные, используя предыдущее уравнение:

```
GLubyte *data = new GLubyte[ width * height * 4 ];

float xFactor = 1.0f / (width - 1);
float yFactor = 1.0f / (height - 1);

for( int row = 0; row < height; row++ ) {
  for( int col = 0 ; col < width; col++ ) {
    float x = xFactor * col;
    float y = yFactor * row;
    float sum = 0.0f;
    float freq = a;
    float scale = b;

    // Вычислить суммы для каждой из октав
    for( int oct = 0; oct < 4; oct++ ) {
      glm::vec2 p(x * freq, y * freq);
      float val = glm::perlin(p) / scale;
      sum += val;
      float result = (sum + 1.0f)/ 2.0f;

      // Сохранить в текстуре
      data[((row * width + col) * 4) + oct] =
                  (GLubyte) ( result * 255.0f );
      freq *= 2.0f; // Удвоить частоту
      scale *= b;   // Следующая степень b
    }
  }
}
```

3. Загрузить данные в текстуру OpenGL.

```
GLuint texID;
glGenTextures(1, &texID);

glBindTexture(GL_TEXTURE_2D, texID);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, width, height);
glTexSubImage2D(GL_TEXTURE_2D,0,0,0,width,height,
  GL_RGBA,GL_UNSIGNED_BYTE,data);

delete [] data;
```

Как это работает...

Библиотека GLM позволяет генерировать двух-, трех- и четырехмерный когерентный шум с помощью функции `glm::perlin`. Эта функция возвращает вещественные значения в диапазоне от -1 до 1.

Сначала в программе создается буфер с именем `data` для хранения сгенерированных значений шума. Затем выполняется цикл по всем текселям и вычисляются

нормализованные координаты x и y. Далее выполняется цикл по октавам. Здесь вычисляется сумма, описываемая уравнением выше, при этом первое слагаемое сохраняется в первом компоненте, сумма двух первых слагаемых – во втором компоненте и т. д. Получаемое значение приводится к диапазону от 0 до 1, затем умножается на 255 и приводится к байту.

Следующие несколько строк кода должны быть понятны без дополнительных пояснений. Здесь выделяются память для текстуры (вызовом glTexStorage2D) и загрузка данных в память GPU (вызовом glTexSubImage2D).

В заключение массив data удаляется, так как он больше не нужен.

И еще...

Используя вещественные значения вместо однобайтовых целых, можно получить более высокое разрешение шума и более качественный результат с более высокой степенью детализации. Для этого достаточно внести в предыдущий код незначительные изменения: использовать внутренний формат GL_RGBA32F вместо GL_RGBA и убрать умножение на 255 перед сохранением значений в массив.

Библиотека GLM реализует также возможность получения периодического шума Перлина в виде перегруженной версии функции glm::perlin, упрощая тем самым создание текстур шума, которые могут стыковаться бесшовно. О том, как пользоваться этой версией функции, рассказывается в следующем рецепте.

См. также

- Обобщенные сведения о когерентном шуме в книге «Graphics Shaders» Майка Бейли (Mike Bailey) и Стива Каннингема (Steve Cunningham).
- Рецепт «Наложение двухмерной текстуры» в главе 4 «Текстуры».
- Рецепт «Создание бесшовной текстуры шума».

Создание бесшовной текстуры шума

Очень удобно иметь текстуру шума, не образующую швов при укладке ее мозаикой. Если просто создать текстуру шума с ограниченными размерами, ее противоположные края могут не согласовываться между собой. В результате при наложении такой текстуры на поверхность, текстурные координаты которой выходят за диапазон от нуля до единицы, границы (швы) между экземплярами текстуры будут бросаться в глаза.

К счастью, в библиотеке имеется периодическая версия функции, генерирующей шум Перлина, с помощью которой можно создавать бесшовные текстуры шума.

На рис. 8.3 показаны изображения, полученные повторением обычной (слева) и периодической (справа) текстуры 4-октавного шума Перлина. Обратите внимание, что на левом изображении отчетливо видны швы, тогда как на правом изображении они незаметны.

В данном примере мы изменим код из предыдущего рецепта так, чтобы он производил бесшовную текстуру шума.

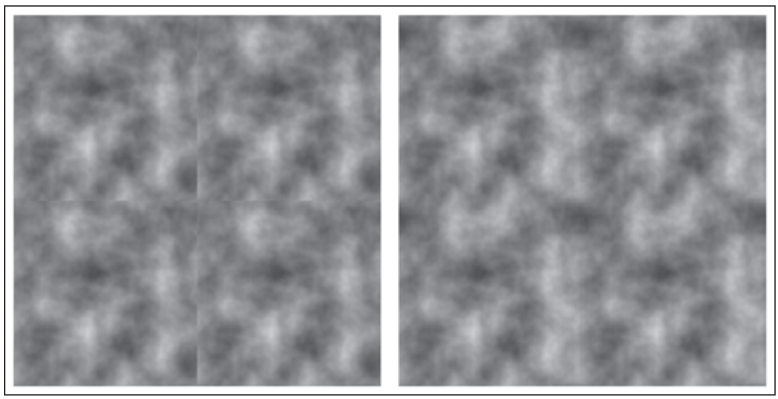

Рис. 8.3 ❖ Изображения, полученные повторением обычной (слева) и периодической (справа) текстуры 4-октавного шума Перлина

Подготовка

Основой этого рецепта будет служить программа из предыдущего рецепта «Создание текстуры шума с использованием GLM».

Как это делается...

Измените код предыдущего рецепта, как показано ниже.

В самом внутреннем цикле вместо функции `glm::perlin` нужно вызвать ее перегруженную версию, реализующую периодический генератор шума Перлина. Замените инструкцию

```
float val = glm::perlin(p) / scale;
```

следующим кодом:

```
float val = 0.0f;
if( periodic ) {
  val = glm::perlin(p, glm::vec2(freq)) / scale;
} else {
  val = glm::perlin(p) / scale;
}
```

Как это работает...

Второй параметр функции `glm::perlin` определяет период по осям X и Y. В качестве периода здесь используется значение `freq`, потому что шум воспроизводится в диапазоне частот от `0` до `freq` для каждой октавы.

См. также

- Рецепт «Создание текстуры шума с использованием GLM».

Создание эффекта облаков

Чтобы создать текстуру, напоминающую небо с облаками, можно использовать шум, значения которого будут играть роль коэффициентов смешивания цвета неба и облаков. Так как обычно облака являются большими образованиями, предпочтительнее использовать нижнюю октаву шума. Однако массивы облаков нередко осложнены более мелкими (высокочастотными) флуктуациями, поэтому добавление высшей октавы может оказаться неплохим дополнением.

На рис. 8.4 показан пример облачного неба, полученного с использованием приема, который описывается в этом рецепте.

Рис. 8.4 ❖ Пример облачного неба

Чтобы создать такой эффект, в этом рецепте будет вычисляться косинус значения шума и использоваться как коэффициент смешивания цвета неба и облаков.

Подготовка

Возьмите за основу программу, реализующую создание бесшовной текстуры, и реализуйте передачу этой текстуры в шейдеры через uniform-переменную `NoiseTex`.

Ниже перечислена пара uniform-переменных внутри фрагментного шейдера, которые должны инициализироваться в основной программе:

- `SkyColor`: цвет фона неба;
- `CloudColor`: цвет облаков.

Как это делается...

Чтобы создать шейдерную программу, использующую текстуру шума для получения эффекта облаков, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер, который будет передавать фрагментному шейдеру координаты текстуры через переменную `TexCoord`.
2. Добавить фрагментный шейдер:

```
#define PI 3.14159265

layout( binding=0 ) uniform sampler2D NoiseTex;
```

```
uniform vec4 SkyColor = vec4( 0.3, 0.3, 0.9, 1.0 );
uniform vec4 CloudColor = vec4( 1.0, 1.0, 1.0, 1.0 );

in vec2 TexCoord;

layout ( location = 0 ) out vec4 FragColor;

void main()
{
  vec4 noise = texture(NoiseTex, TexCoord);
  float t = (cos( noise.g * PI ) + 1.0) / 2.0;
  vec4 color = mix( SkyColor, CloudColor, t );
  FragColor = vec4( color.rgb , 1.0 );
}
```

Как это работает...

Фрагментный шейдер сначала извлекает значение шума из текстуры (переменная `noise`). Зеленый канал содержит шум двух октав, поэтому в рецепте используется этот канал (`noise.g`). Вы можете попробовать использовать другие каналы и посмотреть, не получится ли результат более привлекательным для вас.

Чтобы сделать переходы цвета более резкими, здесь используется функция вычисления косинуса. Значение шума принадлежит диапазону от `0` до `1`, а косинус этого значения будет принадлежать диапазону от `-1` до `1`, поэтому к значению косинуса прибавляется `1.0`, и сумма делится на `2.0`, что может привести окончательный результат к диапазону от `0` до `1`. Без применения функции вычисления косинуса облака на небе будут казаться размазанными по небу. Однако если это именно то, что вам нужно, уберите вычисление косинуса и используйте простое значение шума.

Далее выполняется смешивание цвета неба с цветом облаков в пропорции, определяемой значением `t`, и результат используется как окончательный цвет фрагмента.

И еще...

Если желательно получить эффект с небольшим количеством облаков, можно попробовать усекать значение `t` перед использованием его в качестве коэффициента смешивания. Например, попробуйте такой прием:

```
float t = (cos( noise.g * PI ) + 1.0 ) / 2.0;
t = clamp( t - 0.25, 0.0, 1.0 );
```

В результате получаемые коэффициенты окажутся смещены вниз (в сторону отрицательных значений), а функция `clamp` заменит все отрицательные значения нулем. Это увеличит площадь чистого неба и уменьшит интенсивность облачности.

См. также

- Дополнительную информацию о создании эффекта облаков можно найти в статье: **http://vterrain.org/Atmosphere/Clouds/**.
- Рецепт «Создание бесшовной текстуры шума».

Создание эффекта текстуры древесины

Чтобы сымитировать текстуру древесины, давайте сначала создадим виртуальное «полено» с идеальными концентрическими кольцами роста. Затем распилим полено и добавим искажения в изображение колец с помощью шума из текстуры.

На рис. 8.5 показано виртуальное «полено». Ось полена совпадает с осью Y. Кольца роста находятся на расстояниях от оси Y, выражаемых целыми числами. Они имеют более темный цвет, по сравнению с материалом между кольцами. Ширина каждого кольца выражается целым числом.

Чтобы получить «распил», достаточно просто определить двухмерную область в пространстве полена в виде координат текстуры. Первоначально координаты текстуры определяют квадратную область, координаты в которой изменяются от нуля до единицы. Предполагается, что область ориентирована параллельно плоскости X–Y, то есть текстурная координата S соответствует координате X, текстурная координата T – координате Y, а координата Z всегда равна нулю. После этого область можно преобразовывать как угодно, чтобы создать произвольный двухмерный распил.

После определения распила определяется цвет, исходя из расстояния от оси Y. Но перед этим расстояния изменяются с использованием значений из текстуры шума. Полученный результат должен напоминать текстуру настоящей древесины, как показано на рис. 8.6.

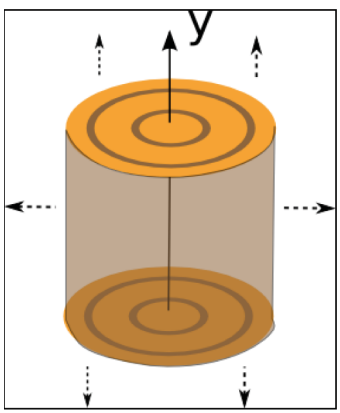

Рис. 8.5 ❖ Виртуальное полено с кольцами роста

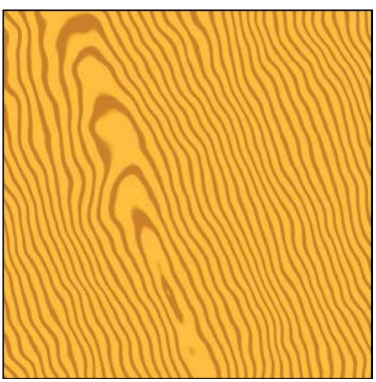

Рис. 8.6 ❖ Изображение текстуры древесины, полученное с помощью текстуры шума

Подготовка

Подготовьте программу для создания текстуры шума и реализуйте ее передачу в шейдеры через uniform-переменную `NoiseTex`.

Во фрагментном шейдере объявите три uniform-переменные, которые должны инициализироваться в основной программе:

- `LightWoodColor`: цвет светлых областей;
- `DarkWoodColor`: цвет темных областей (колец роста);
- `Slice`: матрица, определяющая распил виртуального «полена» и преобразующая область по умолчанию, определяемую координатами текстуры, в некоторую произвольную прямоугольную область.

Как это делается...

Чтобы создать шейдерную программу, генерирующую эффект древесины с применением текстуры шума, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер, передающий координаты текстуры фрагментному шейдеру через переменную `TexCoord`.
2. Добавить фрагментный шейдер:

```
layout(binding=0) uniform sampler2D NoiseTex;

uniform vec4 DarkWoodColor = vec4( 0.8, 0.5, 0.1, 1.0 );
uniform vec4 LightWoodColor = vec4( 1.0, 0.75, 0.25, 1.0 );
uniform mat4 Slice;

in vec2 TexCoord;

layout ( location = 0 ) out vec4 FragColor;

void main()
{
  // Преобразовать координаты текстуры, чтобы
  // получить "распил".
  vec4 cyl = Slice * vec4( TexCoord.st, 0.0, 1.0 );

  // Расстояние от оси Y полена.
  float dist = length(cyl.xz);

  // Изменить расстояние с помощью текстуры шума
  vec4 noise = texture(NoiseTex, TexCoord);
  dist += noise.b;

  // Определить цвет как смесь темного и
  // светлого цветов.
  float t = 1.0 - abs( fract( dist ) * 2.0 - 1.0 );
  t = smoothstep( 0.2, 0.5, t );
  vec4 color = mix( DarkWoodColor, LightWoodColor, t );

  FragColor = vec4( color.rgb , 1.0 );
}
```

Как это работает...

Первая строка в функции `main` фрагментного шейдера расширяет координаты текстуры до трехмерных (однородных) координат, где компонент `z` равен нулю `(s, t, 0, 1)`, а затем преобразует их с помощью матрицы `Slice`. Эта матрица может выполнять масштабирование, смещение и/или поворот координат, чтобы получить желаемую двухмерную плоскость «распила» виртуального «полена».

Чтобы проще было понять суть происходящего, представьте плоскость «распила» как двухмерный единичный квадрат, рассекающий «полено», нижний левый угол которого находится в начале координат. Матрица преобразует этот квадрат внутри полена, определяя направление «распила». Например, можно просто сместить квадрат (`-0.5, -0.5, -0.5`) и масштабировать с коэффициентом `20` по осям X и Y, чтобы получить распил через середину полена.

Далее с помощью встроенной функции `length` определяется расстояние от оси Y (`length(cyl.xz)`). Это расстояние помогает определить, как далеко находится текущий фрагмент от кольца роста. Если фрагмент находится между кольцами роста, для него выбирается светлый цвет древесины, а если близко к середине кольца – темный. Но прежде чем определить цвет, расстояние немного искажается с помощью значения из текстуры шума:

```
dist += noise.b;
```

Следующий шаг – небольшой трюк, определяющий цвет фрагмента, исходя из близости вещественного значения расстояния к целочисленному значению. Сначала отделяется дробная часть числа (`fract(dist)`), умножается на два, из результата вычитается единица, и затем отбрасывается знак, чтобы получить абсолютное значение. Так как дробная часть `fract(dist)` – это значение от нуля до единицы, умножение ее на два, вычитание единицы и отбрасывание знака дает в результате значение также от нуля до единицы. Но результат будет изменяться от `1.0` (для расстояния `dist`, равного `0.0`) до `0.0` (для расстояния `dist`, равного `0.5`) и снова до `1.0` (для расстояния `dist`, равного `1.0`), то есть график изменения значений имеет V-образную форму.

Затем V-образный график переворачивается вверх ногами, вычитанием значения из единицы, и результат сохраняется в `t`. Далее используется функция `smoothstep`, чтобы создать некоторое подобие резкого перехода между светлым и темным цветами. Иными словами, нам нужно, чтобы фрагмент получил темный цвет, когда преобразованное расстояние меньше `0.2`, светлый, когда расстояние больше `0.5`, и переходный цвет для промежуточных расстояний. Результат используется для смешивания светлого и темного цветов с помощью встроенной функции `mix`.

Функция `smoothstep(a, b, x)` действует следующим образом. Она вернет `0.0` для `x <= a`, `1.0` для `x >= b` и значение от `0` до `1`, интерполированное по формуле Эрмита, для `a < x < b`.

В результате получаются узкие полоски темного цвета по окружностям с целочисленными радиусами и более светлые широкие полоски между ними, с плавными, но короткими переходами цвета между ними.

В заключение полученный цвет просто присваивается фрагменту.

И еще...

Если две дощечки, вырезанные из одного и того же полена, склеить друг с другом, получится более широкая доска с симметричным рисунком. В компьютерной графике этот эффект можно сымитировать отражением координат текстуры.

Например, первую строку в предыдущей функции `main` можно было бы заменить следующим кодом:

```
vec2 tc = TexCoord;
if( tc.s > 0.5 ) tc.s = 1.0 - tc.s;
vec4 cyl = Slice * vec4( tc, 0.0, 1.0 );
```

Результат показан на рис. 8.7.

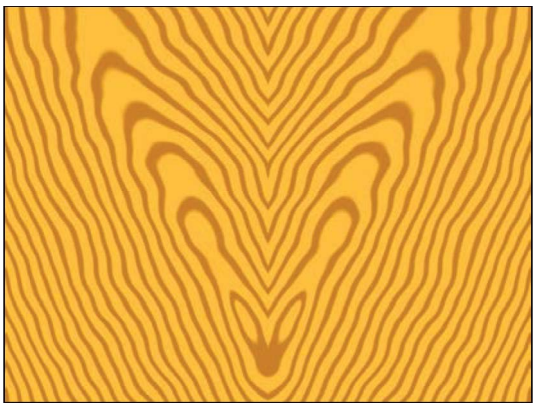

Рис. 8.7 ❖ Создание симметричного рисунка

См. также

- Рецепт «Создание текстуры шума с использованием GLM».

Создание эффекта разрушения

Если использовать текстуру шума в комбинации с ключевым словом `discard` в языке GLSL, легко можно получить имитацию разрушения, или рассыпания объектов. Для этого достаточно просто отбрасывать фрагменты, значения шума для которых окажутся выше некоторого порогового значения. На рис. 8.8 показан чайник, нарисованный с применением этого эффекта. Фрагменты отбрасываются, когда значение шума, соответствующее координатам текстуры, превышает заданный порог.

Подготовка

Подготовьте приложение OpenGL для передачи координат вершин, нормалей и координат текстуры в шейдеры. Обеспечьте передачу координат текстуры до фрагментного шейдера. Определите все uniform-переменные, необходимые для реализации выбранной модели затенения.

Создайте бесшовную текстуру шума (см. рецепт «Создание бесшовной текстуры шума») и сохраните ее в выбранном текстурном слоте.

Рис. 8.8 ❖ Эффект разрушения, или рассыпания объектов

Внутри фрагментного шейдера объявите следующие uniform-переменные и инициализируйте их в основной программе:

- `NoiseTex`: текстура шума;
- `LowThreshold`: нижний порог шума; фрагменты, для которых значение шума окажется ниже указанного порога, будут отбрасываться;
- `HighThreshold`: верхний порог шума; фрагменты, для которых значение шума окажется выше указанного порога, будут отбрасываться.

Как это делается...

Чтобы создать шейдерную программу, генерирующую эффект разрушения, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер, передающий координаты текстуры фрагментному шейдеру через переменную `TexCoord`, а также позицию вершины и нормаль через переменные `Position` и `Normal`.
2. Добавить фрагментный шейдер:

```
// Объявить здесь переменные, необходимые для
// реализации модели затенения по Фонгу

layout(binding=0) uniform sampler2D NoiseTex;

in vec4 Position;
in vec3 Normal;
in vec2 TexCoord;

uniform float LowThreshold;
uniform float HighThreshold;
```

```
layout ( location = 0 ) out vec4 FragColor;

vec3 phongModel() {
  // Вычислить цвет в соответствии с
  // моделью затенения по Фонгу
}

void main()
{
  // Получить значение шума для фрагмента с координатами TexCoord
  vec4 noise = texture( NoiseTex, TexCoord );

  // Если значение ниже нижнего или выше верхнего порога,
  // отбросить его
  if( noise.a < LowThreshold || noise.a > HighThreshold)
    discard;

  // Определить цвет фрагмента
  vec3 color = phongModel();
  FragColor = vec4( color , 1.0 );
}
```

Как это работает...

Сначала функция `main` фрагментного шейдера извлекает значение шума из текстуры (`NoiseTex`) и сохраняет его в переменной `noise`. Для имитации разрушения/рассыпания хорошо использовать шум с высокочастотными флуктуациями, поэтому выбирается шум, охватывающий четыре октавы и хранящийся в альфа-компоненте (`noise.a`).

Затем, если значение шума оказывается ниже нижнего (`LowThreshold`) или выше верхнего порога (`HighThreshold`), фрагмент отбрасывается с помощью ключевого слова `discard`, которое просто прерывает выполнение фрагментного шейдера, из-за чего последующие инструкции не выполняются.

Операция `discard` может оказывать отрицательное влияние на производительность из-за того, что некоторые реализации в таких ситуациях не выполняют раннюю проверку глубины.

В заключение вычисляется цвет фрагмента в соответствии с моделью затенения и присваивается выходной переменной.

См. также

- Рецепт «Создание бесшовной текстуры шума».

Создание эффекта брызг краски

Используя высокочастотный шум, легко воспроизвести эффект случайных брызг краски на поверхности объекта, как показано на рис. 8.9.

Текстура шума в данном случае используется для изменения цвета отдельных фрагментов объекта. Здесь в зависимости от значений шума будет выбираться цвет

Рис. 8.9 ❖ Эффект случайных брызг краски на поверхности объекта

отраженного рассеянного света. Если значение шума выше некоторого порога, будет использоваться цвет краски; в противном случае – основной цвет объекта.

Подготовка

Возьмите в качестве основы программу, реализующую отображение объекта с использованием модели затенения по Фонгу (или любой другой по своему выбору). Добавьте поддержку координат текстуры и реализуйте их передачу до фрагментного шейдера.

Ниже описывается пара uniform-переменных, определяющих параметры эффекта брызг краски:

- `PaintColor`: цвет капель краски;
- `Threshold`: минимальное значение шума, выше которого фрагмент будет получать цвет краски.

Создайте текстуру с высокочастотным шумом.

Организуйте передачу текстуры фрагментному шейдеру через переменную `NoiseTex`.

Как это делается...

Чтобы создать шейдерную программу, генерирующую эффект брызг краски, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер, который будет передавать фрагментному шейдеру координаты текстуры через переменную `TexCoord`, а также позицию вершины и нормаль через переменные `Position` и `Normal`.
2. Добавить фрагментный шейдер:

```
// Переменные для модели затенения по Фонгу
uniform struct LightInfo {
```

```
  vec4 Position;
  vec3 Intensity;
} Light;

uniform struct MaterialInfo {
  vec3 Ka;
  vec3 Kd;
  vec3 Ks;
  float Shininess;
} Material;

// Текстура шума
layout(binding=0) uniform sampler2D NoiseTex;

// Данные из вершинного шейдера
in vec4 Position;
in vec3 Normal;
in vec2 TexCoord;

// Парметры эффекта брызг краски
uniform vec3 PaintColor = vec3(1.0);
uniform float Threshold = 0.65;

layout ( location = 0 ) out vec4 FragColor;

vec3 phongModel(vec3 kd) {
  // Вычислить цвет в соответствии с
  // моделью затенения по Фонгу
}

void main()
{
  vec4 noise = texture( NoiseTex, TexCoord );
  vec3 color = Material.Kd;
  if( noise.g> Threshold ) color = PaintColor;
  FragColor = vec4( phongModel(color) , 1.0 );
}
```

Как это работает...

Функция `main` внутри фрагментного шейдера извлекает значение шума из `NoiseTex` и сохраняет его в переменной `noise`. Следующие две строки присваивают переменной `color` либо базовый цвет отраженного фонового цвета (`Material.Kd`), либо цвет краски `PaintColor`, в зависимости от результатов сравнения значения шума с пороговым значением (`Threshold`). Это обеспечит резкое изменение цвета фрагментов и получение эффекта брызг краски в соответствии с частотой шума.

В заключение вычисляется окончательный цвет фрагмента в соответствии с моделью затенения по Фонгу, с использованием `color` в качестве цвета отраженного фонового света. Результат вычислений применяется к фрагменту.

И еще...

Как отмечалось в рецепте «Создание текстуры шума с использованием GLM», использование низкочастотного шума приведет к увеличению размеров брызг

краски. Уменьшение порогового значения также приведет к увеличению размеров брызг, но, так как порог будет ниже, образуемый брызгами рисунок будет иметь менее случайный и более однородный вид.

См. также

- Рецепт «Создание бесшовной текстуры шума».

Создание эффекта изображения в приборе ночного видения

Шум можно использовать для имитации статических электрических разрядов или других похожих эффектов. В этом рецепте приводится забавный пример реализации одного из таких эффектов – эффект изображения в приборе ночного видения с дополнительным шумом, имитирующим случайные искажения сигнала. Исключительно ради забавы сцене будет придан типичный «бинокулярный» вид, как показано на рис. 8.10.

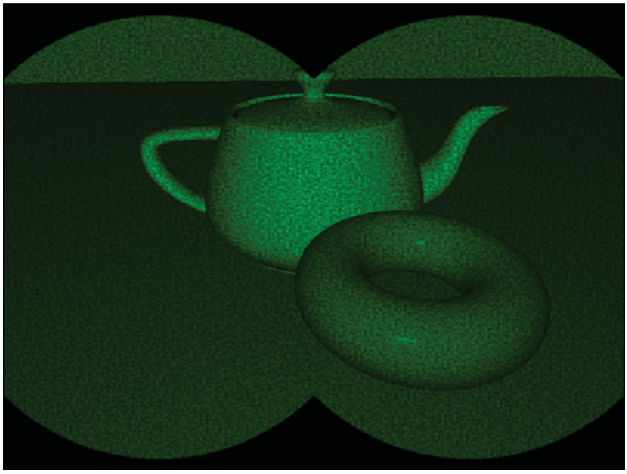

Рис. 8.10 ❖ Эффект изображения в приборе ночного видения с дополнительным шумом

Эффект изображения в приборе ночного видения будет воспроизводиться во втором проходе. Первый проход выполнит отображение сцены в текстуру (см. главу 4 «Текстуры»), а второй применит эффект изображения в приборе ночного видения.

Подготовка

Создайте объект буфера кадра (Framebuffer Object, FBO) для первого прохода. Подключите к нему текстуру в качестве буфера цвета. Как это делается, подробно рассказывается в главе 4 «Текстуры».

Объявите и инициализируйте uniform-переменные для реализации модели затенения. Объявите также следующие uniform-переменные во фрагментном шейдере:

- `Width`: ширина видимой области в пикселях;
- `Height`: высота видимой области в пикселях;
- `Radius`: радиус каждого круга, участвующего в создании «бинокулярного» эффекта (в пикселях);
- `RenderTex`: текстура, в которую выполняется отображение в первом проходе;
- `NoiseTex`: текстура шума;
- `RenderPass`: подпрограмма, используемая для выбора функции, в зависимости от прохода.

Создайте текстуру с высокочастотным шумом и реализуйте ее передачу во фрагментный шейдер через переменную `NoiseTex`. Свяжите текстуру `RenderTex` с объектом буфера кадра.

Как это делается...

Чтобы создать шейдерную программу, генерирующую эффект изображения в приборе ночного видения, нужно выполнить следующие шаги:

1. Добавить вершинный шейдер, который будет передавать фрагментному шейдеру позицию, нормаль и координаты текстуры через переменные `Position`, `Normal` и `TexCoord` соответственно.
2. Добавить фрагментный шейдер:

```
in vec3 Position;
in vec3 Normal;
in vec2 TexCoord;

uniform int Width;
uniform int Height;
uniform float Radius;
layout(binding=0) uniform sampler2D RenderTex;
layout(binding=1) uniform sampler2D NoiseTex;

subroutine vec4 RenderPassType();
subroutine uniform RenderPassType RenderPass;

// Объявить здесь переменные, необходимые для
// реализации модели затенения по Фонгу

layout( location = 0 ) out vec4 FragColor;

vec3 phongModel( vec3 pos, vec3 norm )
{
  // Вычислить цвет в соответствии с
  // моделью затенения по Фонгу
}

// Возвращает относительную светимость для
// заданного значения цвета
```

```
float luminance( vec3 color ) {
  return dot( color.rgb, vec3(0.2126, 0.7152, 0.0722) );
}

subroutine (RenderPassType)
vec4 pass1()
{
  return vec4(phongModel( Position, Normal ),1.0);
}

subroutine( RenderPassType )
vec4 pass2()
{
  vec4 noise = texture(NoiseTex, TexCoord);
  vec4 color = texture(RenderTex, TexCoord);
  float green = luminance( color.rgb );

  float dist1 = length(gl_FragCoord.xy -
      vec2(Width*0.25, Height*0.5));
  float dist2 = length(gl_FragCoord.xy -
    vec2(3.0*Width*0.25, Height*0.5));
  if( dist1 > Radius && dist2 > Radius ) green = 0.0;

  return vec4(0.0, green * clamp(noise.a + 0.25, 0.0, 1.0),
                                  0.0 ,1.0);
}

void main()
{
  // Будет вызывать pass1() или pass2()
  FragColor = RenderPass();
}
```

3. В функции отображения, в основной программе, реализовать следующие шаги:
   1) выполнить все настройки FBO, необходимые для отображения сцены в текстуру;
   2) выбрать функцию `pass1` для подпрограммы через переменную `RenderPass`;
   3) отобразить сцену;
   4) вернуться к использованию буфера кадра по умолчанию;
   5) выбрать функцию `pass2` для подпрограммы через переменную `RenderPass`;
   6) нарисовать прямоугольник, покрывающий всю видимую область, с использованием координат текстуры в диапазоне от 0 до 1 в каждом направлении.

Как это работает...

Фрагментный шейдер делится на две функции подпрограммы, по одной для каждого прохода. Внутри функции `pass1` просто вычисляется цвет фрагмента в соответствии с моделью затенения по Фонгу. Результаты записываются в объект буфера кадра с текстурой, которая будет использоваться во втором проходе.

Во втором проходе выполняется функция `pass2`. Она сначала извлекает значение шума (`noise`) и цвет фрагмента (`color`) из текстуры, куда выполнялось отображение в первом проходе. Затем вычисляет светимость для значения цвета и сохраняет результат в переменную `green`. Это значение будет использоваться в качестве зеленой составляющей окончательного цвета.

На следующем шаге определяется, попадает ли фрагмент в «бинокулярное» поле. Для этого вычисляется расстояние от центра левой линзы (`dist1`), который находится в середине видимого поля по вертикали и в одной четверти ширины от левого края по горизонтали. Центр правой линзы также находится в середине по вертикали, но в одной четверти ширины от правого края по горизонтали. Расстояние от центра правой линзы сохраняется в `dist2`. Если оба расстояния, `dist1` и `dist2`, больше радиуса виртуальных линз, переменной `green` присваивается нулевое значение.

В заключение функция возвращает окончательный цвет, в котором только зеленая составляющая может иметь ненулевое значение. Для добавления эффекта, имитирующего искажения в электрическом сигнале, переменная `green` умножается на значение шума. К значению шума прибавляется `0.25`, и результат преобразуется в диапазон от нуля до единицы вызовом функции `clamp`. Сделано это с целью немного повысить яркость изображения, так как во время экспериментов с программой я обнаружил, что изображение без такого преобразования получается слишком темным.

И еще...

Фрагментный шейдер можно было бы сделать более эффективным, если бы потребовалось сымитировать изменчивость шума при воспроизведении анимации. Для этого достаточно просто изменять значения координат в текстуре с шумом с течением времени. Пример такой реализации можно найти в статье, упоминаемой в разделе «См. также».

См. также

- Рецепт «Отображение в текстуру» в главе 4 «Текстуры».
- Рецепт «Создание текстуры шума с использованием GLM».
- Статью Войцеха Томана (Wojciech Toman): **wtomandev.blogspot.com/2009/09/night-vision-effect.html**, послужившую идеей для этого рецепта.

Глава 9

Системы частиц и анимация

В этой главе описываются следующие рецепты:

- анимация поверхности смещением вершин;
- создание фонтана частиц;
- создание системы частиц с использованием трансформации с обратной связью;
- создание системы частиц клонированием;
- имитация пламени с помощью частиц;
- имитация дыма с помощью частиц.

Введение

Шейдеры дают возможность использовать обширные возможности параллелизма, предлагаемые современными графическими процессорами. Так как они способны трансформировать координаты вершин, их можно использовать непосредственно для реализации анимационных эффектов. Можно получить огромный прирост эффективности, если удастся распараллелить алгоритм анимации и реализовать его в шейдерах.

Кроме того, чтобы задействовать шейдер в воспроизведении анимации, он должен не только вычислять координаты вершин, но и сохранять измененные координаты для использования при создании следующего кадра. Первоначально шейдеры не имели возможности записи в произвольные буферы (кроме, разумеется, буфера кадра). Однако такая возможность появилась в недавних версиях OpenGL в виде буферных объектов хранилищ и механизма загрузки/сохранения изображений. Начиная с версии OpenGL 3.0, появилась также возможность сохранять в произвольном буфере (или буферах) информацию, возвращаемую вершинным или геометрическим шейдером в выходных переменных. Этот механизм получил название **трансформация с обратной связью (Transform Feedback)** и особенно часто используется вместе с системами частиц.

В этой главе мы рассмотрим несколько примеров реализации анимации внутри шейдеров, уделив основное внимание системам частиц. Первый пример «Анимация поверхности смещением вершин» демонстрирует анимационный эффект, получаемый изменением координат вершин с течением времени. В рецепте «Создание фонтана частиц» демонстрируется создание простой системы частиц,

разлетающихся с постоянным ускорением. Рецепт «Создание системы частиц с использованием трансформации с обратной связью» иллюстрирует пример использования механизма трансформации с обратной связью совместно с системой частиц. В рецепте «Создание системы частиц клонированием» рассказывается, как анимировать множество сложных объектов, клонируя их.

Последние два рецепта демонстрируют применение системы частиц для имитации сложных природных явлений, таких как дым и огонь.

Анимация поверхности смещением вершин

Самый очевидный способ использования шейдеров в воспроизведении анимационных эффектов – организовать преобразование вершин внутри вершинного шейдера, опираясь на некоторую функцию времени. Основное приложение определяет статическую геометрию, а вершинный шейдер изменяет ее, используя значение текущего времени (передается в uniform-переменной). В результате вычисления, касающиеся координат вершин, перемещаются из основной программы в шейдер и используются возможности параллелизма, предоставляемые графическим драйвером.

В данном примере будет создан эффект волнения поверхности путем преобразования вершин прямоугольника по синусоидальному закону. Программа будет передавать вниз по конвейеру множество треугольников, составляющих прямоугольную поверхность, лежащую в плоскости X–Z, а вершинный шейдер будет преобразовывать координату Y каждой вершины, опираясь на функцию синуса с временной зависимостью, и вычислять вектор нормали для преобразованной вершины. На рис. 9.1 показан результат, который предполагается получить. (Волны движутся по поверхности слева направо.)

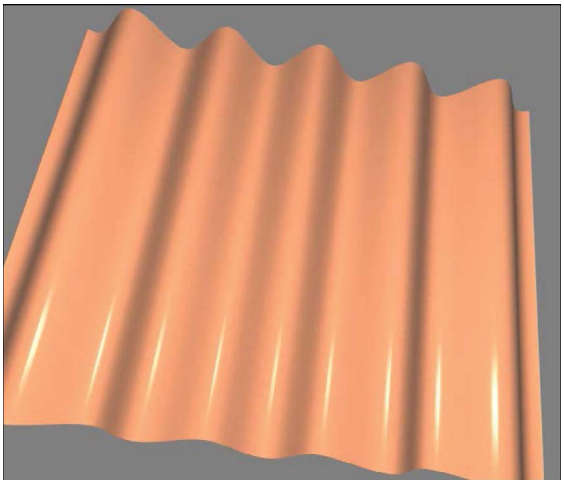

Рис. 9.1 ❖ Эффект волнения поверхности

Для создания эффекта случайных колебаний можно было бы использовать текстуру шума (см. главу 8 «Использование шума в шейдерах»).

Прежде чем перейти к программному коду, я предлагаю познакомиться с математическим аппаратом.

В данном рецепте предполагается преобразовывать координату Y вершин поверхности в зависимости от текущего времени и координаты X. Для этого будет использоваться уравнение плоской волны, изображенное на рис. 9.2, где A – амплитуда волны (высота вершин), λ – длина волны (расстояние между соседними вершинами) и v – скорость движения волны. На рис. 9.2 показан пример волны для `t = 0` и с длиной волны, равной единице. Эти коэффициенты будут передаваться через uniform-переменные из основной программы.

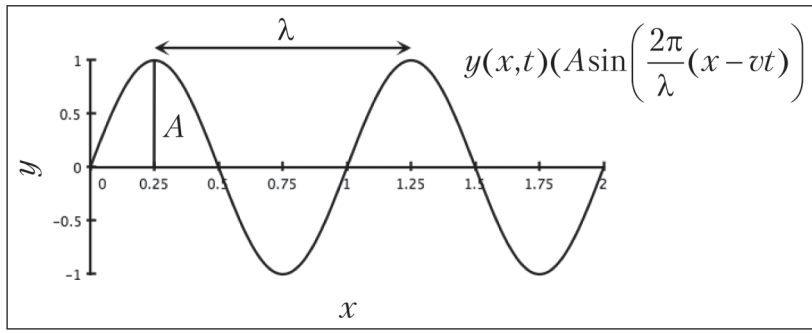

Рис. 9.2 ❖ Функция изменения координаты Y в зависимости от текущего времени и координаты X

Чтобы обеспечить правильное освещение/затенение поверхности, необходимо также вычислить вектор нормали для каждой преобразованной вершины. Сделать это можно, используя частную производную от предыдущей функции:

$$n(x.t) = \left(-A\frac{2\pi}{\lambda}\cos\left(\frac{2\pi}{\lambda}(x-vt)\right), 1\right).$$

Конечно, предыдущий вектор должен быть нормализован перед использованием в модели затенения.

Подготовка

Подготовьте приложение, отображающее плоскую поверхность, разбитую на множество треугольников. Чем большее число треугольников будет использовано, тем более гладкий эффект получится. Организуйте также слежение за временем анимации, используя любой метод по своему выбору. Обеспечьте передачу текущего времени в вершины шейдера через uniform-переменную `Time`.

Ниже перечислены другие важные uniform-переменные, используемые для передачи коэффициентов из предыдущего уравнения:

- K: волновое число ($2\pi/\lambda$);
- Velocity: скорость движения волны;
- Amp: амплитуда.

Добавьте также объявления uniform-переменных для передачи параметров модели затенения.

Как это делается...

Выполните следующие шаги:

1. Добавить вершинный шейдер:

```
layout (location = 0) in vec3 VertexPosition;

out vec4 Position;
out vec3 Normal;

uniform float Time;     // Время анимации

// Параметры волны
uniform float K;        // Волновое число
uniform float Velocity; // Скорость
uniform float Amp;      // Амплитуда

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;

void main()
{
  vec4 pos = vec4(VertexPosition,1.0);

  // Сместить координату Y
  float u = K * (pos.x - Velocity * Time);
  pos.y = Amp * sin( u );

  // Вычислить вектор нормали
  vec3 n = vec3(0.0);
  n.xy = normalize(vec2(-K * Amp *cos( u ), 1.0));

  // Передать координаты и нормаль (в видимых координатах)
  // фрагментному шейдеру
  Position = ModelViewMatrix * pos;
  Normal = NormalMatrix * n;

  // Позиция в усеченных координатах
  gl_Position = MVP * pos;
}
```

2. Добавить фрагментный шейдер, вычисляющий цвет фрагмента на основе значений переменных Position и Normal, используя модель затенения по вашему выбору (см. рецепт «Пофрагментное вычисление освещенности для повышения реализма» в главе 3 «Освещение, затенение и оптимизация»).

Как это работает...

Вершинный шейдер получает позицию вершины и изменяет координату Y, используя уравнение волны, обсуждавшееся выше. После выполнения трех первых инструкций в функции `main` вершинного шейдера переменная `pos` будет хранить копию входной переменной `VertexPosition` с измененной координатой Y.

Затем по формуле частной производной вычисляется вектор нормали и после нормализации сохраняется в переменной `n`. Так как в данном рецепте моделируется двухмерная волна (отсутствует зависимость от координаты Z), компонент `z` вектора нормали получает нулевое значение.

В заключение новая позиция и вектор нормали после преобразования в видимые координаты передаются фрагментному шейдеру. Как обычно, через встроенную переменную `gl_Position` возвращаются координаты в усеченной системе координат.

И еще...

Изменение координат вершин внутри вершинного шейдера – это самый простой, самый непосредственный способ переноса вычислений с центрального процессора CPU на графический процессор GPU. Кроме того, данный прием освобождает от необходимости передавать буферы с вершинами из основной памяти в память графической карты и обратно, чтобы изменить координаты вершин.

Главный недостаток данного приема – недоступность измененных координат для главной программы. Например, знание таких координат могло бы потребоваться для дополнительных вычислений (таких как определение точек столкновения). Однако есть ряд способов передачи данных в обратную сторону. Один из таких способов заключается в том, чтобы использовать объекты буферов кадров. В следующем рецепте будет представлен еще один прием, основанный на достаточно новой возможности, недавно появившейся в OpenGL: трансформации с обратной связью.

См. также

- Рецепт «Пофрагментное вычисление освещенности для повышения реализма» в главе 3 «Освещение, затенение и оптимизация».

Создание фонтана частиц

В компьютерной графике под системой частиц подразумевается группа объектов, используемых для имитации различных «нечетких» систем, таких как дым, брызги жидкостей, пламя, взрывы и другие подобные явления. Каждая частица в подобной системе рассматривается как отдельный точечный объект со своими координатами, но не имеющий размеров. Часто они отображаются как точечные спрайты (с применением режима `GL_POINTS`, как простые точечные примитивы), но могут отображаться как квадраты или треугольники. Каждая частица имеет свой жизненный цикл: она рождается, живет и умирает в соответствии с заданным на-

бором правил. Затем частица может возрождиться заново и повторить весь процесс. В общем случае частицы не взаимодействуют друг с другом и не отражают света. Часто частицы отображаются как простые текстурированные и прозрачные квадраты, ориентированные так, что лицевой стороной всегда смотрят в камеру.

Движение частиц на протяжении всего их жизненного цикла подчиняется установленным правилам. В число таких правил часто включаются базовые законы кинематики, определяющие траектории движения частиц под действием постоянных сил (таких как сила тяготения). Дополнительно могут также учитываться ветер, сила трения и другие факторы. В течение жизни частицы могут изменять форму или прозрачность. По достижении определенного возраста (или позиции) частица «умирает» и может быть «возрождена» и использована повторно.

В данном примере реализована относительно простая система частиц, которая выглядит как фонтан воды. Для простоты частицы здесь не будут «возрождаться». По достижении конца жизненного цикла частицы будут становиться прозрачными, то есть невидимыми. В результате получается фонтан с конечным временем работы, как если бы выбрасывался ограниченный объем воды. В последующих рецептах будет показано несколько способов усовершенствования этой системы за счет возрождения и повторного использования частиц.

На рис. 9.3 показана последовательность кадров, полученных при выводе данной системы частиц.

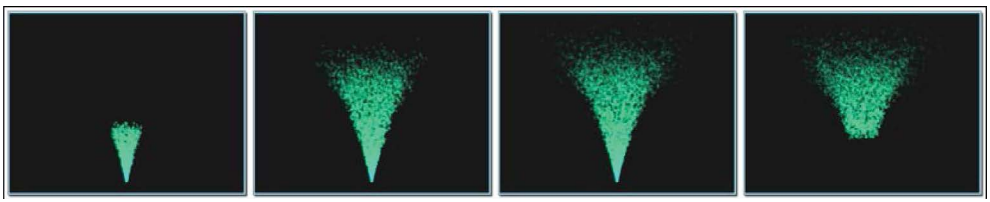

Рис. 9.3 ❖ Последовательность кадров, полученных при выводе простой системы частиц

Для анимации частиц в этом рецепте будет использоваться стандартный закон кинематики, определяющий траектории движения частиц под действием постоянных сил:

$$P(t) = P_0 + v_0 t + \frac{1}{2} a t^2.$$

Данное уравнение позволяет определять координаты в момент времени t. P_0 – это начальная позиция, v_0 – начальная скорость и a – ускорение.

Начальная позиция для всех частиц будет выбрана одна и та же: `(0,0,0)`. Начальная скорость будет выбираться случайно, в определенном диапазоне значений. Времена создания частиц будут немного отличаться, соответственно, время t в уравнении выше для каждой частицы будет измеряться отдельно относительно времени создания данной частицы.

Так как начальное местоположение всех частиц будет одним и тем же, его не требуется передавать шейдеру в отдельной входной переменной, соответственно, остается передать только два значения: начальную скорость и начальный момент времени (момент «рождения» частицы). До момента рождения частицы она отображается абсолютно прозрачной. На протяжении жизненного цикла позиция частицы определяется с помощью уравнения, представленного выше, со значением времени *t*, измеряемым относительно времени ее рождения (`Time - StartTime`).

Каждая частица будет отображаться как текстурированный точечный спрайт (в режиме `GL_POINTS`). Наложение текстуры на точечный спрайт выполняется очень просто благодаря тому, что OpenGL автоматически генерирует координаты текстуры и передает их фрагментному шейдеру через встроенную переменную `gl_PointCoord`. Кроме того, с течением времени прозрачность точечных спрайтов будет увеличиваться в линейной пропорции, чтобы в момент «смерти» частицы исчезали.

Подготовка

В основной программе нужно создать два буфера (или один буфер с перемежающимися значениями) с исходными данными для вершинного шейдера. В первом буфере будут храниться начальные скорости частиц. Они будут выбираться случайно из ограниченного диапазона возможных векторов. Чтобы получился вертикальный «конус» из частиц, как показано на рис. 9.3, все векторы, доступные для выбора, будут находиться в пределах этого конуса. Ниже показано, как это реализуется на практике:

```
vec3 v(0.0f);
float velocity, theta, phi;
GLfloat *data = new GLfloat[nParticles * 3];
for( unsigned int i = 0; i < nParticles; i++ ) {
  // Выбрать вектор, определяющий направление и скорость
  theta = glm::mix(0.0f, (float)PI / 6.0f, randFloat());
  phi = glm::mix(0.0f, (float)TWOPI, randFloat());

  v.x = sinf(theta) * cosf(phi);
  v.y = cosf(theta);
  v.z = sinf(theta) * sinf(phi);

  // Масштабировать величину скорости
  velocity = glm::mix(1.25f,1.5f,randFloat());
  v = v * velocity;

  data[3*i]   = v.x;
  data[3*i+1] = v.y;
  data[3*i+2] = v.z;
}

glBindBuffer(GL_ARRAY_BUFFER, initVel);
glBufferSubData(GL_ARRAY_BUFFER, 0,
                nParticles * 3 * sizeof(float), data);
```

Функция randFloat, используемая в этом фрагменте, возвращает случайное значение в диапазоне от нуля до единицы. Чтобы получить случайные значения в других диапазонах, используется функция mix из библиотеки GLM (она действует точно так же, как одноименная функция mix в языке GLSL, – выполняет линейную интерполяцию между значениями первых двух аргументов). Здесь мы получаем случайное вещественное число от нуля до единицы и интерполируем его в диапазон, заданный двумя граничными значениями.

Чтобы получить векторы в пределах конуса, используются сферические координаты. Значение theta определяет угол между осью и боковой поверхностью конуса. Значение phi определяет возможные направления вокруг оси Y для заданного значения theta. За более подробной информацией о сферических координатах я рекомендую обратиться к специализированной литературе по математике.

После получения направления вектор масштабируется значением скорости в диапазоне от 1.25 до 1.5. Этот диапазон обеспечивает получение желаемого визуального эффекта. Длина вектора направления определяет скорость частицы – вы можете попробовать изменить диапазон масштабирования, чтобы получить более широкий диапазон скоростей частиц.

Последние три инструкции в цикле присваивают компоненты вектора соответствующим элементам массива data. Вслед за циклом выполняется копирование данных в буфер по ссылке initVel, после чего буфер настраивается для передачи в вершинный шейдер через переменную-атрибут с индексом 0.

Во втором буфере будут храниться начальные времена «рождения» частиц – по одному вещественному значению для каждой вершины (частицы). В данном примере все частицы создаются последовательно, с фиксированной скоростью, как показано ниже, где каждая последующая частица рождается через 0.00075 секунды после предыдущей:

```
float * data = new GLfloat[nParticles];
float time = 0.0f, rate = 0.00075f;

for( unsigned int i = 0; i < nParticles; i++ ) {
  data[i] = time;
  time += rate;
}
glBindBuffer(GL_ARRAY_BUFFER,startTime);
glBufferSubData(GL_ARRAY_BUFFER, 0, nParticles * sizeof(float), data);
```

Этот фрагмент просто создает массив вещественных чисел, начиная с нуля, с шагом rate. Затем массив копируется в буфер по ссылке startTime и настраивается для передачи в вершинный шейдер через переменную-атрибут с индексом 1.

Определите внутри основной программы следующие uniform-переменные:

- ParticleTex: текстура для наложения на частицы;
- Time: время, прошедшее с момента начала анимации;
- Gravity: вектор, представляющий половину ускорения из предыдущего уравнения;
- ParticleLifetime: продолжительность существования частицы после рождения.

Выключите функцию проверки глубины и включите альфа-смешивание, как показано ниже:

```
glDisable(GL_DEPTH_TEST);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Также необходимо выбрать разумные размеры для каждого точечного спрайта. Например, следующая строка определяет размер, равный 10 пикселям:

```
glPointSize(10.0f);
```

Как это делается...

Добавьте вершинный шейдер:

```
// Начальные скорость и время
layout (location = 0) in vec3 VertexInitVel;
layout (location = 1) in float StartTime;

out float Transp; // Прозрачность частицы

uniform float Time; // Время анимации
uniform vec3 Gravity = vec3(0.0,-0.05,0.0); // мировые координаты
uniform float ParticleLifetime; // Максимальное время жизни частицы

uniform mat4 MVP;

void main()
{
  // Предполагается, что начальная позиция (0,0,0).
  vec3 pos = vec3(0.0);
  Transp = 0.0;

  // Частица невидима, пока не была рождена
  if( Time > StartTime ) {
    float t = Time - StartTime;
    if( t < ParticleLifetime ) {
      pos = VertexInitVel * t + Gravity * t * t;
      Transp = 1.0 - t / ParticleLifetime;
    }
  }

  gl_Position = MVP * vec4(pos, 1.0);
}
```

Добавьте фрагментный шейдер:

```
in float Transp;
uniform sampler2D ParticleTex;

layout ( location = 0 ) out vec4 FragColor;

void main()
{
  FragColor = texture(ParticleTex, gl_PointCoord);
  FragColor.a *= Transp;
}
```

Как это работает...

Вершинный шейдер получает начальную скорость частицы (VertexInitVel) и время рождения (StartTime). В переменной Time хранится время, прошедшее с начала воспроизведения анимации. Выходной переменной Transp присваивается общая прозрачность частицы.

Функция main внутри вершинного шейдера инициализирует начальную позицию частицы координатами модели (0,0,0) и прозрачность значением 0.0 (полная прозрачность). Затем с помощью инструкции if определяется, существует ли частица. Если текущее время больше времени рождения частицы, следовательно, частица существует, иначе считается, что частица пока не «родилась». В последнем случае позиция частицы остается прежней (в начале координат), и сохраняется ее полная прозрачность.

Если частица существует, для нее определяется «возраст» как разность текущего времени и времени рождения и сохраняется в переменной t. Если значение t больше или равно максимальному времени существования частиц (ParticleLifetime), значит, частица завершила свой жизненный путь и должна стать совершенно прозрачной. Иначе частица остается активной, и выполняется тело вложенной инструкции if, отвечающее за анимацию частицы.

Для существующей частицы определяется ее позиция (pos) по формуле, описанной выше. Прозрачность определяется по возрасту частицы путем линейной интерполяции.

```
Transp = 1.0 - t / ParticleLifetime;
```

В момент рождения частица полностью непрозрачна, и с течением времени ее прозрачность увеличивается прямо пропорционально возрасту. В момент рождения Transp получает значение 1.0 и в конце существования частицы – значение 0.0.

Внутри фрагментного шейдера цвет фрагмента определяется по результатам поиска в текстуре. Так как отображаются примитивы GL_POINT, координаты текстуры определяются автоматически, самой системой OpenGL, и передаются шейдеру через встроенную переменную gl_PointCoord. Перед завершением альфа-компонент полученного цвета умножается на значение переменной Transp, чтобы частица приобрела степень прозрачности в зависимости от своего возраста (который определяется в вершинном шейдере).

И еще...

Этот пример задумывался как упрощенное введение в системы частиц. Существует множество способов расширения возможностей таких систем. Например, можно было бы реализовать изменение размеров частиц или поворачивать их в процессе анимации, чтобы произвести различные эффекты.

Можно было бы усилить ощущение глубины, изменяя размеры частиц в зависимости от расстояния до камеры. Реализовать такое изменение размеров можно с помощью встроенной переменной gl_PointSize, доступной внутри вершинного шейдера. Также вместо точек можно рисовать квадраты или треугольники.

Точечные спрайты (GL_POINTS) – очевидный выбор для отображения частиц, но они имеют ряд недостатков. Во-первых, аппаратное обеспечение часто накладывает существенные ограничения на размеры точек. Если потребуется отобразить достаточно крупные частицы или изменять их размеры с течением времени, можно столкнуться с серьезными ограничениями. Во-вторых, точки обычно отсекаются, только когда их центры выходят за границы видимого объема. Это обстоятельство может стать причиной появления артефактов, разрушающих целостность сцены, если частицы имеют значительные размеры. Типичное решение данной проблемы заключается в замене точечных спрайтов квадратами или треугольниками с использованием приема, описанного в главе 6 «Использование геометрических шейдеров и шейдеров тесселяции».

Один из важнейших недостатков приема, описанного в данном рецепте, заключается в отсутствии простой возможности повторно использовать частицы, завершившие свой жизненный цикл. Когда частицы прекращают существовать, они просто делаются прозрачными. Было бы неплохо иметь возможность повторно использовать их для создания кажущегося непрерывным потока частиц. Кроме того, пригодилась бы также возможность изменять траекторию движения частиц при изменении силы тяжести или под влиянием других факторов (таких как ветер или перемещение источника частиц). Система, описанная здесь, лишена таких возможностей, потому что расчет траектории в ней производится только с применением единственного уравнения, а для реализации описанного выше необходимо дополнительно изменять позиции частиц, исходя из текущей ситуации (суммы сил, действующих в системе).

Чтобы добиться цели, обозначенной выше, необходим некоторый способ передачи вывода вершинного шейдера (обновленные позиции частиц) обратно на вход вершинного шейдера перед отображением следующего кадра. Реализовать это было бы легко, если бы имитация выполнялась не внутри шейдеров, – мы могли бы просто изменять координаты примитивов непосредственно перед отображением. Однако, так как вся работа выполняется внутри фрагментного шейдера, мы ограничены в средствах хранения промежуточных данных.

В следующем рецепте будет представлен пример использования нового механизма OpenGL, который называется **трансформация с обратной связью (transform feedback)**, обеспечивающего именно такую возможность. С его помощью можно определять, какие выходные переменные должны сохраняться в буферах и передаваться на вход последующих циклов отображения.

См. также

- Рецепт «Анимация поверхности смещением вершин».
- Рецепт «Создание системы частиц с использованием трансформации с обратной связью».

Создание системы частиц с использованием трансформации с обратной связью

Механизм **трансформации с обратной связью (Transform Feedback)** позволяет сохранить вывод вершинного (или геометрического) шейдера в буфер и использовать

его в следующих циклах отображения. Этот механизм, впервые появившийся в версии OpenGL 3.0, особенно хорошо подходит для анимации систем частиц, потому что, помимо всего прочего, обеспечивает возможность создавать пошаговые имитации. С его помощью можно изменять позиции частиц в вершинном шейдере и отобразить их с учетом изменений в следующем (или в этом же) цикле. Кроме того, измененные координаты можно использовать при создании следующего кадра анимации.

В данном примере будет реализована та же система частиц, что и в предыдущем рецепте («Создание фонтана частиц»), но на этот раз с применением механизма трансформации с обратной связью. Вместо использования уравнения, описывающего траекторию частиц без учета изменяющихся условий, здесь позиции частиц будут рассчитываться, исходя из их предыдущих местоположений, с применением уравнений, описывающих воздействие различных сил.

Часто в подобных случаях используется прием под названием **метод Эйлера**, вычисления местоположения и скорости частиц в момент времени `t`, исходя из местоположения, скорости и ускорения, имевших место ранее.

$$P_{n+1} = P_n + v_n h;$$
$$v_{n+1} = v_n + a_n h.$$

Нижние индексы в данной системе уравнений представляют шаг во времени (или порядковый номер кадра, если хотите), P – позиция частицы, а v – скорость ее движения. Уравнения описывают позицию и скорость частицы в кадре $n + 1$ как функцию от позиции и скорости в предыдущем кадре (n). Переменная h представляет временной шаг, или величину интервала времени между кадрами. Член a_n представляет мгновенное ускорение, вычисленное на основе позиции частицы. В данном примере это будет постоянная величина, но вообще ее можно изменять в зависимости от состояния окружения (ветер, столкновения с препятствиями, взаимодействия между частицами и т. д.).

Метод Эйлера фактически является численным методом решения дифференциального уравнения Ньютона, описывающего движение. Это один из простейших методов. Однако он является одношаговым методом, а потому может иметь значительную погрешность. В числе более точных методов можно назвать **интегрирование Верле** и **метод Рунге-Кутта**. Так как в нашем случае визуальный эффект имеет большее значение, чем физическая точность, метода Эйлера будет вполне достаточно.

Чтобы обеспечить работу имитации фонтана частиц в данном примере, будет использоваться прием, который иногда называют «пинг-понг с буферами» (попеременное переключение буферов). Программа будет поддерживать два комплекта буферов вершин и менять их в каждом кадре. Например, пусть буфер A используется для передачи позиций и скоростей на вход вершинного шейдера. Вершинный шейдер изменяет позиции со скоростями, используя метод Эйлера, и отправляет результаты в буфер B посредством механизма трансформации с обратной связью. Затем во втором проходе в качестве источника информации о вершинах используется уже буфер B, как показано на рис. 9.4.

При подготовке следующего кадра процедура повторяется, но буферы при этом меняются местами.

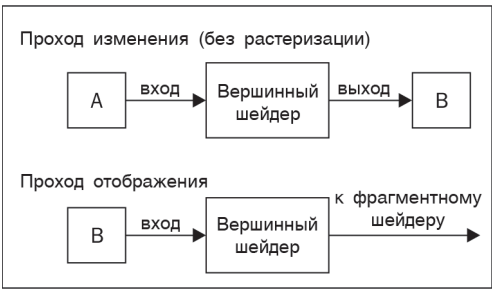

Рис. 9.4 ❖ Прием попеременного переключения буферов

В общем случае механизм трансформации с обратной связью позволяет определить множество выходных переменных шейдера для сохранения в заданном буфере (или во множестве буферов). Вся процедура выполняется в несколько этапов, которые будут продемонстрированы ниже, но основная идея состоит в следующем. Непосредственно перед связыванием шейдерной программы определяются отношения между буферами и выходными переменными шейдера с помощью функции `glTransformFeedbackVaryings`. Во время отображения инициируется проход трансформации с обратной связью. Буферы связываются с соответствующими индексами в точке привязки. (При необходимости можно запретить растеризацию, чтобы частицы не отображались.) Затем включается трансформация с обратной связью вызовом `glBeginTransformFeedback`, после чего осуществляется рисование точечных примитивов. Выход вершинного шейдера сохраняется в соответствующих буферах. Затем трансформация с обратной связью выключается вызовом `glEndTransformFeedback`.

Подготовка

Создайте три пары буферов. Первая пара будет хранить координаты частиц, вторая – скорости их движения, третья – «начальное время» для каждой частицы (время рождения частицы). Для простоты будем называть первый буфер в каждой паре буфер *A*, а второй – буфер *B*. Также нужно создать еще один буфер для хранения начальных скоростей частиц.

Создайте два массива вершин. Первый массив следует связать с буфером координат *A* (атрибут с индексом `0`), буфер скоростей *A* – с атрибутом, имеющим индекс `1`, буфер начальных времен *A* – с атрибутом, имеющим индекс `2`, и буфер начальных скоростей – с атрибутом, имеющим индекс `3`. Второй массив вершин должен быть настроен аналогично с использованием буферов *B* и с тем же буфером начальных скоростей. В следующем фрагменте программного кода дескрипторы двух массивов вершин доступны через массив типа `GLuint` с именем `particleArray`.

Инициализируйте буферы *A* соответствующими начальными значениями. Например, все позиции можно установить в начало координат, а скорости и начальные времена инициализировать так же, как это было сделано в предыдущем ре-

цепте «Создание фонтана частиц». Содержимое для буфера начальных скоростей можно просто скопировать из буфера скоростей.

При использовании механизма трансформации с обратной связью нужно определить буферы, которые будут принимать выходные данные вершинного шейдера, путем связывания буферов с индексами в точке привязки `GL_TRANSFORM_FEEDBACK_BUFFER`. Индекс в точке привязки соответствует индексу выходной переменной вершинного шейдера, как указано в вызове `glTransformFeedbackVaryings`.

Для простоты в данном примере будут использоваться объекты обратной связи. Подготовьте два объекта обратной связи, по одному для каждого комплекта буферов:

```
GLuint feedback[2];  // Объекты обратной связи
GLuint posBuf[2];    // Буферы с координатами (A и B)
GLuint velBuf[2];    // Буферы скоростей (A и B)
GLuint startTime[2]; // Буферы начальных времен (A и B)

// Создать буферы A и B для posBuf, velBuf и startTime
// Подготовить объекты обратной связи
glGenTransformFeedbacks(2, feedback);

// Обратная связь 0
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[0]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,0,posBuf[0]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,1,velBuf[0]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,2,startTime[0]);

// Обратная связь 1
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[1]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,0,posBuf[1]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,1,velBuf[1]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,2,startTime[1]);
```

Вместе с массивами координат объекты обратной связи хранят привязки буферов к точке `GL_TRANSFORM_FEEDBACK_BUFFER`, благодаря чему их можно быстро задействовать позднее. В предыдущем фрагменте создаются два объекта обратной связи, и их дескрипторы сохраняются в массиве `feedback`. Элемент `posBuf[0]` связывается с индексом `0` точки привязки в первом объекте (комплект буферов *A*), `velBuf[0]` – с индексом `1` и `startTime[0]` с индексом `2`. Эти привязки уже соединены с выходными переменными шейдера вызовом `glTransformFeedbackVaryings` (или посредством квалификатора `layout`, см. раздел «И еще...» ниже). Последним аргументом в каждом случае передается дескриптор буфера. Аналогичным образом выполняется связывание второго комплекта буферов (буферы *B*) с точкой привязки во втором объекте.

Далее можно будет определять множество буферов для приема выходных данных шейдера путем связывания того или другого объекта.

Ниже перечислены uniform-переменные, которые следует объявить и инициализировать:

- `ParticleTex`: текстура для наложения на точечные спрайты;

- `Time`: текущее время;
- `H`: время, прошедшее между двумя кадрами;
- `Accel`: используется для определения ускорения;
- `ParticleLifetime`: определяет предельное время жизни частицы, прежде чем она будет использована повторно.

Как это делается...

Выполните следующие шаги:

1. Добавьте вершинный шейдер:

```
subroutine void RenderPassType();
subroutine uniform RenderPassTypeRenderPass;

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexVelocity;
layout (location = 2) in float VertexStartTime;
layout (location = 3) in vec3 VertexInitialVelocity;

out vec3 Position;   // Для обратной связи
out vec3 Velocity;   // Для обратной связи
out float StartTime; // Для обратной связи
out float Transp;    // Для фрагментного шейдера

uniform float Time;  // Время анимации
uniform float H;     // Время, прошедшее между двумя кадрами
uniform vec3 Accel;  // Ускорение
uniform float ParticleLifetime; // Время жизни частиц

uniform mat4 MVP;

subroutine (RenderPassType)
void update() {
  Position = VertexPosition;
  Velocity = VertexVelocity;
  StartTime = VertexStartTime;

  if( Time >= StartTime ) {
    float age = Time - StartTime;
    if( age > ParticleLifetime ) {
      // Срок жизни частицы истек, использовать повторно.
      Position = vec3(0.0);
      Velocity = VertexInitialVelocity;
      StartTime = Time;
    } else {
      // Частица существует, обновить.
      Position += Velocity * H;
      Velocity += Accel * H;
    }
  }
}

subroutine (RenderPassType)
void render() {
```

```
  float age = Time - VertexStartTime;
  Transp = 1.0 - age / ParticleLifetime;
  gl_Position = MVP * vec4(VertexPosition, 1.0);
}

void main()
{
  // Будет вызывать render() или update()
  RenderPass();
}
```

2. Добавьте фрагментный шейдер:

```
uniform sampler2D ParticleTex;
in float Transp;
layout ( location = 0 ) out vec4 FragColor;

void main()
{
  FragColor = texture(ParticleTex, gl_PointCoord);
  FragColor.a *= Transp;
}
```

3. После компиляции шейдерной программы, но перед компоновкой используйте следующий код, чтобы связать выходные переменные вершинного шейдера с выходными буферами:

```
const char * outputNames[] = { "Position", "Velocity",
                               "StartTime" };
glTransformFeedbackVaryings(progHandle, 3, outputNames,
                            GL_SEPARATE_ATTRIBS);
```

4. В функции отображения, в основной программе, реализуйте передачу позиций частиц в вершинный шейдер для обновления и сохраните результаты, используя объекты обратной связи. Входные данные для вершинного шейдера должны браться из буфера *A*, а выходные данные – сохраняться в буфере *B*. Для этого прохода нужно включить `GL_RASTERIZER_DISCARD`, чтобы запретить отображение в буфер кадра:

```
// Выбрать функцию подпрограммы для изменения координат
glUniformSubroutinesuiv(GL_VERTEX_SHADER, 1, &updateSub);

// Установить значения uniform-переменных: H и Time
...

// Запретить отображение
glEnable(GL_RASTERIZER_DISCARD);

// Определить, какой объект используется для обратной связи,
// будет использоваться для рисования
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK,
  feedback[drawBuf]);

// "Нарисовать" точки из входного буфера в
```

```
// буферы с обратной связью
glBeginTransformFeedback(GL_POINTS);
glBindVertexArray(particleArray[1-drawBuf]);
glDrawArrays(GL_POINTS, 0, nParticles);
glEndTransformFeedback();
```

5. Отобразить частицы после изменения их координат, используя буфер *B* в качестве источника входных данных для вершинного шейдера:

```
// Разрешить отображение
glDisable(GL_RASTERIZER_DISCARD);

glUniformSubroutinesuiv(GL_VERTEX_SHADER, 1, &renderSub);
glClear( GL_COLOR_BUFFER_BIT );

// Инициализировать uniform-переменные с
// матрицами преобразований, если необходимо
...

// Нарисовать спрайты из буфера обратной связи
glBindVertexArray(particleArray[drawBuf]);
glDrawTransformFeedback(GL_POINTS, feedback[drawBuf]);
```

6. Поменять назначение буферов:

```
// Поменять буферы
drawBuf = 1 - drawBuf;
```

Как это работает...

Данный пример содержит много нового, но не будем забегать вперед и начнем по порядку, с вершинного шейдера.

Вершинный шейдер разбит на две функции подпрограммы. Функция `update` вызывается в первом проходе и реализует метод Эйлера для изменения координат и скорости частицы. Функция `render` вызывается во втором проходе. Она вычисляет прозрачность частицы, исходя из ее возраста, и передает координаты и прозрачность фрагментному шейдеру.

Вершинный шейдер имеет четыре выходные переменные. Первые три – `Position`, `Velocity` и `StartTime` – используются в первом проходе для записи в буферы обратной связи. Четвертая (`Transp`) используется во втором проходе и служит для передачи значения прозрачности фрагментному шейдеру.

Функция `update` просто обновляет координаты и скорость частицы методом Эйлера, если она уже родилась и еще не прекратила своего существования. Если возраст частицы оказался больше предельно допустимого срока, она утилизируется и передается в повторное использование путем сброса координат в начальное местоположение, времени рождения – в текущее время (`Time`) и скорости – в начальное значение (из входной переменной-атрибута `VertexInitialVelocity`).

Функция `render` вычисляет возраст частицы и, исходя из него, определяет прозрачность, присваивая результат переменной `Transp`. Преобразует позицию частицы в усеченные координаты и сохраняет результат во встроенной выходной переменной `gl_Position`.

Фрагментный шейдер (шаг 2) вызывается только во втором проходе. Он определяет цвет фрагмента по текстуре ParticleTex и применяет значение прозрачности (Transp), полученное от вершинного шейдера.

Код, представленный в шаге 3, который выполняется перед компоновкой шейдерной программы, отвечает за настройку соответствий между выходными переменными шейдера и буферами обратной связи (буферами, связанными с индексами в точке привязки GL_TRANSFORM_FEEDBACK_BUFFER). Функция glTransformFeedbackVaryings принимает три аргумента. Первый – дескриптор объекта шейдерной программы. Второй – число имен выходных переменных, которые будут переданы. Третий – массив имен выходных переменных. Порядок следования имен в массиве должен соответствовать порядку следования индексов в буферах обратной связи. В данном случае переменная Position соответствует индексу 0, переменная Velocity – индексу 1 и переменная StartTime – индексу 2. Загляните чуть назад, в код, где создаются объекты обратной связи (вызов glBindBufferBase), чтобы убедиться в соответствии индексов.

Функцию glTransformFeedbackVaryings можно также использовать для передачи единственного буфера с перемежающимися данными (вместо комплекта отдельных буферов для каждой переменной). Подробности смотрите в документации с описанием OpenGL.

Шаги с 4 по 6 демонстрируют, как можно реализовать функцию отображения в основной программе. В данном примере создаются два важных массива типа GLuint: feedback и particleArray. Каждый из них состоит из двух элементов, хранящих два дескриптора объектов обратной связи и два массива вершин соответственно. Переменная drawBuf хранит простое целое число, используемое для переключения между двумя комплектами буферов. В момент создания любого кадра переменная drawBuf будет хранить либо нуль, либо единицу.

Код в шаге 4 сначала выбирает функцию update подпрограммы, чтобы активировать функциональность обновления в вершинном шейдере, и затем готовит uniform-переменные Time и H. Следующий вызов glEnable(GL_RASTERIZER_DISCARD) отключает растеризацию, чтобы предотвратить вывод результатов, полученных в этом проходе. Вызов glBindTransformFeedback выбирает комплект буферов, соответствующих индексу в переменной drawBuf, куда должен осуществляться вывод для обратной связи.

Прежде чем нарисовать точки (и тем самым обеспечить вызов вершинного шейдера), вызывается функция glBeginTransformFeedback, разрешающая трансформацию с обратной связью. В качестве аргумента ей передается тип примитивов, которые будут передаваться вниз по конвейеру (в данном случае GL_POINTS). Вывод вершинного (или геометрического) шейдера будет сохраняться в буферы, связанные с точкой привязки GL_TRANSFORM_FEEDBACK_BUFFER, пока не будет вызвана функция glEndTransformFeedback. Далее выбирается массив вершин с индексом 1 - drawBuf (если drawBuf хранит значение 0, будет использоваться массив с индексом 1, и наоборот) и выполняется рисование частиц.

В конце первого прохода (шаг 5) снова разрешается растеризация вызовом glEnable(GL_RASTERIZER_DISCARD), и выполняется переход к этапу отображения.

Проход отображения сам по себе достаточно прост; он начинается с выбора функции render подпрограммы, вслед за которым осуществляется рисование частиц из массива, соответствующего значению drawBuf. Однако вместо glDrawArrays используется функция glDrawTransformFeedback, потому что она специально предназначена для применения с механизмом преобразования с обратной связью. Объект обратной связи хранит число записанных вершин. Функция glDrawTransformFeedback получает этот объект в третьем аргументе и использует число вершин из него для рисования. По сути, эта функция действует подобно функции glDrawArrays, которой во втором аргументе передается число 0, а счетчик извлекается из объекта обратной связи.

В заключение в конце прохода отображения (шаг 6) производится перестановка буферов присваиванием переменной drawBuf значения 1 - drawBuf.

И еще...

Возможно, кто-то из вас уже задался вопросом: зачем нужно было организовывать вычисления в два прохода? Разве нельзя сохранить фрагментный шейдер активным и производить изменения и отображение в одном проходе? Да, в данном примере это возможно, более того, реализация с одним проходом будет даже эффективнее. Однако я решил продемонстрировать версию с двумя проходами, так как она является более универсальной. Обычно частицы образуют лишь часть сцены, и логика обработки частиц может быть никак не связана с другой, большей ее частью. Поэтому на практике чаще имеет смысл выделить изменение частиц в отдельный проход, выполняемый перед проходом отображения, чтобы разорвать связь между логикой изменения и логикой отображения.

Использование квалификаторов layout

В версии OpenGL 4.4 появились квалификаторы инструкции layout, позволяющие определять отношения между выходными переменными шейдера и буферами обратной связи непосредственно в шейдере, то есть без использования функции glTransformFeedbackVaryings. Любые выходные переменные, используемые для организации обратной связи, можно описать с помощью квалификаторов xfb_buffer, xfb_stride и xfb_offset.

Запрос результатов трансформации с обратной связью

Часто бывает полезно иметь возможность узнать количество примитивов, записанных в проходе трансформации с обратной связью. Например, при наличии геометрического шейдера число записанных примитивов может отличаться от числа примитивов, переданных по конвейеру.

Запросить эту информацию можно с помощью объекта запроса. Для этого нужно сначала создать объект запроса:

```
GLuint query;
glGenQueries(1, &query);
```

Затем перед началом прохода трансформации с обратной связью запустить процесс подсчета:

```
glBeginQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, query);
```

И по завершении прохода вызвать glEndQuery, чтобы остановить подсчет:

```
glEndQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN);
```

Далее можно будет получить число примитивов, как показано ниже:

```
GLuintprimWritten;
glGetQueryObjectuiv(query, GL_QUERY_RESULT, &primWritten);
printf("Primitives written: %d\n", primWritten);
```

Повторное использование частиц

В данном примере повторное использование частиц реализовано путем установки их координат и скорости в первоначальные значения. Такой подход может вызвать эффект «слипания» частиц с течением времени. Более удачное решение состояло бы в генерировании нового, случайного значения скорости и, возможно, координат (в зависимости от желаемых результатов). К сожалению, в настоящее время генератор случайных чисел не поддерживается в шейдерных программах. Решить проблему можно было бы реализацией собственного генератора случайных чисел, например с помощью текстуры со случайными значениями или текстуры шума (см. главу 8 «Использование шума в шейдерах»).

См. также

- Рецепт «Создание фонтана частиц».

Создание системы частиц клонированием

Чтобы нарисовать множество частиц, имеющих более сложную геометрию, можно воспользоваться поддержкой **отображения клонированием (instanced rendering)**. Отображение клонированием – это удобный и эффективный прием рисования нескольких копий одного и того же объекта. В OpenGL отображение клонированием поддерживается с помощью функций glDrawArraysInstanced и glDrawElementsInstanced.

В данном рецепте за основу взята система частиц из предыдущих рецептов, но вместо точечных спрайтов будут отображаться более сложные объекты. На рис. 9.5 показан пример отображения системы частиц, где каждая из них представлена тором.

Суть приема отображения клонированием заключается в простом вызове одной из указанных выше функций и передаче ей числа копий (экземпляров). Однако имеются некоторые тонкости в том, как должны передаваться атрибуты вершин в шейдер. Если все частицы нарисовать с одними и теми же атрибутами, получится скучный и невыразительный результат, потому что все частицы окажутся в одной позиции и будут иметь одинаковую ориентацию. Так как хотелось бы, чтобы

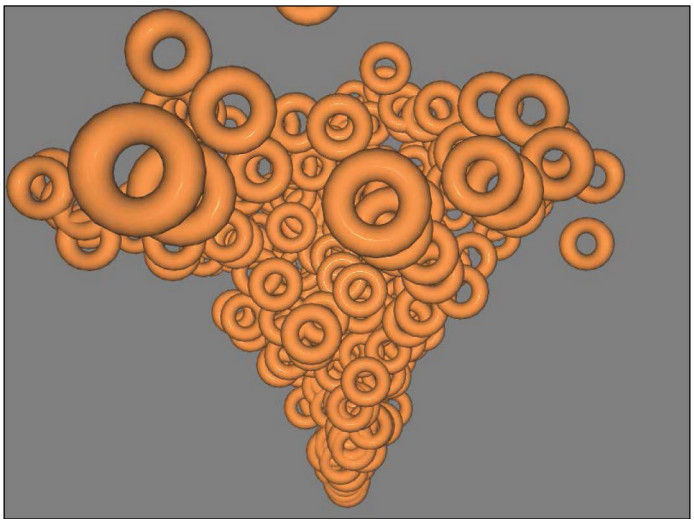

Рис. 9.5 ❖ Пример системы частиц, где каждая частица представлена тором

каждая копия была нарисована в собственной позиции, нужно каким-то способом передать в вершинный шейдер всю необходимую для этого информацию (в данном случае время рождения) отдельно для каждой частицы.

Ключом к решению задачи является функция `glVertexAttribDivisor`. Она определяет шаг перемещения по массиву значений атрибута в ходе отображения копий. Например, в следующем вызове:

```
glVertexAttribDivisor(1, 1);
```

первый аргумент определяет индекс атрибута, а второй – число экземпляров, которые будут переданы между изменениями атрибута. Иными словами, предыдущая инструкция определяет, что все вершины первого экземпляра получат первое значение из буфера, соответствующего атрибуту с индексом `1`. Все вершины второго экземпляра получат второе значение и т. д. Если во втором аргументе передать значение `2`, тогда первое значение из буфера получат первые два экземпляра, второе значение – следующие два экземпляра и т. д.

По умолчанию для каждого атрибута используется делитель `0`, то есть атрибуты вершин обрабатываются как обычно (каждое значение атрибута соответствует одной вершине, а не некоторому числу экземпляров). Если делитель не равен нулю, соответствующий атрибут называют **атрибутом экземпляра (instanced attribute)**.

Подготовка

Возьмите за основу систему частиц из рецепта «Создание фонтана частиц». Вам потребуется внести в нее всего несколько изменений. Обратите внимание, что с равным успехом можно было бы использовать реализацию с поддержкой транс-

формации с обратной связью, но, чтобы не усложнять пример, я предлагаю использовать более простую систему частиц. Вы легко сможете адаптировать данный пример для включения обратной связи, если потребуется.

Определяя объект с массивом вершин для системы частиц, добавьте два новых атрибута экземпляров для передачи начальной скорости и времени рождения. Ниже показан один из возможных вариантов реализации:

```
glBindVertexArray(myVArray);

// Определить указатели для атрибутов 0, 1 и 2 (координаты,
// нормаль и координаты текстуры)
...

// Начальная скорость (атрибут 3)
glBindBuffer(GL_ARRAY_BUFFER, initVel);
glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(3);
glVertexAttribDivisor(3, 1);

// Время рождения (атрибут 4)
glBindBuffer(GL_ARRAY_BUFFER, startTime);
glVertexAttribPointer(4, 1, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(4);
glVertexAttribDivisor(4, 1);

// Связать буфер с элементом массива, если необходимо
```

Обратите внимание, как используется функция `glVertexAttribDivisor` в этом фрагменте. Здесь с ее помощью определяется, что атрибуты `3` и `4` являются атрибутами экземпляров (один элемент в массиве значений атрибута соответствует экземпляру, а не вершине). Соответственно, размеры буферов должны быть пропорциональны числу экземпляров, а не числу вершин в экземплярах. Буферы для значений атрибутов `0`, `1` и `2` должны (как обычно) иметь размеры, соответствующие числу вершин.

Значение делителя атрибута вершины будет сохранено в объекте массива вершин наряду с другой информацией, благодаря чему оно может меняться при связывании других объектов массивов вершин.

Как это делается...

Выполните следующие шаги:

1. Вершинный шейдер в данном примере почти идентичен вершинному шейдеру из рецепта «Создание фонтана частиц». Разница заключается в объявлениях входных и выходных переменных:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec3 VertexTexCoord;
layout (location = 3) in vec3 VertexInitialVelocity;
layout (location = 4) in float StartTime;

out vec3 Position;
out vec3 Normal;
```

2. Внутри функции `main` измените обновление позиции вершины, как показано ниже:

```
Position = VertexPosition + VertexInitialVelocity * t +
           Gravity * t * t;
```

3. Не забудьте вместе с обновленной позицией передать фрагментному шейдеру вектор нормали (в видимых координатах).
4. Внутри фрагментного шейдера реализуйте модель затенения по своему выбору.
5. В основной программе, в функции отображения, выполните отображение экземпляров:

```
glBindVertexArray(myVArray);
glDrawElementsInstanced(GL_TRIANGLES, nEls,
              GL_UNSIGNED_INT, 0, nParticles);
```

Как это работает...

Напомню, что первые три входных атрибута вершинного шейдера не являются атрибутами экземпляров, в том смысле что значения этих атрибутов определяются для каждой вершины (и повторяются для каждого экземпляра). Последние два (атрибуты с индексами `3` и `4`), напротив, являются атрибутами экземпляров, и их значения определяются для экземпляров целиком. В результате каждый экземпляр перемещается в соответствии с заданным уравнением движения.

Функция `glDrawElementsInstanced` (шаг 5) нарисует `nParticles` экземпляров объекта. Как вы уже наверняка догадались, `nEls` – это число вершин в каждом экземпляре.

И еще...

Вершинные шейдеры в OpenGL имеют встроенную переменную `gl_InstanceID`. Это простой счетчик, в котором передаются порядковые номера отображаемых экземпляров. Для первого экземпляра этот счетчик будет хранить значение `0`, для второго – `1` и т. д. Это может пригодиться для индексации данных в текстуре. Также порядковый номер можно использовать, например, в качестве начального значения генератора случайных чисел и получить для каждого экземпляра свою последовательность случайных чисел.

См. также

- Рецепт «Создание фонтана частиц».
- Рецепт «Создание системы частиц с использованием трансформации с обратной связью».

Имитация пламени с помощью частиц

Чтобы создать эффект, напоминающий пламя, достаточно внести всего несколько небольших изменений в нашу первую, простую систему частиц. Так как пламя

весьма незначительно подвержено влиянию силы тяготения, можно особенно не заботиться об использовании ускорения свободного падения в расчетах. В действительности в этом примере вектор ускорения будет направлен вверх, чтобы обеспечить эффект рассеивания частиц в верхней части пламени. Кроме того, начальные позиции частиц будут равномерно распределяться по горизонтали, чтобы основание пламени не было сосредоточено в одной точке. Здесь также будет использоваться текстура для частиц, чтобы получить красные и оранжевые тона, характерные для пламени.

На рис. 9.6 показан пример эффекта пламени, реализованного с помощью системы частиц.

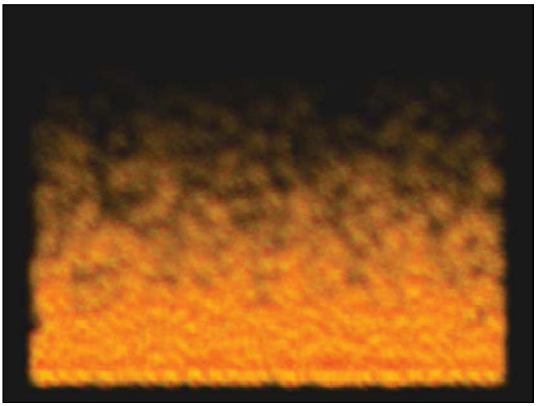

Рис. 9.6 ❖ Пример эффекта пламени, реализованного с помощью системы частиц

Текстура, используемая для отображения частиц, имеет вид фрагмента пламени с красно-оранжевыми цветами. Она не показана здесь, потому что, напечатанная на бумаге, не позволяет получить полное представление о ней.

Подготовка

Возьмите за основу систему частиц из рецепта «Создание системы частиц с использованием трансформации с обратной связью».

Объявите uniform-переменную `Accel` и инициализируйте ее небольшим значением, таким как `(0.0, 0.1, 0.0)`.

Объявите uniform-переменную `ParticleLifetime` и присвойте ей значение, близкое к четырем секундам.

Создайте и загрузите текстуру для отображения частиц с цветами пламени. Свяжите ее с первым текстурным слотом и присвойте uniform-переменной `ParticleTex` значение `0`.

Установите размер точек равным примерно 50.0. Это оптимальный размер для текстуры, используемой в данном рецепте, но вы можете выбрать другой размер, в зависимости от числа частиц и текстуры.

Как это делается...

Выполните следующие шаги:

1. При установке начальных позиций частиц координату X следует генерировать случайным образом, например:

```
GLfloat *data = new GLfloat[nParticles * 3];
for( int i = 0; i < nParticles * 3; i += 3 ) {
  data[i] = glm::mix(-2.0f, 2.0f, randFloat());
  data[i+1] = 0.0f;
  data[i+2] = 0.0f;
}
glBindBuffer(GL_ARRAY_BUFFER, posBuf[0]);
glBufferSubData(GL_ARRAY_BUFFER, 0, size, data);
```

2. При определении начальных скоростей компонентам `x` и `z` вектора скорости следует присвоить нулевые значения, а компоненту `y` – случайное значение. Это в сочетании с выбранным ускорением (см. предыдущий фрагмент кода) заставит частицы двигаться только в вертикальном направлении (вдоль оси Y):

```
// Заполнить сначала буфер скоростей случайными значениями
for( unsigned int i = 0; i < nParticles; i++ ) {
  data[3*i] = 0.0f;
  data[3*i+1] = glm::mix(0.1f,0.5f,randFloat());
  data[3*i+2] = 0.0f;
}
glBindBuffer(GL_ARRAY_BUFFER,velBuf[0]);
glBufferSubData(GL_ARRAY_BUFFER, 0, size, data);
glBindBuffer(GL_ARRAY_BUFFER,initVel);
glBufferSubData(GL_ARRAY_BUFFER, 0, size, data);
```

3. В вершинном шейдере, когда истечет время жизни частицы, нужно сбросить только координаты `y` и `z`, а координату `x` оставить как есть:

```
if( age > ParticleLifetime ) {
  // Время жизни частицы истекло
  Position = vec3(VertexPosition.x, 0.0, 0.0);
  Velocity = VertexInitialVelocity;
  StartTime = Time;
}
```

Как это работает...

В этом рецепте координаты X частиц равномерно распределяются в диапазоне от `-2.0` до `2.0` случайным образом, а начальные скорости – в диапазоне от `0.1` до `0.5` вдоль оси Y. Так как в векторе ускорения определен только компонент `y`, частицы будут перемещаться строго по вертикали, вверх. Компонент `x` или `z` вектора с координатами частицы всегда должен оставаться равным нулю. То есть по истечении срока жизни частицы достаточно просто сбросить ее координату Y в нулевое значение, чтобы переместить ее в начальную позицию и подготовить к повторному использованию.

И еще...

Конечно, чтобы получить колеблющееся пламя, перемещающееся в разных направлениях (например, из-за ветра), ускорение должно принимать разные значения. И в этом случае наша маленькая хитрость с возвратом частиц в начальную позицию перестанет работать. Но в этом нет никакой проблемы, достаточно добавить в систему частиц еще один буфер (подобный буферу с начальными скоростями), сохранить в нем начальные позиции частиц и извлекать их оттуда в нужные моменты времени.

См. также

- Рецепт «Создание системы частиц с использованием трансформации с обратной связью».

Имитация дыма с помощью частиц

Как известно, дым – это множество маленьких частиц, поднимающихся вверх от источника и расплывающихся по сторонам по мере движения вверх. Сымитировать эффект всплытия частиц можно, определив небольшое ускорение вверх (или задав постоянную скорость), но сымитировать диффузное распространение частиц в стороны будет не так просто. Однако сымитировать диффузное распространение множества маленьких частиц можно посредством увеличения их размеров с течением времени.

На рис. 9.7 показано, как выглядит имитация дыма, созданная в данном рецепте.

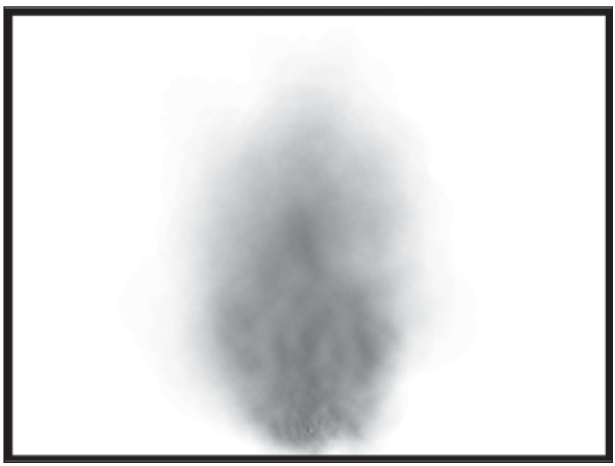

Рис. 9.7 ❖ Имитация дыма

Текстура, которая будет накладываться на каждую частицу, представляет собой «размытое» пятно в серых тонах.

Для реализации увеличения частиц в размерах с течением времени будет использоваться функция системы OpenGL под названием `GL_PROGRAM_POINT_SIZE`, которая позволяет изменять размеры точек в вершинном шейдере.

При желании можно было бы рисовать квадраты, в том числе и с помощью геометрического шейдера, как было показано в главе 6 «Использование геометрических шейдеров и шейдеров тесселяции».

Подготовка

Возьмите за основу систему частиц из рецепта «Создание системы частиц с использованием трансформации с обратной связью».

Объявите uniform-переменную `Accel` и инициализируйте ее небольшим значением, таким как `(0.0, 0.1, 0.0)`.

Присвойте uniform-переменной `ParticleLifetime` значение, близкое к шести секундам.

Создайте и загрузите текстуру для отображения частиц с изображением дыма. Свяжите ее с первым текстурным слотом и присвойте uniform-переменной `ParticleTex` значение `0`.

Присвойте uniform-переменным `MinParticleSize` и `MaxParticleSize` значения `10` и `200` соответственно.

Как это делается...

Выполните следующие шаги:

1. Установите исходные позиции частиц в начало координат. Определите начальные скорости, как было описано в рецепте «Создание системы частиц с использованием трансформации с обратной связью». Однако эффект получится более зрелищным, если выбрать более широкий диапазон изменения значений `theta`.
2. Добавьте следующие uniform-переменные в вершинный шейдер:

```
uniform float MinParticleSize;
uniform float MaxParticleSize;
```

3. Там же, внутри вершинного шейдера, замените реализацию функции `render`:

```
subroutine (RenderPassType)
void render() {
  float age = Time - VertexStartTime;
  Transp = 0.0;
  if( Time >= VertexStartTime ) {
    float agePct = age/ParticleLifetime;
    Transp = 1.0 - agePct;
    gl_PointSize =
      mix(MinParticleSize,MaxParticleSize,agePct);
  }
  gl_Position = MVP * vec4(VertexPosition, 1.0);
}
```

4. В основной программе перед отображением частиц включите функцию GL_PROGRAM_POINT_SIZE:

```
glEnable(GL_PROGRAM_POINT_SIZE);
```

Как это работает...

Функция render подпрограммы присваивает встроенной переменной gl_PointSize значение в диапазоне от MinParticleSize до MaxParticleSize в зависимости от возраста частицы, в результате чего частицы растут в размерах с течением времени.

Обратите внимание, что переменная gl_PointSize игнорируется, если не включена функция OpenGL GL_PROGRAM_POINT_SIZE.

См. также

- Рецепт «Создание системы частиц с использованием трансформации с обратной связью».

Глава 10

Вычислительные шейдеры

В этой главе описываются следующие рецепты:

- реализация системы частиц с помощью вычислительного шейдера;
- имитация полотнища ткани с помощью вычислительного шейдера;
- определение границ с помощью вычислительного шейдера;
- создание фракталов с помощью вычислительного шейдера.

Введение

Впервые **вычислительные шейдеры (compute shaders)** появились в версии OpenGL 4.3. Вычислительный шейдер представляет этап в шейдерной программе, в ходе которого можно выполнять произвольные вычисления. Он дает возможность использовать мощь GPU и присущий ему параллелизм для решения любых вычислительных задач, которые прежде можно было реализовать только на CPU. Вычислительные шейдеры с особым успехом можно использовать для задач, не связанных непосредственно с отображением графики, таких как симуляция физических процессов.

В настоящее время уже существуют библиотеки, обеспечивающие возможность выполнения универсальных вычислений на GPU, такие как OpenCL и CUDA, однако они никак не связаны с OpenGL. Вычислительные шейдеры, напротив, интегрированы непосредственно в OpenGL и, как следствие, лучше подходят для организации вычислений, имеющих отношение к отображению графики.

Вычислительный шейдер не является традиционным этапом выполнения шейдерной программы, как фрагментный или вершинный шейдер. Он не вызывается в ответ на команды отображения. Фактически, когда вычислительный шейдер компонуется с вершинным, фрагментным или другими шейдерами, он остается инертным в отношении команд рисования. Единственный способ запустить вычислительный шейдер – выполнить OpenGL-команду `glDispatchCompute` или `glDispatchComputeIndirect`.

Вычислительные шейдеры не имеют входных и выходных переменных, определяемых пользователем. Все исходные данные они могут получать только прямым обращением к памяти, используя функции доступа к изображениям или посредством буферных объектов хранилищ. Аналогично результаты вычислений должны сохраняться в те же самые или в другие объекты. Исключение составляет

множество входных переменных, определяющих место вызова шейдера в «пространстве» выполнения.

Число вызовов вычислительного шейдера целиком и полностью определяется пользователем. Оно никак не связано ни с числом вершин, ни с числом фрагментов. Число вызовов задается числом рабочих групп и числом вызовов в каждой рабочей группе.

Вычислительное пространство и рабочие группы

Число вызовов вычислительного шейдера зависит от вычислительного пространства, определяемого пользователем. Это пространство делится на множество рабочих групп. Каждая рабочая группа, в свою очередь, разбивается на число вызовов. Такую организацию можно представить как глобальное вычислительное пространство (все вызовы шейдера), которое делится на локальные пространства групп (вызовы внутри каждой конкретной рабочей группы). Вычислительное пространство может иметь 1, 2 или 3 измерения.

Технически вычислительное пространство всегда имеет три измерения, но любое из трех измерений может быть определено как единичное (с размером, равным 1), что фактически приводит к исключению измерения из пространства.

Например, на рис. 10.1 показано одномерное вычислительное пространство с пятью рабочими группами, по три вызова в каждой. Толстыми рамками окружены рабочие группы, а тонкими – вызовы в каждой рабочей группе.

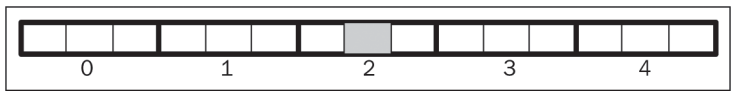

Рис. 10.1 ❖ Одномерное вычислительное пространство с пятью рабочими группами, по три вызова в каждой

В данном случае имеется `5 * 3 = 15` вызовов шейдера. Серым цветом в рабочей группе `2` выделен вызов с индексом `1` (отсчет индексов вызовов начинается с нуля). В глобальном пространстве этот же вызов имеет глобальный индекс `7`, отсчет которых также начинается с нуля. Глобальный индекс определяет местоположение вызова в глобальном вычислительном пространстве, а не внутри какой-то группы. Он является результатом произведения индекса рабочей группы (`2`) на общее число вызовов в рабочей группе (`3`) плюс локальный индекс (`1`), то есть `2 * 3 + 1 = 7`. Глобальный индекс – это просто индекс в общем, глобальном пространстве вызовов.

На рис. 10.2 показано, как выглядит двухмерное вычислительное пространство, поделенное на 20 рабочих групп: четыре по оси X и пять по оси Y. Каждая рабочая группа, в свою очередь, делится на девять вызовов: три по оси X и три по оси Y.

Ячейка, выделенная серым цветом, представляет вызов `(0, 1)` в рабочей группе `(2, 0)`. Общее число вызовов вычислительного шейдера в данном примере составляет `20 * 9 = 180`. Глобальный индекс выделенного вызова равен `(6, 1)`. Как и в слу-

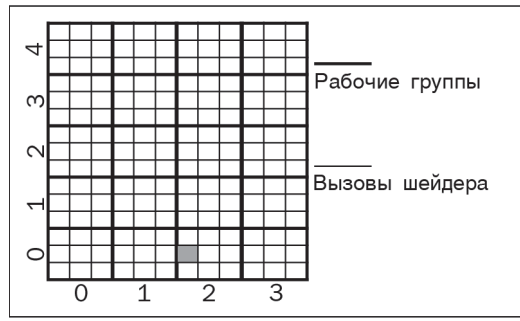

Рис. 10.2 ❖ Двухмерное вычислительное пространство, поделенное на 20 рабочих групп

чае с одномерным вычислительным пространством, данный индекс определяется (для каждой оси) как произведение числа вызовов в рабочей группе на индекс рабочей группы плюс локальный индекс вызова. Для оси X получается: `3 * 2 + 0 = 6`; для оси Y: `3 * 0 + 1 = 1`.

Эту идею легко распространить на трехмерное вычислительное пространство. Вообще, размерность выбирается в зависимости от организации обрабатываемых данных. Например, при работе с системой частиц обрабатывается простой список частиц, поэтому имеет смысл использовать одномерное вычислительное пространство. С другой стороны, в имитации полотнища ткани данные организованы в виде таблицы, поэтому для их обработки лучше подходит двухмерное вычислительное пространство.

Общее число рабочих групп и локальных вызовов шейдера ограничено. Узнать эти ограничения можно программно (с помощью семейства функций `glGetInteger*`), используя параметры `GL_MAX_COMPUTE_WORK_GROUP_COUNT`, `GL_MAX_COMPUTE_WORK_GROUP_SIZE` и `GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS`.

Порядок выполнения рабочих групп и, как следствие, отдельных вызовов шейдера не определен, и система может выполнять их в любом порядке. Поэтому не следует полагаться на какой-то определенный порядок выполнения рабочих групп. Локальные вызовы внутри каждой конкретной группы выполняются параллельно (если это возможно). Поэтому любые взаимодействия между вызовами должны осуществляться с большой осторожностью. Взаимодействия между вызовами внутри рабочей группы возможны с использованием локальных разделяемых данных, но вызовы в разных рабочих группах не должны пытаться взаимодействовать друг с другом (в общем случае) без учета всех возможных проблем, таких как взаимоблокировка и гонка за ресурсами. В действительности те же проблемы могут возникать также при взаимодействиях между вызовами в пределах одной рабочей группы, поэтому и в этом случае следует предпринять все меры, чтобы избежать их. Вообще говоря, из соображений эффективности лучше стремиться ограничить взаимодействия рамками одной рабочей группы. В параллельном программировании на GLSL, как и на любом другом языке, «прячутся драконы».

OpenGL поддерживает множество атомарных операций и барьеров, способных помочь в организации взаимодействий между вызовами. С некоторыми из них мы познакомимся в последующих рецептах.

Выполнение вычислительного шейдера

Запуская вычислительный шейдер на выполнение, необходимо определить вычислительное пространство. Число рабочих групп определяется параметрами функции `glDispatchCompute`. Например, чтобы выполнить вычислительный шейдер в двухмерном вычислительном пространстве с 4 рабочими группами по оси X и 5 рабочими группами по оси Y (это вычислительное пространство показано на рис. 10.2), можно произвести следующий вызов:

```
glDispatchCompute( 4, 5, 1 );
```

Число локальных вызовов внутри каждой рабочей группы определяется не основной программой, а самим вычислительным шейдером, с помощью спецификатора `layout`. Например, следующая инструкция задает девять локальных вызовов на рабочую группу: три по оси X и три по оси Y.

```
layout (local_size_x = 3, local_size_y = 3) in;
```

Размер по оси Z можно опустить (по умолчанию он равен единице).

Когда выполняется конкретный вызов вычислительного шейдера, обычно бывает необходимо определить его местоположение в глобальном вычислительном пространстве. В GLSL для этого имеется множество встроенных входных переменных. Большая часть из них перечислена в следующей таблице:

Переменная	Тип	Назначение
`gl_WorkGroupSize`	`uvec3`	Число вызовов на рабочую группу в каждом измерении. Это значение определяется спецификатором `layout`
`gl_NumWorkGroups`	`uvec3`	Общее число рабочих групп в каждом измерении
`gl_WorkGroupID`	`uvec3`	Индекс текущей рабочей группы
`gl_LocalInvocationID`	`uvec3`	Индекс текущего вызова внутри рабочей группы
`gl_GlobalInvocationID`	`uvec3`	Индекс текущего вызова в глобальном вычислительном пространстве

Значение последней переменной из предыдущей таблицы (`gl_GlobalInvocationID`) вычисляется как (все операции выполняются покомпонентно):

```
gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID
```

С ее помощью можно определить местоположение текущего вызова в глобальном вычислительном пространстве (см. предыдущие примеры).

В языке GLSL также имеется переменная `gl_LocalInvocationIndex` – «плоская» форма индекса `gl_LocalInvocationID`. Она может пригодиться в ситуациях, когда многомерные данные хранятся в линейном буфере, но в примерах, следующих далее, эта переменная не используется.

Реализация системы частиц с помощью вычислительного шейдера

В этом рецепте будет реализована простая имитация. Вычислительный шейдер будет производить вычисления в соответствии с физической моделью и изменять координаты частиц, а затем частицы будут отображаться как точки. Без вычислительного шейдера пришлось бы изменять координаты частиц в основной программе, последовательно обходя их в массиве, или использовать прием трансформации с обратной связью, как было показано в рецепте «Создание системы частиц с использованием трансформации с обратной связью» в главе 9 «Системы частиц и анимация». Реализации подобных анимаций с применением вершинного шейдера получаются запутанными и требуют приложения дополнительных усилий (например, по настройке обратной связи). С помощью же вычислительного шейдера расчеты можно выполнять на GPU параллельно, настроив вычислительное пространство так, чтобы получить наибольшую отдачу.

На рис. 10.3 показано, как выглядит имитация системы из одного миллиона частиц. Каждая частица отображается как точка с размерами 1×1. Частицы полупрозрачны, а аттракторы частиц отображаются как небольшие квадраты 5×5 (едва видимые).

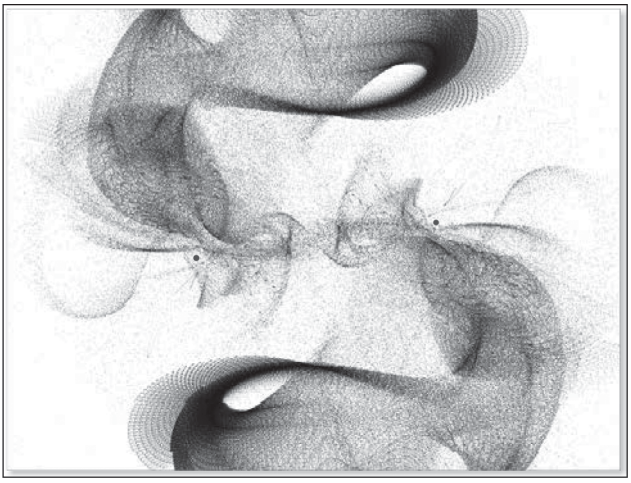

Рис. 10.3 ❖ Имитация системы из одного миллиона частиц

Подобные имитации могут создавать интересные, абстрактные фигуры, а их создание доставляет массу удовольствия.

Для данной имитации нужно определить несколько аттракторов (два в этом случае, но при желании их можно создать больше), которые я буду называть **черными дырами**. Они являются единственными объектами, воздействующими на

частицы с определенной силой, величина которой обратно пропорциональна расстоянию от частицы до черной дыры. Более формально сила, воздействующая на каждую частицу, определяется следующей формулой:

$$F = \sum_{i=1}^{N} \frac{G_i}{|r_i|} \frac{r_i}{|r_i|},$$

где N – число черных дыр (аттракторов), r_i – вектор между i-м аттрактором и частицей (определяется как разность позиции аттрактора и позиции частицы), G_i – мощность i-го аттрактора.

Для реализации имитации необходимо вычислить силу, воздействующую на каждую частицу, и изменить позицию этой частицы, интегрируя уравнения движения Ньютона. Существует множество хорошо зарекомендовавших себя численных методов интегрирования уравнений движения. Для данного примера вполне достаточно будет простого метода Эйлера. Согласно методу Эйлера, позиция частицы в момент времени $t + \Delta t$ определяется следующим уравнением:

$$P(t + \Delta t) = P(t) + v(t)\Delta t + \frac{1}{2} a(t)\Delta t^2,$$

где P – позиция частицы, v – скорость и a – ускорение. Аналогично определяется скорость:

$$v(t + \Delta t) = v(t) + a(t)\Delta t.$$

Эти уравнения получены разложением в ряд Тейлора функции позиции в момент времени t. Результат зависит от размера приращения времени (Δt) и показывает высокую точность при очень маленьких приращениях времени.

Ускорение прямо пропорционально силе воздействия на частицу, то есть, вычисляя силу (используя уравнение выше), мы фактически получаем ускорение. С целью имитации движения для каждой частицы здесь будут сохраняться координаты и скорость, определяться сила воздействия черных дыр и затем изменяться координаты, с использованием уравнений.

Все расчеты по формулам будут выполняться вычислительным шейдером. Так как предполагается использовать простой список частиц, вычислительное пространство будет иметь единственное измерение с рабочими группами по 1000 частиц. Каждый вызов вычислительного шейдера будет отвечать за изменение позиции единственной частицы.

Для хранения позиций и скоростей будет использоваться буферный объект хранилища, благодаря чему во время отображения частиц мы сможем использовать буфер с координатами непосредственно.

Подготовка

В основной программе нам потребуется один буфер для координат частиц и один буфер для скоростей. Создайте буфер с координатами, инициализированный начальными позициями частиц, и буфер с нулевыми начальными скоростями. В этом примере будут использоваться четырехкомпонентные координаты и ско-

рости, чтобы избежать проблем с организацией хранения данных. Например, создать буфер с координатами можно примерно так:

```
vector<GLfloat> initPos;

... // Установить начальные координаты

GLuint bufSize = totalParticles * 4 * sizeof(GLfloat);

GLuint posBuf;
glGenBuffers(1, &posBuf);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, posBuf);
glBufferData(GL_SHADER_STORAGE_BUFFER, bufSize, &initPos[0],
             GL_DYNAMIC_DRAW);
```

Буфер со скоростями создается аналогично, только связать его нужно с индексом 1 в точке привязки `GL_SHADER_STORAGE_BUFFER`:

```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, velBuf);
```

Подготовьте объект массива вершин, использующий тот же буфер с позициями в качестве источника координат вершин.

Для отображения точек подготовьте вершинный и фрагментный шейдеры, которые просто выводят фрагменты, окрашенные сплошным цветом. Разрешите смешивание, задействовав стандартную функцию смешивания.

Как это делается...

Выполните следующие шаги:

1. Добавьте вычислительный шейдер, изменяющий позиции частиц.

```
layout( local_size_x = 1000 ) in;

uniform float Gravity1 = 1000.0;
uniform vec3 BlackHolePos1;
uniform float Gravity2 = 1000.0;
uniform vec3 BlackHolePos2;

uniform float ParticleInvMass = 1.0 / 0.1;
uniform float DeltaT = 0.0005;

layout(std430, binding=0) buffer Pos {
  vec4 Position[];
};

layout(std430, binding=1) buffer Vel {
  vec4 Velocity[];
};

void main() {
  uint idx = gl_GlobalInvocationID.x;

  vec3 p = Position[idx].xyz;
  vec3 v = Velocity[idx].xyz;
```

```
    // Сила воздействия черной дыры #1
    vec3 d = BlackHolePos1 - p;
    vec3 force = (Gravity1 / length(d)) * normalize(d);

    // Сила воздействия черной дыры #2
    d = BlackHolePos2 - p;
    force += (Gravity2 / length(d)) * normalize(d);

    // Выполнить интегрирование простым методом Эйлера
    vec3 a = force * ParticleInvMass;
    Position[idx] = vec4(
        p + v * DeltaT + 0.5 * a * DeltaT * DeltaT, 1.0);
    Velocity[idx] = vec4( v + a * DeltaT, 0.0);
}
```

2. В функции отображения, в основной программе, вызовите вычислительный шейдер, чтобы изменить позиции частиц.

```
glDispatchCompute(totalParticles / 1000, 1, 1);
```

3. Убедитесь, что все результаты записаны в буфер, приостановившись на барьере.

```
glMemoryBarrier( GL_SHADER_STORAGE_BARRIER_BIT );
```

4. В заключение отобразите частицы, используя данные в буфере с координатами.

Как это работает...

Вычислительный шейдер начинается с определения числа вызовов на группу:

```
layout( local_size_x = 1000 ) in;
```

Эта инструкция указывает, что каждая рабочая группа содержит 1000 вызовов по оси X. Вы можете выбрать другое значение, лучше соответствующее вашему аппаратному обеспечению, но не забудьте соответствующим образом скорректировать число рабочих групп. По умолчанию каждое измерение вычислительного пространства имеет размер 1, поэтому не требуется определять размеры по осям Y и Z.

Затем объявляются uniform-переменные, предназначенные для хранения параметров имитации. Переменные `Gravity1` и `Gravity2` определяют мощность двух черных дыр (член G в уравнении выше), а переменные `BlackHolePos1` и `BlackHolePos2` – их позиции. Переменная `ParticleInvMass` хранит значение, обратное массе одной частицы, – оно будет использоваться для преобразования силы в ускорение. Наконец, переменная `DeltaT` определяет приращение времени, используемое в методе Эйлера интегрирования уравнений движения.

Далее объявляются буферы с координатами и скоростями. Обратите внимание, что значения `binding` здесь соответствуют индексам привязки в основной программе, которые использовались при инициализации буферов.

В функции `main` сначала определяется индекс частицы. Так как обрабатывается линейный список частиц и число частиц совпадет с числом вызовов шейдера,

индекс частицы прямо соответствует индексу вызова в глобальном вычислительном пространстве. Этот индекс доступен в виде встроенной входной переменной `gl_GlobalInvocationID.x`. Мы используем глобальный индекс, потому что нам нужен индекс в границах всего буфера, – локальный индекс в рамках рабочей группы адресует лишь часть общего массива.

Далее из буферов извлекаются координаты частицы и ее скорость, вычисляется сила, с какой воздействует каждая черная дыра, и в переменной `force` сохраняется сумма сил. Затем сила преобразуется в ускорение, и с помощью метода Эйлера вычисляются новые координаты и скорость частицы. Результаты сохраняются в те же элементы буферов, откуда извлекались исходные данные. Так как разные вызовы не имеют совместно используемых данных, операция сохранения не несет никакой угрозы.

В функции отображения, в основной программе, инициируется запуск вычислительного шейдера (шаг 2 в разделе «Как это делается...») и при этом определяется число рабочих групп на каждое измерение. В вычислительном шейдере размер рабочей группы установлен равным `1000`. Так как предполагается, что один вызов будет обрабатывать одну частицу, число рабочих групп можно получить делением общего числа частиц на `1000`.

Наконец, в шаге 3 перед отображением частиц нужно приостановить выполнение программы на барьере, чтобы дождаться, пока все вызовы вычислительного шейдера будут выполнены.

См. также

- Другие реализации систем частиц в главе 9 «Системы частиц и анимация». Большинство из них реализовано с применением механизма трансформации с обратной связью, но их точно так же можно реализовать и с помощью вычислительного шейдера.

Имитация полотнища ткани с помощью вычислительного шейдера

Вычислительный шейдер хорошо подходит для симуляции физических процессов с использованием GPU. Отличным примером может служить имитация движения полотнища ткани. В этом рецепте представлена простая имитация на основе системы «подпружиненных» частиц, реализованная с применением вычислительного шейдера. На рис. 10.4 показано, как выглядит имитация полотнища ткани, закрепленного пятью прищепками. (Представьте себе, как эта ткань колышется.)

Система частиц, связанных между собой «пружинами», – это распространенный способ имитации полотнищ ткани. Модель полотнища представляет собой двухмерную сетку центров масс, каждый из которых связан с соседними центрами идеальными пружинами. На рис. 10.5 показан один из таких центров масс (в середине), связанный с соседними центрами. Соединительные линии на рисунке

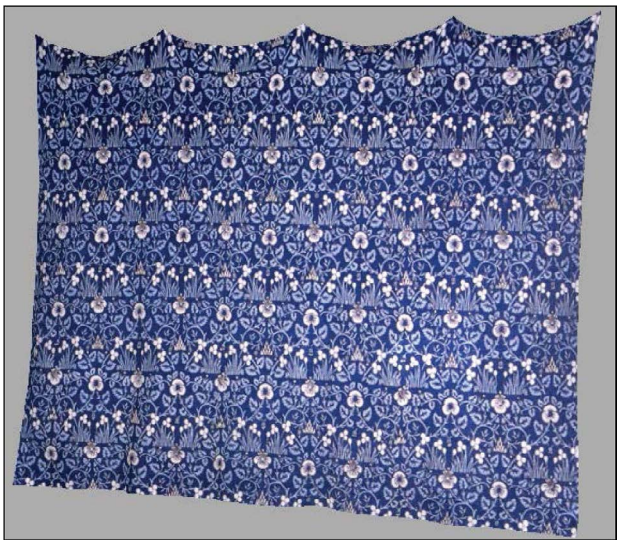

Рис. 10.4 ❖ Имитация куска ткани, закрепленного пятью прищепками

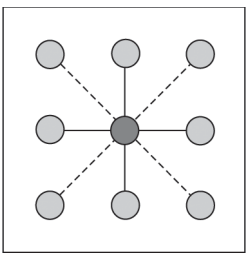

Рис. 10.5 ❖ Модель полотнища: двухмерная сетка центров масс, каждый из которых связан с соседними центрами идеальными пружинами

представляют пружины. Сплошными линиями изображены вертикальные и горизонтальные пружины, а пунктирными – диагональные.

Общая сила, действующая на частицу, является суммой сил, обусловленных действием восьми пружин. Сила воздействия одной пружины определяется следующим уравнением:

$$F = K(|r|-R)\frac{r}{|r|},$$

где K – жесткость пружины, R – длина пружины в состоянии покоя (когда сила действия пружины равна нулю) и r – вектор между соседней и данной частицами (определяется как разность позиций соседней и данной частиц).

Как и в предыдущем рецепте, суть имитации сводится к вычислению суммы сил, воздействующих на каждую частицу, и интегрированию уравнений движения Ньютона с использованием того или иного численного метода. В этом примере снова будет использоваться метод Эйлера. Более полное описание этого метода приводится в предыдущем рецепте «Реализация системы частиц с помощью вычислительного шейдера».

Совершенно очевидно, что модель полотнища в виде сетки из подпружиненных частиц имеет двухмерную структуру, поэтому имеет смысл отобразить ее в двухмерное вычислительное пространство. Мы определим двухмерные рабочие группы и в каждом вызове шейдера будем обрабатывать одну частицу. В каждом вызове нужно будет прочитать координаты восьми соседних частиц, вычислить сумму сил, воздействующих на данную частицу, и обновить ее координаты и скорость.

Обратите внимание, что в каждом вызове необходимо будет прочитать координаты соседних частиц, которые будут изменяться другими вызовами шейдера. Так как система OpenGL не гарантирует какого-то определенного порядка следования вызовов шейдера, мы не сможем использовать для чтения и записи один и тот же буфер, ибо мы не можем быть уверены, что читаем еще не обновленные, исходные координаты соседних частиц. Чтобы исключить эту проблему, будет использоваться пара буферов. В каждом шаге имитации один буфер будет служить источником исходных данных, а другой – хранилищем результатов. В конце шага мы будем менять буферы местами и повторять цикл имитации.

Теоретически можно было бы использовать один и тот же буфер для чтения и для записи, используя локальную разделяемую память; однако в этом случае описанная проблема сохраняется для частиц, расположенных вдоль границ рабочих групп: соседние с ними частицы находятся в других группах, локальная память которых недоступна из данной группы.

Данная модель весьма чувствительна к точности вычислений, поэтому здесь будет использоваться очень маленький шаг интегрирования по времени. Вполне приемлемые результаты дает значение `0.000005`. Кроме того, имитация выглядит более реалистично, когда к частицам применяется демпфирующая сила, имитирующая сопротивление воздуха. Для этого мы будем добавлять в силу, величина которой прямо пропорциональна длине и обратно пропорциональна направлению вектора скорости:

$$F = -Dv,$$

где D – величина демпфирующей силы и v – вектор скорости частицы.

Подготовка

Сначала создайте два буфера для координат и два буфера для скоростей частиц. Свяжите буферы координат с индексами `0` и `1` в точке привязки `GL_SHADER_STORAGE_BUFFER`, а буферы скоростей – с индексами `2` и `3`. Организация данных в этих буферах играет важную роль. Координаты/скорости будут располагаться в буферах построчно, в направлении от левого нижнего угла решетки к правому верхнему.

Создайте также объект массива вершин для рисования полотнища, используя частицы как вершины треугольников. Нам также потребуются буферы для хранения нормалей и координат текстуры. Для краткости я опущу их из обсуждения, но в программном коде примера они присутствуют.

Как это делается...

Выполните следующие шаги:

1. Определите в вычислительном шейдере число вызовов на одну рабочую группу.

```
layout( local_size_x = 10, local_size_y = 10 ) in;
```

2. Определите uniform-переменные для параметров модели.

```
uniform vec3 Gravity = vec3(0,-10,0);
uniform float ParticleMass = 0.1;
uniform float ParticleInvMass = 1.0 / 0.1;
uniform float SpringK = 2000.0;
uniform float RestLengthHoriz;
uniform float RestLengthVert;
uniform float RestLengthDiag;
uniform float DeltaT = 0.000005;
uniform float DampingConst = 0.1;
```

3. Объявите пары буферов для хранения координат и скоростей.

```
layout(std430, binding=0) buffer PosIn {
  vec4 PositionIn[];
};
layout(std430, binding=1) buffer PosOut {
  vec4 PositionOut[];
};
layout(std430, binding=2) buffer VelIn {
  vec4 VelocityIn[];
};
layout(std430, binding=3) buffer VelOut {
  vec4 VelocityOut[];
};
```

4. В функции `main` нужно извлечь координаты частицы, за обработку которой отвечает данный вызов.

```
void main() {
  uvec3 nParticles = gl_NumWorkGroups * gl_WorkGroupSize;
  uint idx = gl_GlobalInvocationID.y * nParticles.x +
             gl_GlobalInvocationID.x;
  vec3 p = vec3(PositionIn[idx]);
  vec3 v = vec3(VelocityIn[idx]), r;
```

5. Инициализировать `force` силой притяжения.

```
  vec3 force = Gravity * ParticleMass;
```

6. Добавить силу воздействия частицы сверху.

```
if( gl_GlobalInvocationID.y < nParticles.y - 1 ) {
  r = PositionIn[idx + nParticles.x].xyz - p;
  force += normalize(r)*SpringK*(length(r) -
  RestLengthVert);
}
```

7. Повторить предыдущий шаг для частиц снизу, слева и справа. Затем добавить силу воздействия частицы по диагонали, слева сверху.

```
if( gl_GlobalInvocationID.x > 0 &&
    gl_GlobalInvocationID.y < nParticles.y - 1 ) {
  r = PositionIn[idx + nParticles.x - 1].xyz - p;
  force += normalize(r)*SpringK*(length(r) -
  RestLengthDiag);
}
```

8. Повторить предыдущий шаг для трех других диагональных частиц. Затем добавить демпфирующую силу.

```
force += -DampingConst * v;
```

9. Далее выполнить интегрирование уравнений движения методом Эйлера.

```
vec3 a = force * ParticleInvMass;
PositionOut[idx] = vec4(
    p + v * DeltaT + 0.5 * a * DeltaT * DeltaT, 1.0);
VelocityOut[idx] = vec4( v + a * DeltaT, 0.0);
```

10. Наконец, «закрепить» несколько частиц сверху «прищепками», чтобы они никуда не смещались.

```
if( gl_GlobalInvocationID.y == nParticles.y - 1 &&
    (gl_GlobalInvocationID.x == 0 ||
     gl_GlobalInvocationID.x == nParticles.x / 4 ||
     gl_GlobalInvocationID.x == nParticles.x * 2 / 4 ||
     gl_GlobalInvocationID.x == nParticles.x * 3 / 4 ||
     gl_GlobalInvocationID.x == nParticles.x - 1)) {
  PositionOut[idx] = vec4(p, 1.0);
  VelocityOut[idx] = vec4(0,0,0,0);
}
}
```

11. В функции отображения, в основной программе, вызовите вычислительный шейдер так, чтобы каждая рабочая группа обрабатывала 100 частиц. Так как шаг во времени очень мал, процесс обработки потребуется выполнить много раз (1000), при этом каждый раз меняя буферы местами.

```
for( int i = 0; i < 1000; i++ ) {
  glDispatchCompute(nParticles.x/10, nParticles.y/10, 1);
  glMemoryBarrier( GL_SHADER_STORAGE_BARRIER_BIT );

  // Поменять буферы
  readBuf = 1 - readBuf;

  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,0,
```

```
                    posBufs[readBuf]);
  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,1,
                    posBufs[1-readBuf]);
  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,2,
                    velBufs[readBuf]);
  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,3,
                    velBufs[1-readBuf]);
}
```

12. В заключение нужно отобразить полотнище, используя координаты из буфера.

Как это работает...

Размер рабочих групп был выбран равным `100` вызовам, по `10` в каждом измерении. Первая инструкция в вычислительном шейдере определяет число вызовов на рабочую группу.

```
layout( local_size_x = 10, local_size_y = 10 ) in;
```

Следующие далее uniform-переменные определяют константы для уравнения вычисления силы и длины горизонтальных, вертикальных и диагональных пружин в состоянии покоя. Переменная `DeltaT` определяет шаг во времени. Далее объявляются буферы координат и скоростей. Буферы координат связываются с индексами `0` и `1`, а буферы скоростей – с индексами `2` и `3`.

В функции `main` (шаг 4) сначала определяется число частиц в каждом измерении как произведение числа рабочих групп на размер рабочей группы. Затем определяется глобальный индекс обрабатываемой частицы. Так как частицы располагаются в буферах построчно, он вычисляется как произведение глобального индекса по оси Y на число частиц по оси X плюс глобальный индекс по оси X.

В шаге 5 будущая сумма сил (переменная `force`) инициализируется силой тяготения – произведением ускорения свободного падения `Gravity` на массу частицы `ParticleMass`. Обратите внимание, что эту операцию можно было бы оптимизировать, потому что все частицы имеют одинаковые массы. Мы могли бы заранее вычислить это произведение и объявить его как константу.

В шагах 6 и 7 в сумму добавляются силы воздействия восьми соседних частиц, связанных с данной частицей воображаемыми пружинами. Сила действия каждой пружины вычисляется отдельно, но перед этим необходимо убедиться, что обрабатываемая частица не находится на границе сетки. Если частица располагается на краю, у нее отсутствуют некоторые соседи (см. рис. 10.6). Например, прежде чем вычислить силу действия пружины/частицы сверху, программный код сравнивает значение `gl_GlobalInvocationID.y` с числом частиц на оси Y минус `1`. Если оно оказывается меньше, значит, соседняя частица сверху существует. В противном случае текущая частица находится на верхнем крае сетки и не имеет соседей сверху. (По сути, переменная `gl_GlobalInvocationID` хранит местоположение частицы в сетке.) Аналогичные проверки выполняются в трех других направлениях по вертикали/горизонтали. Перед вычислением силы воздействия

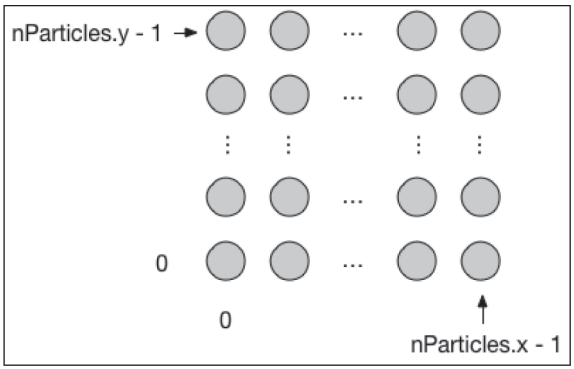

Рис. 10.6 ❖ У частиц на краю отсутствуют некоторые соседи

диагональных пружин также необходимо убедиться, что текущая частица не находится одновременно на вертикальном и горизонтальном краю. Так, в примере выше проверяется наличие частицы сверху слева. Условие выполняется, если `gl_GlobalInvocationID.x` больше нуля (то есть текущая частица не находится на левом краю) и `gl_GlobalInvocationID.y` меньше числа частиц по оси Y минус `1` (текущая частица не находится на верхнем краю).

После проверки существования соседней частицы вычисляется сила, действующая с этой стороны, и прибавляется к сумме. Частицы хранятся в буфере построчно, поэтому для доступа к координатам соседних частиц необходимо к индексу текущей частицы прибавить/вычесть число частиц по оси X, чтобы сместиться по вертикали, и/или прибавить/вычесть единицу, чтобы сместиться по горизонтали.

В шаге 8 к сумме действующих сил добавляется демпфирующая сила, имитирующая сопротивление воздуха. Она определяется как произведение величины силы `DampingConst` на вектор скорости. Знак «минус» гарантирует, что вектор силы будет направлен в сторону, противоположную вектору скорости.

В шаге 9 вычисляются новые координаты и скорость частицы методом Эйлера. Сначала вычисляется ускорение умножением суммы действующих сил на величину, обратную массе частицы, затем выполняется интегрирование методом Эйлера, и результаты сохраняются в выходные буферы.

Наконец, в шаге 10 восстанавливаются позиции пяти частиц, закрепленных «прищепками».

В функции отображения, в основной программе (шаг 11), производится многократный вызов вычислительного шейдера с переключением буферов после каждого вызова. После вызова функции `glDispatchCompute` вызывается функция `glMemoryBarrier`, чтобы дождаться окончания записи в буферы перед их сменой. По окончании цикла программа отображает полотнище с использованием позиций частиц из буфера.

И еще...

Для целей отображения хорошо также иметь векторы нормалей. Добиться желаемого можно с помощью еще одного вычислительного шейдера, который будет пересчитывать векторы нормалей после обновления координат частиц. Например, после 1000-кратного выполнения предыдущего вычислительного шейдера можно было бы организовать однократный запуск другого вычислительного шейдера для обновления нормалей, а затем уже отображать полотнище.

Кроме того, используя локальные разделяемые данные в пределах рабочих групп, можно повысить производительность. В реализации выше координаты каждой частицы читаются из буфера не менее восьми раз. Каждая такая операция чтения является довольно дорогостоящей в смысле времени выполнения. Гораздо быстрее осуществляется чтение из памяти, которая ближе к GPU. Чтобы увеличить производительность, можно прочитать данные в локальную разделяемую память один раз и впоследствии читать необходимые данные оттуда. Как это делается, я расскажу в следующем рецепте. Вы без труда сможете применить описанный там прием в этом рецепте.

См. также

- Рецепт «Определение границ с помощью вычислительного шейдера».

Определение границ с помощью вычислительного шейдера

В рецепте «Применение фильтра выделения границ» в главе 5 «Обработка изображений и приемы работы с экранным пространством» был представлен пример реализации фильтра выделения границ на основе фрагментного шейдера. Фрагментный шейдер хорошо подходит для множества операций, связанных с обработкой изображений, потому что при отображении прямоугольника, покрывающего экран, он вызывается для каждого пикселя. Так как фильтры изображений часто применяются к результатам отображения, есть возможность организовать отображение в текстуру, затем вызвать фрагментный шейдер для каждого пикселя (отображением прямоугольника, покрывающего экран), после чего внутри фрагментного шейдера обрабатывать каждый пиксель в отдельности. В каждом вызове может потребоваться читать разные пиксели из разных точек текстуры, и каждый тексель может читаться в разных вызовах шейдера.

Подобный подход с успехом можно использовать в самых разных ситуациях, но нужно понимать, что фрагментный шейдер вообще не предназначен для обработки изображений. Используя вычислительный шейдер, можно получить более полный контроль над распределением вызовов шейдера и обеспечить более высокую эффективность обработки за счет использования локальной памяти.

В данном примере представлена версия фильтра выделения границ, реализованная на основе вычислительного шейдера. В ней будет использоваться локальная (для рабочей группы) память, чтобы получить более высокую скорость обра-

ботки. Так как локальная память находится ближе к GPU, операции доступа к ней выполняются быстрее, чем операции чтения из буферов хранилищ (или текстур).

Как и в рецепте из главы 5, здесь также будет реализован оператор Собеля, состоящий из двух фильтров 3×3:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}; \quad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

За подробностями об операторе Собеля обращайтесь к главе 5 «Обработка изображений и приемы работы с экранным пространством». Ключевой особенностью данного оператора является необходимость прочитать значения восьми соседних пикселей для каждого данного пикселя. Это означает, что в процессе обработки значение каждого пикселя будет извлечено восемь раз (при обработке соседей данного пикселя). Чтобы получить дополнительную скорость, все необходимые данные будут копироваться в локальную разделяемую память рабочей группы и затем извлекаться оттуда, а не из буфера.

Операции доступа к разделяемой локальной памяти рабочей группы обычно выполняются существенно быстрее, чем операции доступа к текстуре или памяти в буфере.

В данном примере вычислительный шейдер будет вызываться один раз для каждого пикселя, а двухмерные рабочие группы будут иметь размеры 25×25. Перед вычислением оператора Собеля все необходимые пиксели будут копироваться в локальную разделяемую память рабочей группы. Для обработки фильтром каждого пикселя потребуется прочитать значения восьми соседних пикселей. Чтобы обсчитать пиксели на границе рабочей группы, нужно сохранить в локальной памяти дополнительные пиксели, обрамляющие рабочую группу. То есть для рабочей группы с размерами 25×25 нам потребуется хранилище 27×27.

Подготовка

Сначала создайте **объект буфера кадра Framebuffer Object (FBO)** с текстурой, подключенной в качестве буфера цвета, куда будет выполняться отображение исходной сцены. Создайте еще одну текстуру для сохранения результатов применения фильтра. Свяжите эту последнюю текстуру с текстурным слотом `0`. Она будет использоваться вычислительным шейдером для вывода. С помощью `glBindImageTexture` свяжите текстуру FBO со слотом текстуры изображения `0`, а вторую – со слотом текстуры изображения `1`.

Затем подготовьте вершинный и фрагментный шейдеры для отображения непосредственно в объект буфера кадра и отображения полноэкранной текстуры.

Как это делается...

Выполните следующие шаги:

1. Вычислительный шейдер, как обычно, начинается с определения числа вызовов для одной рабочей группы.

```
layout (local_size_x = 25, local_size_y = 25) in;
```

2. Далее объявляются uniform-переменные для исходного и итогового изображений, а также для хранения порогового значения фильтра. Под исходным здесь понимается изображение, помещенное в объект буфера кадра, а под итоговым – результат работы фильтра выделения границ.

```
uniform float EdgeThreshold = 0.1;
layout(binding=0, rgba8) uniform image2D InputImg;
layout(binding=1, rgba8) uniform image2D OutputImg;
```

3. Затем следует объявление разделяемой памяти рабочей группы как массива 27×27.

```
shared float
     localData[gl_WorkGroupSize.x+2][gl_WorkGroupSize.y+2];
```

4. В шейдере также определяется функция `luminance` для вычисления светимости пикселя. Так как это та же самая функция, что использовалась в нескольких рецептах в главе 5, я не буду повторять ее здесь.
5. Далее определяется функция, применяющая оператор Собеля к текущему пикселю. Она читает исходные данные непосредственно из локальной памяти.

```
void applyFilter()
{
  uvec2 p = gl_LocalInvocationID.xy + uvec2(1,1);

  float sx = localData[p.x-1][p.y-1] +
             2*localData[p.x-1][p.y] +
             localData[p.x-1][p.y+1] -
             (localData[p.x+1][p.y-1] +
              2 * localData[p.x+1][p.y] +
              localData[p.x+1][p.y+1]);
  float sy = localData[p.x-1][p.y+1] +
             2*localData[p.x][p.y+1] +
             localData[p.x+1][p.y+1] -
             (localData[p.x-1][p.y-1] +
              2 * localData[p.x][p.y-1] +
              localData[p.x+1][p.y-1]);
  float g = sx * sx + sy * sy;

  if( g > EdgeThreshold )
    imageStore(OutputImg,
      ivec2(gl_GlobalInvocationID.xy), vec4(1.0));
  else
    imageStore(OutputImg,
      ivec2(gl_GlobalInvocationID.xy), vec4(0,0,0,1));
}
```

6. В функции `main` сначала копируется светимость данного пикселя в локальную память.

```
void main()
{
  localData
    [gl_LocalInvocationID.x+1][gl_LocalInvocationID.y+1] =
    luminance(imageLoad(InputImg,
              ivec2(gl_GlobalInvocationID.xy)).rgb);
```

7. Если пиксель находится на границе рабочей группы, необходимо скопировать в локальную память один или более дополнительных пикселей, чтобы заполнить обрамление вокруг пограничных пикселей. То есть нужно проверить, находится ли пиксель на границе рабочей группы (сравнив `gl_LocalInvocationID`), и затем определить, какие пиксели следует скопировать. Это не сложно, но код получается длинным из-за необходимости убедиться, что внешний пиксель действительно существует. Например, если текущая рабочая группа находится на краю общего изображения, некоторые пиксели на ее границах не будут иметь соседей снаружи (за пределами изображения). Из-за объемности программного кода я не буду приводить его здесь – вы сможете самостоятельно исследовать его, загрузив с сайта GitHub.
8. После копирования данных нужно дождаться, пока то же самое не будет сделано другими вызовами, поэтому здесь вызывается функция `barrier`. Далее вызывается функция `applyFilter`, которая вычисляет значение фильтра и сохраняет результат в итоговое изображение.

```
  barrier();

  // Применить фильтр
  applyFilter();
}
```

9. Функция отображения в основной программе сначала отображает сцену в объект буфера кадра, затем вызывает вычислительный шейдер и ждет, пока все вызовы не сохранят результат в итоговое изображение.

```
glDispatchCompute(width/25, height/25, 1);
glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
```

10. В заключение итоговое изображение накладывается как текстура на прямоугольник, покрывающий экран.

Как это работает...

В шаге 1 определяется, что для каждой рабочей группы шейдер должен быть вызван `625` раз, по `25` в каждом измерении. В зависимости от системы, где выполняется код, это число можно изменить, выбрав то, которое лучше соответствует характеристикам аппаратного обеспечения.

Переменные типа `image2D` (шаг 2) – это исходное и итоговое изображения. Обратите внимание, как определяются привязки в квалификаторах `layout`. Они соответствуют слотам изображений в вызовах `glBindImageTexture` в основной программе. Исходное изображение должно содержать отображаемую сцену и соответствует текстуре, связанной с объектом буфера кадра. Итоговое изображение служит для сохранения результатов фильтрации. Отметьте также, что здесь указан формат `rgb8`. Он должен совпадать с форматом, использовавшимся при создании изображения вызовом `glTexStorage2D`.

В шаге 3 объявляется массив localData с квалификатором shared. Это – локальная разделяемая память рабочей группы. Размер 27×27 выбран таким, чтобы в массив можно было сохранить дополнительные обрамляющие пиксели. Здесь будут храниться значения светимости всех пикселей в рабочей группе плюс светимости обрамляющих пикселей.

Функция applyFilter (шаг 5) вычисляет оператор Собеля для данных в localData. Единственная сложность в этой функции – необходимость вычисления смещений для выборки внешних, обрамляющих пикселей. Светимость текущего пикселя, для обработки которого выполнен данный вызов, находится:

```
p = gl_LocalInvocationID.xy + uvec2(1,1);
```

Без дополнительных обрамляющих пикселей мы использовали бы просто gl_LocalInvocationID, но в данной ситуации приходится добавлять смещения на один пиксель в каждом направлении.

Следующие несколько инструкций просто вычисляют оператор Собеля, определяют величину градиента и сохраняют в g. При этом из локальной памяти localData читаются светимости восьми соседних пикселей.

В конце функции applyFilter результат применения фильтра записывается в OutputImg. Это будет либо значение (1,1,1,1), либо (0,0,0,1), в зависимости от результата сравнения g с пороговым значением. Обратите внимание, что здесь в качестве координат в итоговом изображении используется gl_GlobalInvocationID. Глобальный индекс как раз подходит на роль координат в глобальном изображении, тогда как локальный сообщает координаты в пределах рабочей группы и больше подходит для доступа к локальной разделяемой памяти.

В функции main (шаг 6) вычисляется светимость пикселя, соответствующего вызову шейдера (с индексом gl_GlobalInvocationID), и сохраняется в локальной памяти (localData), в элементе с индексом gl_LocalInvocationID + 1. И снова смещение + 1 обусловлено наличием дополнительных обрамляющих пикселей.

В следующем шаге (шаг 7) выполняется копирование обрамляющих пикселей. Это необходимо делать, только если шейдер вызван для обработки пикселя на границе рабочей группы. При этом необходимо определить, существуют ли обрамляющие пиксели. За дополнительными подробностями обращайтесь к исходному коду примера.

В шаге 8 вызывается встроенная функция barrier. Она синхронизирует все вызовы шейдера внутри рабочей группы в этой точке, гарантируя сохранение всех данных в локальной памяти. Без вызова этой функции у нас не было бы никаких гарантий, что операции записи в localData во всех вызовах шейдера завершены, и при последующей обработке могли бы получиться неверные результаты. Я думаю, будет интересно (и поучительно), если вы уберете этот вызов и исследуете получившиеся результаты.

В заключение вызывается функция applyFilter, вычисляющая оператор Собеля и записывающая результат в итоговое изображение.

В функции отображения, в основной программе, вычислительный шейдер вызывается так, чтобы охватить изображение достаточно большим числом рабочих

групп. Так как размер рабочей группы составляет 25×25, программа вызывает `width/25` рабочих групп по оси X и `height/25` – по оси Y. В результате шейдер вызывается один раз для каждого пикселя в исходном/итоговом изображении.

И еще...

Это был достаточно простой пример использования локальной памяти. Единственная сложность заключалась в необходимости применять дополнительные строки/столбцы обрамляющих пикселей. Но, вообще говоря, локальные данные можно использовать для организации любых взаимодействий между вызовами внутри рабочей группы. В данном случае локальная память использовалась не для взаимодействий, а для увеличения эффективности за счет уменьшения операций обращения к изображению.

Имейте в виду, что существуют (иногда строгие) ограничения на размеры локальной разделяемой памяти. Определить максимальный поддерживаемый объем можно с помощью семейства функций `glGetInteger*` и аргумента `GL_MAX_COMPUTE_SHARED_MEMORY_SIZE`. Спецификация OpenGL требует от реализаций, чтобы объем локальной памяти был не меньше 32 Кбайт.

См. также

- Рецепт «Применение фильтра выделения границ» в главе 5 «Обработка изображений и приемы работы с экранным пространством».

Создание фракталов с помощью вычислительного шейдера

Эту главу мы завершим примером использования вычислительного шейдера для создания изображения фрактала. Здесь будет использоваться классическое множество Мандельброта (Mandelbrot).

Множество Мандельброта основано на итерационной последовательности вычисления следующего комплексного полинома:

$$z_{n+1} = z_n^2 + c,$$

где z и c – комплексные числа. Итерации начинаются со значения $z = 0 + 0i$ и повторяются, пока не будет достигнуто максимальное допустимое число итераций или значение z не превысит заданного максимума. Если для данного значения c итерации остаются стабильными (значение z не превысило установленного максимума), считается, что точка, соответствующая числу c, принадлежит множеству Мандельброта, и соответствующая ей позиция окрашивается черным светом. В противном случае цвет от числа итераций, выполненных до того, как значение превысило максимум.

На рис. 10.7 изображено множество Мандельброта, наложенное на куб в виде текстуры.

В этом примере для вычисления множества Мандельброта будет использоваться вычислительный шейдер. Так как это еще один прием создания изображений,

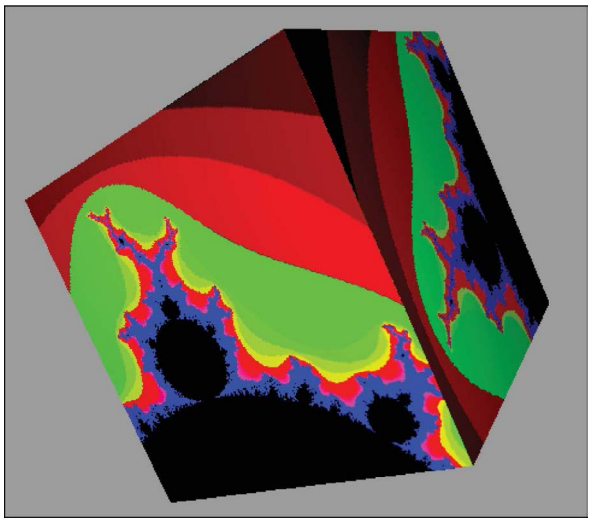

Рис. 10.7 ❖ Множество Мандельброта, наложенное на куб в виде текстуры

будет использоваться двухмерное вычислительное пространство, в котором шейдер будет вызываться один раз для каждого пикселя. Все вызовы будут работать независимо, и им не потребуется совместно использовать каких-либо данных.

Подготовка

Создайте текстуру для хранения результатов вычисления фрактала. Свяжите ее со слотом `0` текстуры изображения вызовом `glBindImageTexture`.

```
GLuint imgTex;
glGenTextures(1, &imgTex);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, imgTex);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, 256, 256);
glBindImageTexture(0, imgTex, 0, GL_FALSE, 0, GL_READ_WRITE,
                   GL_RGBA8);
```

Как это делается...

Выполните следующие шаги:

1. Вычислительный шейдер начинается с определения числа вызовов на рабочую группу.

   ```
   layout( local_size_x = 32, local_size_y = 32 ) in;
   ```

2. Далее объявите переменную для сохранения итогового изображения, а также другие uniform-переменные.

   ```
   layout( binding = 0, rgba8) uniform image2D ColorImg;
   #define MAX_ITERATIONS 100
   ```

```
uniform vec4 CompWindow;
uniform uint Width = 256;
uniform uint Height = 256;
```

3. Определите функцию для выполнения итераций в заданной точке комплексной плоскости.

```
uint mandelbrot( vec2 c ) {
  vec2 z = vec2(0.0,0.0);
  uint i = 0;
  while(i < MAX_ITERATIONS && (z.x*z.x + z.y*z.y) < 4.0) {
    z = vec2( z.x*z.x-z.y*z.y+c.x, 2 * z.x*z.y + c.y );
    i++;
  }
  return i;
}
```

4. В функции `main` сначала нужно вычислить размер пикселя в комплексном пространстве.

```
void main() {
  float dx = (CompWindow.z - CompWindow.x) / Width;
  float dy = (CompWindow.w - CompWindow.y) / Height;
```

5. Затем определить значение *c* для этого вызова.

```
  vec2 c = vec2(
    dx * gl_GlobalInvocationID.x + CompWindow.x,
    dy * gl_GlobalInvocationID.y + CompWindow.y);
```

6. Далее вызвать функцию `mandelbrot` и определить цвет по числу итераций.

```
  uint i = mandelbrot(c);
  vec4 color = vec4(0.0,0.5,0.5,1);
  if( i < MAX_ITERATIONS ) {
    if( i < 5 )
      color = vec4(float(i)/5.0,0,0,1);
    else if( i < 10 )
      color = vec4((float(i)-5.0)/5.0,1,0,1);
    else if( i < 15 )
      color = vec4(1,0,(float(i)-10.0)/5.0,1);
    else color = vec4(0,0,1,0);
  }
  else
    color = vec4(0,0,0,1);
```

7. В заключение записать цвет в итоговое изображение.

```
  imageStore(ColorImg,
    ivec2(gl_GlobalInvocationID.xy), color);
}
```

8. В функции отображения, в основной программе, запустить вычислительный шейдер и вызвать функцию `glMemoryBarrier`.

```
glDispatchCompute(256/32, 256/32, 1);
glMemoryBarrier( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
```

9. Отобразить сцену, наложив текстуру на соответствующие объекты.

Как это работает...

Объявленная в шаге 2 uniform-переменная `ColorImg` – это итоговое изображение. Согласно определению, оно связано со слотом `0` текстуры изображения (параметр `binding` в инструкции `layout`). Обратите также внимание на выбор формата `rgb8` для изображения – он должен совпадать с форматом, указанным в вызове `glTexStorage2D`, с помощью которого создается текстура.

`MAX_ITERATIONS` – это максимальное число итераций вычисления комплексного полинома, упоминавшегося выше. `CompWindow` – это область комплексного пространства, с которой мы работаем. Первые два компонента `CompWindow.xy` – действительная и мнимая части левого нижнего угла окна, а `CompWindow.zw` – правого верхнего угла. `Width` и `Height` определяют размеры изображения текстуры.

Функция `mandelbrot` (шаг 3) принимает значение `c` в виде параметра и в цикле вычисляет значение комплексного полинома, пока не будет достигнуто максимальное число итераций или пока абсолютное значение `z` не станет больше `2`. Обратите внимание, что здесь не используется операция извлечения квадратного корня, вместо этого мы просто сравниваем абсолютное значение с `4` (квадратом двойки). Функция возвращает общее число итераций.

Функция `main` (шаг 4) начинается с вычисления размера пикселя в комплексном окне (`dx`, `dy`). Это просто размер окна, деленный на число текселей в каждом измерении.

Каждый вызов вычислительного шейдера отвечает за обработку текселя с индексом `gl_GlobalInvocationID.xy`. Поэтому далее вычисляется точка на комплексной плоскости, соответствующая этому текселю. Для определения координаты X (ось действительных значений) размер текселя по оси X (`dx`) умножается на `gl_GlobalInvocationID.x` (в результате получается расстояние от левого края окна) и прибавляется координата левого края окна (`CompWindow.x`). Аналогично определяется координата Y (ось мнимых значений).

В шаге 6 вызывается функция `mandelbrot` с только что найденным значением `c`, и по числу выполненных итераций определяется цвет.

В шаге 7 вызовом `imageStore` цвет сохраняется в итоговом изображении, в пикселе `gl_GlobalInvocationID.xy`.

В функции отображения в основной программе (шаг 8) выполняется запуск вычислительного шейдера с таким числом вызовов, чтобы каждый вызов соответствовал одному текселю. Вызов функции `glMemoryBarrier` гарантирует окончание работы всех вызовов перед продолжением.

И еще...

До появления поддержки вычислительных шейдеров все это можно было бы реализовать с использованием фрагментного шейдера. Однако вычислительные шейдеры обладают большей гибкостью, позволяя нам определять, какую работу выполнять на GPU. Кроме того, с их помощью можно более эффективно использовать память, отказавшись от создания полноценного объекта буфера кадра и используя простую текстуру.

Предметный указатель

Дэвид Вольф

OpenGL 4. Язык шейдеров. Книга рецептов

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 34,5. Тираж 100 экз.

Веб-сайт издательства: www.дмк.рф