Open GL 4. Язык шейдеров. Книга рецептов

Более 70 рецептов, демонстрирующих простые и продвинутые приемы создания высококачественной трехмерной графики в реальном масштабе времени с применением OpenGL и GLSL 4.x

Дэвид Вольф

[PACKT] PUBLISHING

ДМК издательство

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

Предисловие 12

Глава 1 ❖ Введение в GLSL 18

Введение 18
Использование загрузчика функций для доступа к новейшей функциональности OpenGL 21
Использование GLM для математических вычислений 25
Использование типов GLM для передачи данных в OpenGL 26
Определение версий GLSL и OpenGL 27
Компиляция шейдера 29
Компоновка шейдерной программы 33
Передача данных в шейдер с использованием вершинных атрибутов и вершинных буферных объектов 36
Получение списка активных атрибутов и их индексов 45
Передача данных в шейдер с использованием uniform-переменных 48
Получение списка активных uniform-переменных 51
Использование uniform-блоков и uniform-буферов 53
Получение отладочных сообщений 59
Создание класса C++, представляющего шейдерную программу 62
Рассеянное отражение с единственным точечным источником света 68

Глава 2 ❖ Основы шейдеров GLSL 66

Введение 66
Фоновый, рассеянный и отраженный свет 73
Использование функций в шейдерах 80
Реализация двустороннего отображения 83
Реализация модели плоского затенения 87
Использование подпрограмм для выбора функциональности в шейдере 89
Отбрасывание фрагментов для получения эффекта решетчатой поверхности 94

Глава 3 ❖ Освещение, затенение и оптимизация 98

Введение 98
Освещение несколькими точечными источниками света 98
Освещение источником направленного света 101
Пофрагментное вычисление освещенности для повышения реализма 104
Использование вектора полупути для повышения производительности 107
Имитация узконаправленных источников света 110
Придание изображению «мультяшного» вида 113
Имитация тумана 116
Настройка проверки глубины 119

Глава 4 ❖ Текстуры 122

Введение 122
Наложение двухмерной текстуры 123
Наложение нескольких текстур 128

Использование карт прозрачности для удаления пикселей ... 131
Использование карт нормалей ... 134
Имитация отражения с помощью кубической текстуры ... 140
Имитация преломления с помощью кубической текстуры ... 147
Наложение проецируемой текстуры ... 152
Отображение в текстуру ... 157
Использование объектов-семплеров ... 162

Глава 5 ❖ Обработка изображений и приемы работы с экранным пространством ... 165

Введение ... 165
Применение фильтра выделения границ ... 166
Применение фильтра размытия по Гауссу ... 172
Преобразование диапазона яркостей HDR с помощью тональной компрессии ... 179
Эффект размытости на границах ярких участков ... 184
Повышение качества изображения с помощью гамма-коррекции ... 189
Сглаживание множественной выборкой ... 192
Отложенное освещение и затенение ... 197
Реализация порядконезависимой прозрачности ... 203

Глава 6 ❖ Использование геометрических шейдеров и шейдеров тесселяции ... 215

Введение ... 215
Отображение точечных спрайтов с помощью геометрического шейдера ... 220
Наложение каркаса на освещенную поверхность ... 225
Рисование линий силуэта с помощью геометрического шейдера ... 233
Тесселяция кривой ... 242
Тесселяция двухмерного прямоугольника ... 247
Тесселяция трехмерной поверхности ... 252
Тесселяция с учетом глубины ... 257

Глава 7 ❖ Тени ... 261

Введение ... 261
Отображение теней с помощью карты теней ... 261
Сглаживание границ теней методом PCF ... 272
Смягчение границ теней методом случайной выборки ... 275
Создание теней с использованием приема теневых объемов и геометрического шейдера ... 282

Глава 8 ❖ Использование шума в шейдерах ... 291

Введение ... 291
Создание текстуры шума с использованием GLM ... 293
Создание бесшовной текстуры шума ... 296
Создание эффекта облаков ... 298
Создание эффекта текстуры древесины ... 300
Создание эффекта разрушения ... 303
Создание эффекта брызг краски ... 305

Создание эффекта изображения в приборе ночного видения 308

Глава 9 ❖ Системы частиц и анимация 312

Введение 312
Анимация поверхности смещением вершин 313
Создание фонтана частиц 316
Создание системы частиц с использованием трансформации с обратной связью 322
Создание системы частиц клонированием 331
Имитация пламени с помощью частиц 334
Имитация дыма с помощью частиц 337

Глава 10 ❖ Вычислительные шейдеры 340

Введение 340
Реализация системы частиц с помощью вычислительного шейдера 344
Имитация полотнища ткани с помощью вычислительного шейдера 348
Определение границ с помощью вычислительного шейдера 355
Создание фракталов с помощью вычислительного шейдера 360

Предметный указатель 364

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 викистранице 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.

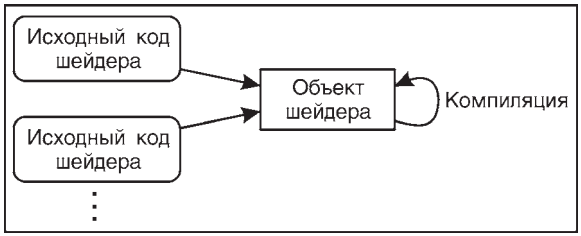

Рис. 1.1 ❖ Процесс компиляции шейдера

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».