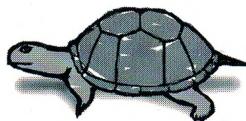


АЛЕКСЕЙ БОРЕСКОВ

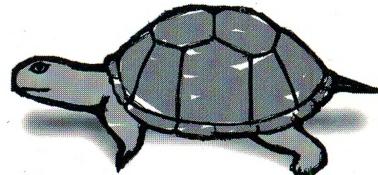


РАЗРАБОТКА И ОТЛАДКА ШЕЙДЕРОВ

Вершинные и фрагментные шейдеры



Шумовая функция в моделировании



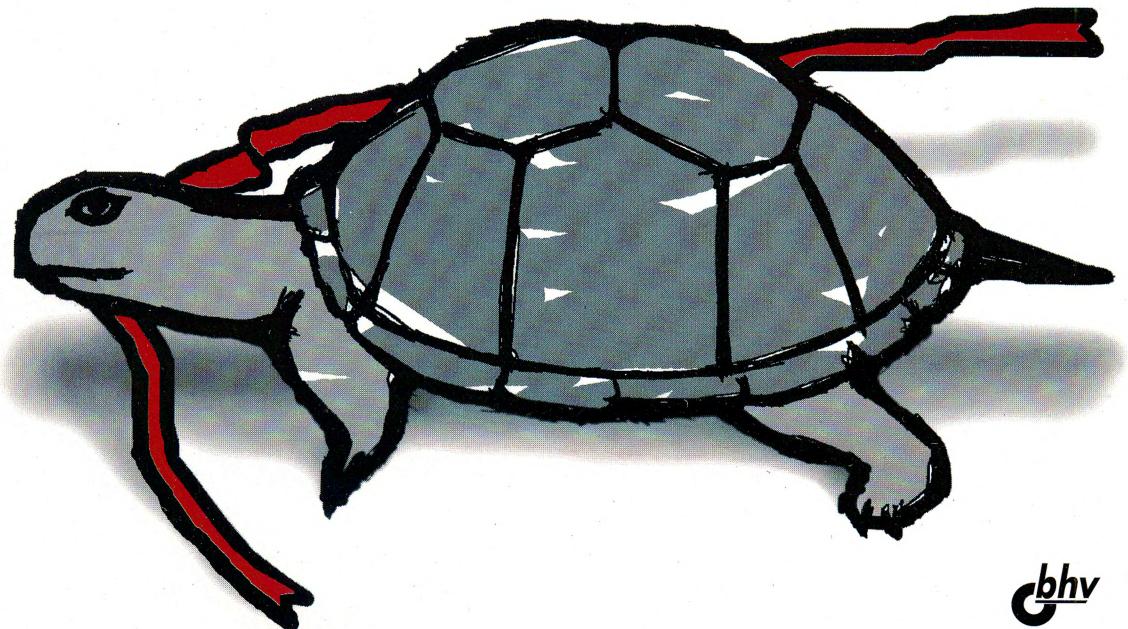
Полигональные модели

Реализация р-буфера и фреймбуфера

Рендеринг в текстуру

Основные модели освещения

Библиотеки libCamera, libTexture и libTexture3D



Алексей Боресков

РАЗРАБОТКА И ОТЛАДКА ШЕЙДЕРОВ

Санкт-Петербург

«БХВ-Петербург»

2006

УДК 681.3.06
ББК 32.973.26-018.1
Б82

Боресков А. В.
Б82 Разработка и отладка шейдеров. — СПб.: БХВ-Петербург, 2006. —
496 с.: ил.
ISBN 5-94157-712-5

Практическое пособие по разработке кросс-платформенных шейдеров на языке OpenGL Shader Language (GLSL) в среде RenderMonkey для использования в операционных системах Windows и Linux с различными версиями библиотеки OpenGL. Рассматривается написание и отладка вершинных и фрагментных шейдеров, использование шумовой функции в моделировании и основных моделей освещения, моделирование преломления и дифракции, обработка изображений на GLSL, работа с полигональными моделями, практическое применение GLSL в программах на C++ и работа с библиотеками libCamera, libTexture и libTexture3D.

*Для специалистов в области компьютерной графики,
а также студентов и аспирантов*

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зав. редакцией	<i>Григорий Добин</i>
Редактор	<i>Татьяна Темкина</i>
Компьютерная верстка	<i>Натальи Караваевой</i>
Корректор	<i>Виктория Пиотровская</i>
Дизайн обложки	<i>Инны Тачиной</i>
Зав. производством	<i>Николай Тверских</i>

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 20.01.06.
Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 39,99.
Тираж 2000 экз. Заказ № 48
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.
Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-712-5

© Боресков А. В., 2006
© Оформление, издательство "БХВ-Петербург", 2006

Оглавление

Введение.....	1
ЧАСТЬ I. ЭВОЛЮЦИЯ OPENGL.....	7
Глава 1. Развитие OpenGL	9
Версия 1.1	11
Версия 1.2	13
Версия 1.3	15
Версия 1.4	17
Версия 1.5	19
Сводка изменений в версиях OpenGL.....	20
Глава 2. Расширения ARB_vertex_program и ARB_fragment_program.	
Появление Cg	23
Глава 3. Появление OpenGL 2.0 и GLSL	67
ЧАСТЬ II. ДОПОЛНИТЕЛЬНЫЕ БИБЛИОТЕКИ	73
Глава 4. Библиотека GLUT.....	75
Инициализация.....	76
Начало цикла обработки сообщений	78
Работа с окнами.....	79
Работа с меню	82
Установка обработчиков сообщений	84
Получение информации о текущем состоянии GLUT	88
Вывод текста	91
Вывод простейших объектов.....	92
Сфера.....	92
Куб	92
Конус.....	93

Тор	93
Додекаэдр	94
Октаэдр	94
Тетраэдр	94
Икосаэдр	95
"Чайник"	95
Глава 5. Общие сведения о библиотеках GLH, NV_MATH и NV_UTIL.....	96
Библиотека GLH	96
Инкапсуляция основных математических классов.....	96
Объектная надстройка над GLUT	105
Объектная надстройка над OpenGL	109
Библиотека NV_MATH.....	114
Библиотека NV_UTIL.....	119
Чтение текстур в формате TGA	120
Чтение текстур в формате JPG.....	121
Чтение данных из ZIP-архивов.....	122
Чтение моделей в формате ASE	123
Глава 6. Основные классы для работы с векторами, матрицами и кватернионами	125
Работа с векторами.....	125
Работа с матрицами.....	135
Работа с кватернионами	137
Глава 7. Работа с расширениями, библиотека libExt.....	139
Глава 8. Библиотеки libTexture и libTexture3D	142
Глава 9. Реализация <i>p</i>-буфера в виде класса, использование расширения EXT_framebuffer_object для создания внеэкранных буферов.....	149
Работа с <i>p</i> -буфером в Microsoft Windows.....	151
Непосредственное создание <i>p</i> -буфера	151
Выбор <i>p</i> -буфера как текущей цели для рендеринга	154
Уничтожение <i>p</i> -буфера	154
Обработка переключения видеорежима	154
Копирование данных из <i>p</i> -буфера в текстуру	155
Связывание <i>p</i> -буфера с текстурой	155
Реализация <i>p</i> -буфера для платформы Windows.....	157
Работа с <i>p</i> -буфером в Linux.....	160
Использование расширения EXT_framebuffer_object	165

Глава 10. Понятие камеры. Работа с библиотекой libCamera.....	179
Глава 11. Библиотека libMesh. Работа с полигональными моделями и их загрузка из популярных форматов.....	204
ЧАСТЬ III. ШЕЙДЕРЫ	241
Глава 12. Введение в GLSL. Описание синтаксиса, простые примеры.....	243
Основные типы данных и переменных	243
Структуры	244
Массивы.....	245
Переменные.....	245
Операторы и выражения	248
Конструкторы.....	250
Работа с компонентами векторов и матриц	251
Работа со структурами.....	253
Основные операции над векторами и матрицами	253
Основные операторы и конструкции.....	255
Стандартные переменные, атрибуты и константы.....	258
Специальные переменные для вершинных шейдеров	258
Специальные переменные для фрагментных шейдеров	259
Стандартные константы.....	260
Стандартные атрибуты для вершинного шейдера	261
Стандартные <i>uniform</i> -переменные — параметры состояния OpenGL.....	261
Стандартные <i>varying</i> -переменные.....	265
Стандартные функции	265
Тригонометрические функции и функции для работы с углами.....	266
Экспоненциальные функции (возведение в степень, нахождение логарифмов).....	266
Функции общего назначения	267
Геометрические функции	269
Матричные функции	270
Функции для сравнения векторов	270
Функции для доступа к текстурам	271
Функции для работы с производными.....	273
Шумовые функции	274
Простейший пример использования вершинных и фрагментных шейдеров	275

Глава 13. Простейшие вершинные и фрагментные шейдеры.	
Реализация основных моделей освещения.....	276
Диффузная модель освещения.....	277
Бликовое освещение по Блинну.....	280
Бликовое освещение по Фонгу.....	283
Использование карт нормалей.....	287
Анизотропные модели освещения.....	291
Подсветка края	297
Модель освещения Гуч	299
Глава 14. Практическое использование GLSL в программах на C++, необходимые расширения.....	302
Необходимые расширения	302
Расширение GL_ARB_shading_language_100	302
Расширение GL_ARB_shader_objects.....	303
Расширение GL_ARB_vertex_shader.....	308
Расширение GL_ARB_fragment_shader	310
Получение информации о поддержке GLSL	311
Простейшая программа, использующая GLSL.....	315
"Заворачиваем" шейдеры на GLSL в класс	324
Глава 15. Разработка шейдеров на GLSL в интегрированной среде RenderMonkey	347
Настройка RenderMonkey	351
Создание нового проекта	352
Глава 16. Использование основных моделей освещения, моделирование преломления и дифракции, обработка изображений на GLSL	375
Использование шейдеров из главы 13.....	375
Использование шейдеров для анимации.....	386
Моделирование преломления	394
Моделирование дифракции света	399
Обработка изображений средствами GLSL.....	412
Фильтр Sepia.....	413
Гамма-коррекция	419
Фильтр выделения границ	420
Коррекция цвета в пространстве HSV	422
Размытие изображения	425

Глава 17. Использование шумовой функции в моделировании.	
Моделирование облаков, волн и материалов.....	427
Использование шумовой функции для рендеринга "мыльных пузырей"	432
Искажение нормали при помощи шумовой функции.....	440
Турбулентность	443
Ржавление.....	450
Рябь на воде	459
Облака.....	460
Мрамор	463
Дерево	469
Система "сверкающих" частиц.....	472
Огонь.....	474
Приложение. Описание компакт-диска	477
Список литературы и интернет-ресурсов	479
Предметный указатель	481

Введение

Вся современная компьютерная графика, будь то игры, системы научной визуализации или системы для создания спецэффектов, просто немыслима без использования шейдеров (shader).

Изначально шейдером называли программу, определяющую цвет пикселов изображения. Однако для современных графических процессоров (GPU, Graphics Processing Unit) шейдер — скорее, просто некоторая программа, выполняемая непосредственно графическим процессором и не обязательно связанная с вычислением цвета пикселов.

Огромные возможности современных GPU дают возможность переложить на эти устройства целый ряд задач (например, таких как анимация объектов), разгрузив за счет этого центральный процессор.

Область применения шейдеров огромна и продолжает постоянно расширяться. Например, есть направление GPGPU (Generic Programming on GPU), посвященное неграфическим задачам (в том числе, решению нелинейных дифференциальных уравнений) средствами GPU.

Однако написание шейдеров — программ, выполняемых на GPU, — сильно отличается от программирования для традиционных процессоров. Дело в том, что GPU — это процессор с весьма специфической архитектурой, ориентированной на задачи рендеринга (т. е. визуализации) трехмерных объектов. Подобная специализация позволяет обеспечить огромное быстродействие графического процессора, но при этом накладывает определенные ограничения на выполняемые на нем программы.

Центральным понятием для современного GPU является так называемый *программируемый графический конвейер* (programmable graphics pipeline), рис. В1.

Как видно из этого рисунка, GPU работает в ответ на поступающие на его вход данные (так называемая модель потока данных, dataflow). В качестве этих данных обычно выступают значения различных параметров вершин. Эти данные проходят целый ряд преобразований, приводящих к изменению содержимого *фреймбуфера*, т. е. набора пикселов.

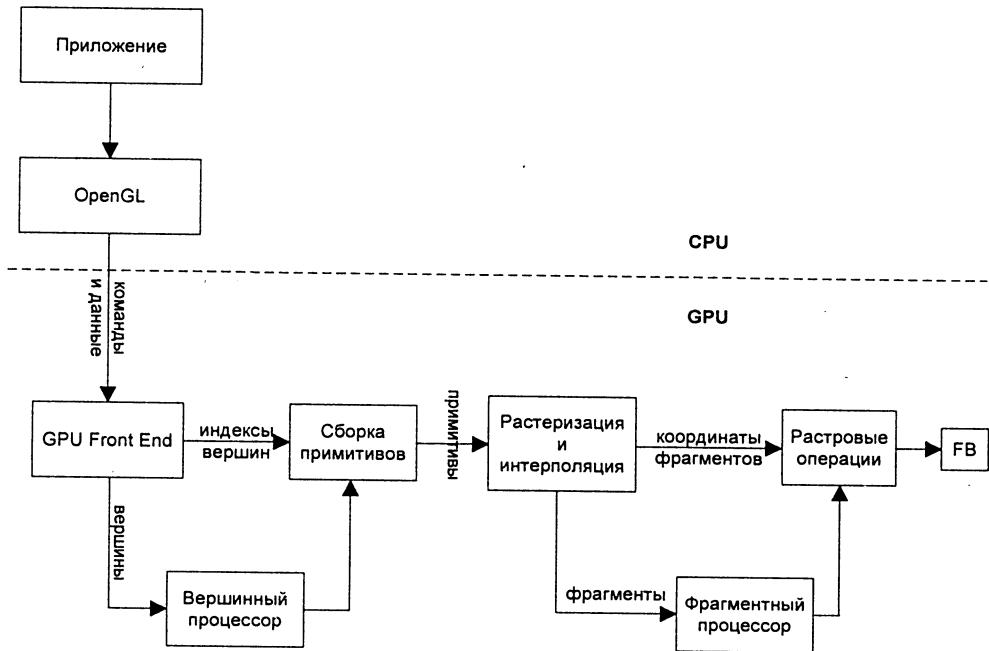


Рис. В1. Программируемый графический конвейер

Таким образом, GPU работает как с данными вершин, так и с отдельными фрагментами изображения. Эти два сильно различающихся типа данных определяют два типа шейдеров — вершинные и фрагментные.

Вершинный шейдер (vertex shader), или *вершинная программа* (vertex program) — это выполняемая графическим процессором программа, обрабатывающая данные вершин геометрических примитивов. Вершинная программа не обладает доступом к топологической информации (такой как организация отдельных вершин в геометрические примитивы). Вершинная программа не может изменять существующую топологию (например, добавлять новые вершины или удалять существующие). Также вершинная программа не имеет доступа к данным других вершин. Последнее связано с тем, что обычно вершины обрабатываются параллельно (и, следовательно, независимо друг от друга).

Фрагментный шейдер (fragment shader), или *фрагментная программа* (fragment program) — это выполняемая графическим процессором программа обработки отдельных фрагментов, получаемых при растеризации геометрических примитивов. Фрагментные программы также выполняются параллельно для нескольких фрагментов.

И на фрагментные и на вершинные программы, имеющиеся GPU накладывают ряд ограничений, не свойственных программам для традиционных процессоров. Эти ограничения связаны как с типом программы (вершинная или фрагментная), так и с типом конкретного используемого графического

процессора. С появлением все более мощных графических процессоров этих ограничений постепенно становится меньше.

Первые шейдеры писались на специальных языках, очень похожих на ассемблер (например, языки, вводимые расширениями `GL_ARB_vertex_program` и `GL_ARB_fragment_program` [1]), причем поддержка фрагментных программ появилась заметно позднее поддержки вершинных программ.

Писать шейдеры на подобных низкоуровневых языках было сложно. Рост возможностей и быстродействия графических процессоров позволил использовать для написания шейдеров специальные языки высокого уровня — *шейдерные языки*. К высокоуровневым языкам написания программ с использованием библиотеки OpenGL относятся язык Cg (C for graphics), разработанный компанией NVIDIA, и OpenGL Shading Language (GLSL), ставший стандартом для программирования шейдеров с помощью библиотеки OpenGL 2.0.

Шейдерные языки достаточно просты и удобны в применении со всеми версиями библиотеки OpenGL, даже с версией 1.1, с которой многие (большинство пользователей Microsoft Windows) работают до сих пор. Для поддержки этих языков нужны последние версии драйверов, а для языка Cg требуется также наличие Cg SDK или Cg runtime.

Кроме шейдерных языков, важную роль в программировании графики играют различные библиотеки, облегчающие написание приложений с использованием OpenGL. Подобные библиотеки позволяют заметно упростить работу как с применяемым математическим аппаратом, так и собственно с OpenGL, обеспечить удобный доступ к различным форматам ресурсов (текстурам, полигональным моделям и т. п.), а также к возможностям самого GPU (таким как *p*-буферы, задаваемые пользователем фреймбуферы и т. д.).

Данная книга пытается охватить все эти аспекты использования графической библиотеки OpenGL, соответственно она разбита на три части.

Часть I посвящена истории и эволюции библиотеки OpenGL — стандарта программирования трехмерной графики во всем мире. В этой части рассматриваются все основные версии OpenGL, включая версию 2.0, и приводится информация об особенностях этих версий. В конце *главы 1* приведена таблица всех расширений OpenGL, вошедших в состав OpenGL, с указанием номера версии, начиная с которой данные расширения становились полноправной частью OpenGL.

В *главе 2* рассматривается написание шейдеров на языке ассемблера (через расширения `ARB_vertex_program` и `ARB_fragment_program`) и приводятся примеры шейдеров на языке Cg. Для всех этих типов шейдеров приведены их инкапсуляции в классы языка C++, облегчающие их использование.

Часть II посвящена различным библиотекам, облегчающим написание приложений с использованием OpenGL. Непосредственно сами шейдеры в этой части не рассматриваются (они как бы ортогональны этой части).

Классическими библиотеками, часто используемыми с OpenGL, являются библиотека GLUT, рассмотренная в главе 4, и различные библиотеки, входящие в состав NVIDIA SDK, рассмотренные в главе 5.

В главе 6 рассматриваются созданные мной библиотеки, инкапсулирующие работу с такими математическими понятиями, как векторы, матрицы и кватернионы. Все эти понятия реализуются в виде классов языка C++ и непосредственно пригодны к использованию.

В главах 7 и 8 рассматриваются разработанные автором библиотеки libExt и libTexture. Первая обеспечивает простой доступ к основным расширениям OpenGL (что используется в дальнейшем при работе с шейдерами), а вторая — к текстурам различных форматов (BMP, TGA, PNG, JPG и DDS). Плюсами этих библиотек являются небольшой размер и простота их использования.

В главе 9 посвящена рендерингу в текстуру, необходимым для этого расширениям и практической работе с ними. В ней приводятся кроссплатформенные реализации *p*-буферов и задаваемых пользователем фреймбуферов, используемые в дальнейшем при написании шейдеров, осуществляющих постобработку изображений.

В главе 10 вводятся два довольно простых, но полезных класса — Camera и Frustum. Первый является абстракцией камеры (наблюдателя) в пространстве, второй — абстракцией усеченной пирамиды видимости, используемой в OpenGL.

Глава 11 посвящена использованию полигональных моделей. В этой главе не только вводятся необходимые классы, обеспечивающие эффективное хранение и вывод сложных трехмерных моделей, но и рассматривается ряд классов, обеспечивающих загрузку готовых моделей из ряда популярных форматов, таких как 3DS, ASE, OBJ, LWO, MD3 и MD5. Непосредственное описание этих форматов и загрузки моделей из них, к сожалению, не вошло в книгу из-за ограничений по размеру (большинство из них достаточно сложны и подробное описание загрузки моделей из них заняло бы всю книгу). Однако на прилагаемом к книге компакт-диске находится полный исходный код для всех этих классов, который вы легко можете использовать в своих программах.

Часть III книги посвящена самим шейдерам и языку GLSL. Здесь подробно описан язык GLSL, его стандартные функции и переменные, а также расширения, необходимые для использования в ваших программах шейдеров, написанных на этом языке.

Специфика вершинных и фрагментных программ очень сильно влияет на процесс их написания и отладки. Часто на точный подбор различных параметров и методов расчета уходит много времени. Использование специальных сред позволяет в разы (если не на порядки) сократить траты времени и сил, необходимых для написания и отладки шейдеров.

Глава 15 полностью посвящена одной из таких сред — программе Render-Monkey компании ATI. В этой главе подробно изложены основные возможности среды и на примерах показано ее использование для написания и отладки сложных, многопроходных шейдеров.

Глава 16 посвящена реализации ряда "классических" эффектов, моделированию преломления и дифракции, созданию систем частиц. В этой главе вы найдете как известные модели освещения — Фонга, Блинна, так и гораздо менее известные модели Миннеарта, анизотропного освещения, Уорда. Также в этой главе рассматривается применение шейдеров для создания пост-обработки изображений средствами графического ускорителя.

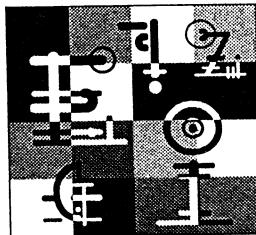
Глава 17 посвящена шейдерам, использующим так называемую шумовую функцию Перлина (Perlin noise) для моделирования ряда природных явлений и материалов. В этой главе на примерах показано, что с помощью шумовой функции (и производных от нее функций) можно имитировать материалы, встречающиеся в природе, и природные явления. Характерной чертой такого подхода является огромная гибкость — в некоторых программах вы можете прямо во время их выполнения настраивать параметры, кардинально изменяя вид материала. В *главе 17* рассмотрено моделирование неба с анимированным облачным покровом, плотность которого можно изменять прямо во время выполнения программы. Еще один интересный пример применения шумовой функции, представленный в этой главе, — моделирование я比 на воде.

В *Приложении* описан сопроводительный компакт-диск книги, который содержит используемые в книге авторские библиотеки (исходный код) и примеры программ (исходный код и исполняемые модули для платформ Windows и Linux) в папках, соответствующих главам книги. Также на этом диске размещены используемые в примерах текстуры и трехмерные модели в различных форматах.

В конце книги приведен *Список литературы и интернет-ресурсов*. На русском языке опубликовано довольно мало книг по шейдерам, поэтому может оказаться очень полезным перечень интернет-ресурсов.

Предметный указатель помогает быстро отыскать в книге нужную тему.

Исправления и дополнения к материалам книги вы можете найти на сайте автора (www.steps3d.narod.ru). Если в ходе чтения книги у вас появились замечания, пожелания и вопросы, вы можете обратиться к автору по электронному адресу steps3d@narod.ru.



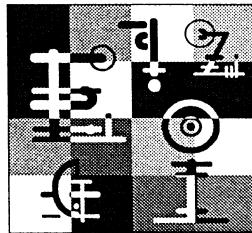
Часть I

Эволюция OpenGL

Глава 1. Развитие OpenGL

**Глава 2. Расширения ARB_vertex_program
и ARB_fragment_program.
Появление Cg**

Глава 3. Появление OpenGL 2.0 и GLSL



Глава 1

Развитие OpenGL

Появившаяся в июле 1992 года графическая библиотека OpenGL по-прежнему остается стандартом в профессиональной трехмерной графике (сейчас OpenGL зачастую уже используется и для обработки двухмерной графики и видео).

Более того, OpenGL также используется в целом ряде игр. Например, все игры, основанные на технологиях от компании idSoftware (начиная с Quake), используют OpenGL. Популярная игра Serious Sam также была написана с использованием OpenGL, возможность использования графической библиотеки Direct3D от компании Microsoft появилась в ней позже.

Явными преимуществами библиотеки OpenGL являются ее простота (в чем библиотека Direct3D сильно отстает), расширяемость и спокойное эволюционное развитие (здесь Direct3D находится практически на нуле). Кроме того, библиотека OpenGL является кроссплатформенной, что позволяет применять ее на самых различных аппаратных и программных платформах.

Пожалуй, наиболее важной чертой, сыгравшей огромную роль в развитии OpenGL, является изначально заложенный механизм расширений [1], позволяющий производителям графических ускорителей самостоятельно добавлять к OpenGL новые возможности, поддерживаемые их GPU. Тем самым новые возможности практически сразу же (как только будет написан соответствующий драйвер) становятся доступными разработчикам программ.

Каждая следующая версия OpenGL отличается от предыдущей только списком расширений, вошедших в саму библиотеку. Из этого следует, что любая программа, написанная для одной версии OpenGL, будет успешно работать и на всех следующих версиях.

У каждого расширения есть имя (например, GL_ARB_fragment_program), состоящее из *префикса платформы* (GL соответствует платформо-независимым расширениям, WGL — расширениям для платформы Windows, GLX — расширениям для платформы X Window System, используемой в различных версиях операционной системы Unix), *префикса производителя* (разработчика)

и собственно *имени расширения*. Подобный способ именования позволяет не путать расширения от различных производителей (табл. 1.1).

Таблица 1.1. Префиксы производителей для основных типов расширений OpenGL

Префикс	Производитель расширения
ARB	OpenGL Architecture Review Board.
EXT	Несколько производителей, разработавших расширение совместно
3DFX	3DFX
APPLE	Apple Inc.
ATI	ATI Technologies Inc.
HP	Hewlett-Packard Co.
IBM	Internation Buisness Machines Inc.
INTEL	Intel Corp.
KTX	Kinetix
NV	NVIDIA
MESA	Расширения, введенные в реализации Mesa библиотеки OpenGL
SGI, SGIX, SGIS	Silicon Graphics Inc.
SUN	Sun Microsystems
WIN	Microsoft Corp.

Кроме имени, каждое расширение вводит свои константы и функции (в их именовании поддерживается принятый в OpenGL стиль, при этом в конец имени также добавляется префикс платформы, например `glActiveTextureARB`).

Непосредственно на стадии выполнения программа может получить список всех поддерживаемых расширений и начать использовать требуемые возможности при наличии их поддержки. Этот механизм сыграл огромную роль в развитии OpenGL — фактически каждая следующая версия OpenGL отличается от предыдущей только тем, что какие-то расширения стали обязательными и вводимые ими функции и константы вошли в состав самой библиотеки OpenGL. В силу такого подхода сохраняется полная совместимость старых версий OpenGL по отношению к новым — можно взять любую программу, написанную с применением более ранней версии OpenGL (даже написанную под другую платформу), и она будет успешно компилироваться и работать с новой версией.

Таким образом, развитие OpenGL идет эволюционным путем, сохраняя совместимость с предыдущими версиями. В частности подобное развитие

гарантирует, что время и силы, затраченные на изучение одной версии OpenGL, не пропадут (или обесценятся) с выходом новой версии (в отличие от Direct3D). Более того, может оказаться, что вы уже давно задействуете возможности новой версии, только через механизм расширений. Неслучайно все игры компании idSoftware (от GLQuake до Doom 3), работающие с графическими ускорителями, используют именно библиотеку OpenGL.

В этой главе мы рассмотрим существующие на данный момент версии OpenGL и их отличительные особенности.

Версия 1.1

Версия 1.1 была первой версией библиотеки OpenGL, вышедшей после начальной версии 1.0. В нее вошли многие важные дополнения, заметно расширявшие возможности OpenGL.

Одной из главных возможностей, добавленных в OpenGL 1.1, явилась поддержка так называемых *вершинных массивов* (*vertex arrays*). Их использование позволило передавать данные OpenGL большими блоками, существенно сократив количество необходимых для этого вызовов OpenGL. Это усовершенствование также позволило заметно повысить эффективность передачи данных, например, за счет использования аппаратно ускоренного прямого доступа к памяти (DMA, Direct Memory Access). В основу механизма вершинных массивов было положено расширение `EXT_vertex_array` с небольшими изменениями.

Еще одним важным добавлением стало использование так называемого *смещения* (*polygon offset*) для изменения глубин фрагментов, получающихся при растеризации примитивов. Необходимость в подобном смещении часто возникает при выводе сразу нескольких граней, лежащих в одной плоскости. В подобном случае даже небольшие погрешности в вычислении глубины (практически неизбежные из-за ограниченной точности вычисления) могут повлиять на видимость фрагмента.

Добавление к вычисленному значению глубины дополнительного смещения (заведомо превосходящего погрешность вычисления) позволяет гарантировать корректное определение видимости [18]. Это добавление реализуется с помощью расширения `EXT_polygon_offset`.

Расширение `EXT_blend_logic_op` легло в основу еще одного добавления — использования логических операций при выводе фрагментов в фреймбуфер для RGBA-режимов (а не только для режима цветовых индексов, как в OpenGL 1.0).

Также в версии 1.1 было введено понятие внутренних форматов текстур (вместо используемого ранее числа компонент). Внутренний формат задает как количество бит на компоненту, так и организацию данных. Возможность

явного задания внутренних форматов позволяет управлять как точностью, так и отводимым под текстуру объемом видеопамяти. Измененная команда `glTexImage2D` (аналогичным образом изменяются другие команды):

```
void glTexImage2D( GLenum target, int level,  
                    int internalFormat, GLsizei width, GLsizei height,  
                    int border, GLenum format, GLenum type, void * data );
```

Сохранилась возможность в качестве значения переменной `internalFormat` задавать количество компонент, но также можно передавать специальные форматы, такие как `GL_RGBA8` и др. Данную возможность предоставляет расширение `EXT_texture`.

Расширение `EXT_texture` также позволяет явно задавать способ замены цвета фрагмента в текстуре.

Еще одной возможностью, предоставленной расширением `EXT_texture`, является введение так называемых прокси-текстур (`texture proxies`), позволяющих получать информацию о максимальном возможном размере текстуры в зависимости от других параметров текстуры (например, от ее формата) без явного создания текстуры. Для этого в команды `glTexImage` в качестве типа текстуры передается одна из констант `GL_PROXY_TEXTURE_1D`, `GL_PROXY_TEXTURE_2D` и т. д.

Также в версии 1.1 появилась возможность копирования прямоугольных блоков из обычной памяти или фреймбуфера непосредственно в текстуру. Основой для данной возможности стали расширения `EXT_copy_texture` и `EXT_subtexture`.

Еще одним важным улучшением стало трактование каждой текстуры (со всеми промежуточными уровнями) и связанного с ней состояния как некоторого объекта. Данную возможность предоставило расширение `EXT_texture_object`.

Также в версии 1.1 появились отдельные небольшие новшества:

- стало возможным задавать индексы цвета как беззнаковые байты;
- при растеризации точек и битовых изображений происходит деление текстурных координат (s, t, r, q) на компоненту q . Ранее эта возможность была документирована только для отрезков и многоугольников;
- изменен алгоритм растеризации отрезков для обеспечения корректной растеризации вертикальных отрезков на границе между пикселами;
- для текстур без альфа-канала при запросах значения альфа было документировано возвращение единицы, что не было указано в версии 1.0;
- для величин, задающих начало и конец тумана (`fogStart` и `fogEnd`) стало возможным задавать отрицательные значения.

Версия 1.2

Версия 1.2 библиотеки OpenGL, вышедшая 16 марта 1998 года, включила в себя целый ряд новых возможностей по сравнению с версиями 1.0 и 1.1 (оставаясь при этом полностью совместимыми с ними).

Одной из наиболее важных возможностей, добавленных в OpenGL 1.2, стала поддержка трехмерных текстур, основанная на расширении `EXT_texture3D`.

Еще одной возможностью, особенно удобной для пользователей Microsoft Windows, стала поддержка нового формата `BGRA` (совпадающего по структуре пикселов с форматом `DIB`) (`Device Independent Bitmap`). Эта возможность реализована в расширении `EXT_bgra` [1].

Также в этой версии появилась поддержка упакованных (`packed`) форматов пикселов, когда тип значения для всего пикселя является законным для компьютера (байт, слово, двойное слово), но типы его отдельных компонентов могут не являться законными. Эта возможность реализована в расширении `EXT_packed_pixels`.

Также появилась возможность автоматического масштабирования вектора нормали при помощи коэффициентов, получаемых из матрицы модельного преобразования. Во многих случаях подобное масштабирование работает заметно быстрее явного нормирования нормалей. Эта возможность реализована в расширении `EXT_rescale_normal`.

Также в эту версию вошла поддержка дополнительного цвета (`secondary color`) [1], используемого при расчете освещенности и не изменяющегося в ходе операции текстурирования (в отличие от основного цвета). Данная возможность облегчила создание бликов на поверхности, цвет которых зависит только от цвета источника света. Эта возможность реализована в расширении `EXT_separate_specular_color`.

Еще одной удобной возможностью стало введение нового режима отсечения текстурных координат — `GL_CLAMP_TO_EDGE` [1], гарантирующего, что при использовании пирамидального фильтрования значения цвета пикселов на границе текстуры не влияют на значения внутренних пикселов промежуточных mipmap-уровней. Эта возможность реализована в расширении `SGIS_texture_edge_clamp`.

Расширение `SGIS_texture_lod` принесло этой версии OpenGL сразу две возможности, связанные с пирамидальным фильтрованием и позволяющие экономить видеопамять для текстур большого размера:

- задавать диапазон, в пределах которого будет изменяться параметр λ , отвечающий за выбор уровня в пирамидальном фильтровании;
- загружать для текстур не все уровни в пирамидальном фильтровании, а только некоторое их подмножество.

При работе с вершинными массивами теперь можно отрисовывать не все заданные массивами грани, а только некоторую их часть. Эта возможность реализована в расширении `EXT_draw_range_element`.

Также в версию 1.2 вошел целый ряд возможностей, связанных с обработкой изображений.

Появилась поддержка одно- и двухмерной свертки (*convolution*) в процессе переноса пикселов. Сами ядра свертки трактуются как одномерные и двухмерные текстуры, которые могут загружаться как из оперативной памяти, так и из фреймбуфера. Эта возможность реализована с помощью расширений `EXT_convolution` и `HP_convolution_border_modes`.

Одной из новых возможностей стало использование *lookup*-таблиц для RGBA-режима. При том сами эти таблицы трактуются скорее как одномерные текстуры со своими форматами и фильтрами свертки. К таким таблицам может происходить до трех обращений — до свертки, после свертки, но до преобразования цвета при помощи матрицы, и после преобразования цвета матрицей. Эта возможность реализована с помощью расширений `EXT_color_table` и `EXT_color_subtable`.

Также была добавлена поддержка еще одного матричного преобразования и стека соответствующих этому преобразованию матриц (`GL_COLOR_MATRIX`). В качестве этого преобразования выступает преобразование RGBA-цвета при помощи умножения его на матрицу 4×4 . После умножения на матрицу компоненты получившегося вектора сдвигаются и масштабируются. Эта возможность реализована в расширении `SGI_color_matrix`.

Последние две новые возможности связаны со смешением цветов (*blending*).

Во-первых, стало возможным задавать специальный цвет, который будет использоваться для получения коэффициентов смешения (простейший пример — смешение двух RGB-текстур с постоянными коэффициентами). Эта возможность реализована в расширении `EXT_blend_color`.

Во-вторых, появилась поддержка новых уравнений для смешения цветов (отличных от взвешенной суммы двух цветов). Два из этих уравнений выбирают максимум (минимум) из соответствующих компонент смешиваемых цветов. Еще два уравнения похожи на стандартное уравнение смешения цветов, но только вместо суммы цветов используется их разность. Данная возможность реализована в расширениях `EXT_blend_minmax` и `EXT_blend_subtract`. Для этого используется команда:

```
void glBlendEquation(GLenum mode);
```

В параметре `mode` передается требуемый тип операции:

- `GLFUNC_ADD`,
- `GL_FUNC_SUBTRACT`,
- `GL_FUNC_REVERSE_SUBTRACT`,
- `GL_MIN`, `GL_MAX`,
- `GL_LOGIC_OP`.

Версия 1.3

Версия OpenGL 1.3, вышедшая 14 августа 2001 года, сохранила полную совместимость с версиями 1.0—1.2, добавив целый ряд новых возможностей, в том числе и введенных ранее как ARB-расширения.

Одним из наиболее важных новшеств стало мультитекстурирование (рис. 1.1), т. е. возможность наложения сразу нескольких текстур со своими наборами текстурных координат, способами наложения и т. п., введенное расширением ARB_multitexture [1, 18].

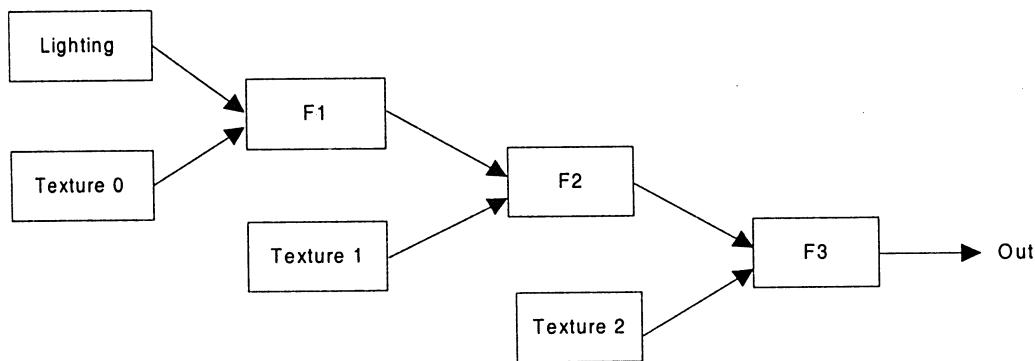


Рис. 1.1. Мультитекстурирование

Еще одно новшество — поддержка кубических текстурных карт (texture cube maps) и двух новых способов вычисления текстурных координат — GL_REFLECTION_MAP и GL_NORMAL_MAP. Эти возможности реализованы в расширении ARB_texture_cubemap [1, 3].

Также в данной версии впервые появилась поддержка сжатых текстур (расширение ARB_texture_compression).

В OpenGL 1.3 появилась и поддержка мультисэмплинга (multisampling) — очень простого (правда, весьма дорогостоящего) средства борьбы с так называемыми *погрешностями дискретизации* (aliasing artifacts).

Мультисэмплинг заключается в том, что для каждого пикселя выбирается несколько образцов (sample) и для каждого из них вычисляется цвет (рис. 1.2). Значения цвета для всех образцов, соответствующих данному пикселю, усредняются для получения итогового цвета соответствующего пикселя.

При этом к фреймбуферу добавляется еще один буфер (multisample buffer). Значения, соответствующие отдельным образцам, хранятся в этом буфере. Эта возможность реализована в расширении ARB_multisample.

•	•	•	•
•	•	•	•
•	•	•	•
•	•	•	•

Рис. 1.2. Мультисэмплинг: одному пикселу соответствует матрица вычисляемых образцов; цвет пикселя — среднее значение цвета по всем этим образцам

Также в эту версию OpenGL вошла поддержка нескольких новых режимов наложения текстуры (texture environment mode) — `GL_ADD`, `GL_DOT3` и `GL_COMBINE` [1, 20]. Введение этих режимов заметно расширило возможности OpenGL по использованию текстур, особенно для режима мультитекстурирования. Эта возможность реализована в расширениях `ARB_texture_env_add`, `ARB_texture_env_dot3` и `ARB_texture_env_combine`.

Еще в OpenGL 1.3 вошел новый режим отсечения текстурных координат `GL_CLAMP_TO_BORDER`, являющийся в определенном смысле дополнительным к режиму `GL_CLAMP_TO_EDGE`, введенному в версии 1.2. В этом режиме при построении промежуточных уровней в пирамидальном фильтровании для текселов,¹ лежащих на границе текстуры, соответствующий фильтр использует только значения пикселов с границы текстуры. Эта возможность реализована в расширении `ARB_texture_border_clamp`.

Также в эту версию вошли функции и константы, позволяющие хранить матрицы по столбцам, а не по строкам (как принято в OpenGL). Это удобно тем, что позволяет свободно использовать двухмерные матрицы из языка С.

Для этой цели были введены следующие команды:

```
void glLoadTransposeMatrix{fd}( T m [16] );
void glMultTransposeMatrix{fd}( T m [16] );
```

Данные команды эквивалентны командам `glLoadMatrix` и `glMultMatrix`, но в качестве входного параметра принимают матрицу 4×4 , организованную по столбцам. Эта возможность реализована в расширении `ARB_transpose_matrix`.

¹ Тексел — пиксель текстуры, от англ. texture element.

Версия 1.4

Версия OpenGL 1.4, вышедшая 24 июня 2002 года, также включила в себя целый ряд новых возможностей.

Расширение SGIS_generate_mipmap в OpenGL 1.4 добавило возможность автоматического вычисления (и перевычисления при необходимости) всех промежуточных уровней для пирамидального фильтрования. Для этого достаточно установить параметр текстуры GL_GENERATE_MIPMAP_SGIS в значение GL_TRUE:

```
glTexParameteri ( target, GL_GENERATE_MIPMAP_SGIS, GL_TRUE );
```

Также были добавлены новые режимы для вычисления коэффициентов при смешении цветов (расширение NV_blend_square). Например, чтобы задать закон вычисления коэффициентов для источника (*source*), можно использовать GL_SRC_COLOR и GL_ONE_MINUS_SRC_COLOR. Аналогично, в качестве способа вычисления коэффициентов для приемника (*destination*) стало возможным использовать GL_DST_COLOR и GL_ONE_MINUS_DST_COLOR.

Также в этой версии появилась полноценная поддержка текстур глубины (depth textures) и их использования для вычисления теней через режим наложения текстуры GL_TEXTURE_COMPARE_MODE. Эти возможности реализованы в расширениях ARB_depth_texture и ARB_shadow [18].

Еще одной полезной возможностью, ставшей наконец полноценной частью OpenGL, стала поддержка явного задания величины тумана в вершинах (расширение EXT_fog_coord).

Также в эту версию вошли команды glMultiDrawArrays и glMultiDrawElements (расширение EXT_multi_draw_arrays), позволяющие всего одной командой осуществить вывод сразу нескольких групп граней.

Расширение ARB_point_parameters добавило возможность задания дополнительных свойств точек, позволяя расстоянию до точки влиять на ее размер и прозрачность. Данная возможность оказалось особенно удобной для рендеринга различных систем частиц (particle systems).

Стало возможным задавать вторичный (secondary) цвет как вершинный параметр (расширение EXT_secondary_color).

Расширение EXT_blend_func_separate позволило (через функцию glBindFuncSeparate) задавать разные режимы смешения для цветовой и альфа-частей с помощью введенной функции glBindEquationSeparate.

Еще одной возможностью, весьма важной для создания теней методом теневых объемов [18], стало введение новых операций над буфером трафарета — GL_INCR_WRAP и GL_DECR_WRAP. В этих режимах, если значение в буфере трафарета выходит за пределы диапазона возможных значений (т. е. отведенных под значение бит в буфере трафарета уже не хватает), то значение

"вылезает" с другого конца диапазона возможных значений — увеличение на единицу максимального возможного значения (как правило, это 255) дает минимальное возможное значение (как правило, это 0), а уменьшение на единицу минимального возможного значения дает максимальное возможное значение (рис. 1.3).

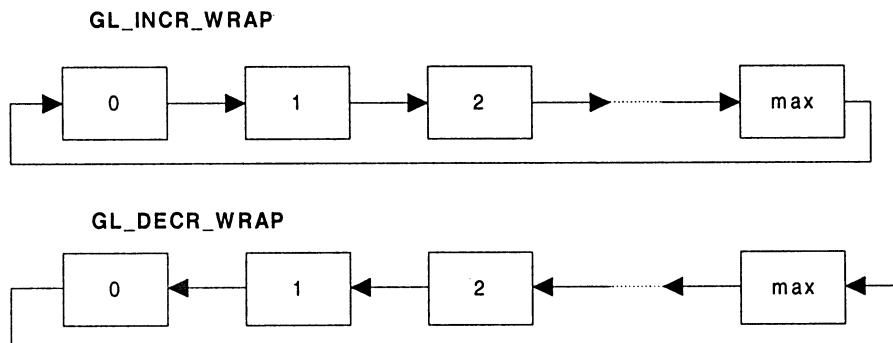


Рис. 1.3. Работа режимов GL_INCR_WRAP и GL_DECR_WRAP

Стандартные операции `GL_INCR` и `GL_DECR` подобной возможностью не обладают — когда значение в буфере трафарета достигает максимального возможного значения, последующие операции увеличения значения не изменяют его. Аналогично, уменьшение значения после достижения минимального возможного значения не имеет никакого эффекта (рис. 1.4).

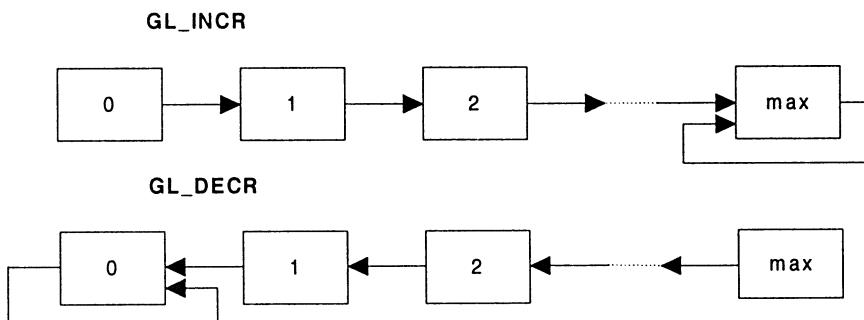


Рис. 1.4. Работа стандартных режимов GL_INCR и GL_DECR

Эта возможность реализована в расширении `EXT_stencil_wrap`.

Расширение `ARB_texture_env_crossbar` добавило возможность при задании режима наложения текстуры `GL_COMBINE` использовать цветовые значения из разных текстурных блоков в качестве источников для функции наложения.

Также была добавлена возможность задавать смещение для параметра λ , отвечающего за выбор уровня в пирамидальном фильтровании — введен новый параметр текстуры `GL_TEXTURE_LOD_BIAS`, определяющий это смещение. Эта возможность реализована в расширении `EXT_texture_lod_bias`.

В этой версии был введен еще один режим отсечения текстурных координат — `GL_MIRRORED_REPEAT`, что позволило во многих случаях отказаться от использования периодических текстур (рис. 1.5). Эта возможность реализована в расширении `ARB_texture_mirrored_repeat`.

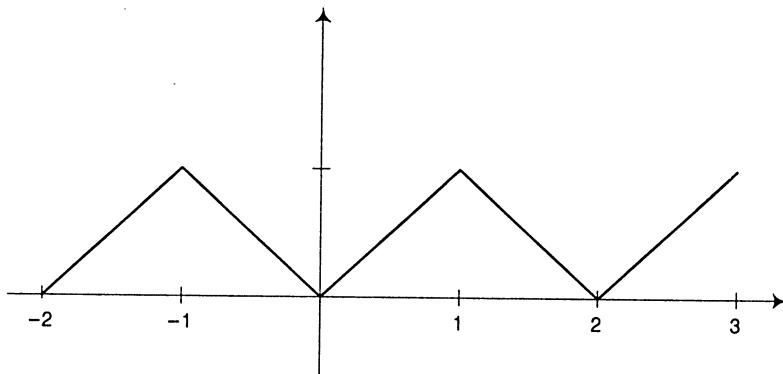


Рис. 1.5. Отсечение текстурных координат в режиме `GL_MIRRORED_REPEAT`

Расширение `ARB_window_pos` добавило возможность задавать при помощи команды `glWindowPos` растровое положение в координатах окна.

Версия 1.5

Эта версия OpenGL вышла 29 июля 2003 года, с ее выходом были одобрены расширения `ARB_shader_objects`, `ARB_vertex_shader` и `ARB_fragment_shader`, позволяющие использовать шейдерный язык высокого уровня в программах, применяющих OpenGL.

В эту версию из расширения `ARB_vertex_buffer_object` была перенесена поддержка буферов-объектов, позволяющих эффективно использовать память графического процессора для хранения данных [1, 18].

Также в версии 1.5 появилась поддержка запросов на определение видимости (`occlusion queries`), пришедшая из расширения `ARB_occlusion_query` [1].

Был расширен набор функций сравнения для теневых карт — теперь стало возможным использовать все восемь стандартных функций сравнения (а не только `GL_EQUAL` и `GL_GEQUAL`). Эта возможность реализована в расширении `EXT_shadow_funcs`.

Сводка изменений в версиях OpenGL

В табл. 1.2 приведена краткая сводка всех добавленных возможностей с указанием расширения, из которого соответствующая возможность была добавлена.

Таблица 1.2. Изменения в различных версиях OpenGL

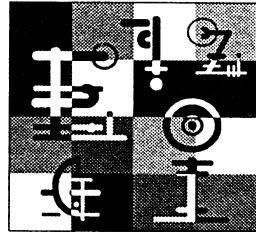
Версия OpenGL	Возможность	Исходное расширение
1.1	Поддержка вершинных массивов	EXT_vertex_array
	Смещение глубины фрагментов	EXT_polygon_offset
	Использование логических операций при выводе фрагментов	EXT_blend_logic_op
	Задание внутреннего формата текстур	EXT_texture
	Использование прокси-текстур для определения параметров текстуры	EXT_texture
	Копирование прямоугольных блоков в текстуру	EXT_copy_texture EXT_subtexture
	Введение текстурного объекта с параметрами состояния	EXT_texture_object
1.2	Поддержка трехмерных текстур	EXT_texture3D
	Поддержка формата BGRA	EXT_bgra
	Поддержка упакованных форматов пикселов	EXT_packed_pixels
	Автоматическое масштабирование вектора нормали	EXT_rescale_normal
	Поддержка вторичного цвета	EXT_separate_specular_color
	Режим отсечения текстурных координат GL_CLAMP_TO_EDGE	SGIS_texture_edge_clamp
	Задание диапазона изменения параметра для выбора уровня в пирамидальном фильтровании	SGIS_texture_lod
	Загрузка не всех уровней для пирамидального фильтрования	SGIS_texture_lod
	Поддержка операции свертки при переносе пикселов	EXT_convolution HP_convolution_border_modes
	Поддержка lookup-таблиц для RGBA-режимов	EXT_color_table EXT_color_subtable

Таблица 1.2 (продолжение)

Версия OpenGL	Возможность	Исходное расширение
1.2	Введение матрицы для умножения на цвет	SGI_color_matrix
	Использование специального цвета для смешения цветов	EXT_blend_color
	Новый способ смешения цветов — минимум и максимум	EXT_blend_minmax
	Новый способ смешения цветов — разность	EXT_blend_subtract
1.3	Мультитекстурирование	ARB_multitexture
	Поддержка кубических текстурных карт	ARB_texture_cube_map
	Мультисэмплинг	ARB_multisample
	Поддержка режимов наложения текстур — GL_ADD, GL_DOT3 и GL_COMBINE	ARB_texture_env_add ARB_texture_env_dot3 ARB_texture_env_combine
	Режим отсечения текстурных координат GL_CLAMP_TO_BORDER	ARB_texture_border_clamp
	Задание матриц по столбцам	ARB_transpose_matrix
1.4	Автоматическое построение уровней в пирамидальном фильтровании	SGIS_generate_mipmap
	Новые режимы вычисления коэффициентов для смешения цветов	NV_blend_square
	Режим наложения текстуры GL_TEXTURE_COMPARE_MODE	ARB_depth_texture ARB_shadow
	Поддержка задания тумана в вершинах	EXT_fog_coord
	Задание дополнительных свойств точек	ARB_point_parameters
	Задание вторичного цвета как атрибута вершины	EXT_secondary_color
	Задание различных режимов смешения для RGB- и альфа-составляющих	EXT_blend_func_separate
	Новые операции над буфером трафарета — GL_INCR_WRAP и GL_DECR_WRAP	EXT_stencil_wrap

Таблица 1.2 (окончание)

Версия OpenGL	Возможность	Исходное расширение
1.4	<p>Использование при задании режима наложения GL_COMBINE цветовых значений из различных текстурных блоков</p> <p>Задание смещения для параметра λ</p> <p>Режим отсечения текстурных координат GL_MIRRORED_REPEAT</p> <p>Задание растрового положения в координатах окна</p>	ARB_texture_env_crossbar EXT_texture_lod_bias ARB_texture_mirrored_repeat ARB_window_pos
1.5	<p>Поддержка буферов-объектов (VBO)</p> <p>Поддержка запросов на определение видимости</p> <p>Использование при построении теневых карт всех стандартных функций сравнения</p>	ARB_vertex_buffer_object ARB_occlusion_query EXT_shadow_funcs



Глава 2

Расширения ARB_vertex_program и ARB_fragment_program. Появление Cg

Все, рассмотренные в предыдущей главе добавления в OpenGL, только расширяют возможности стандартного конвейера рендеринга (rendering pipeline), при этом все равно вся обработка данных выполняется посредством стандартного конвейера (рис. 2.1).

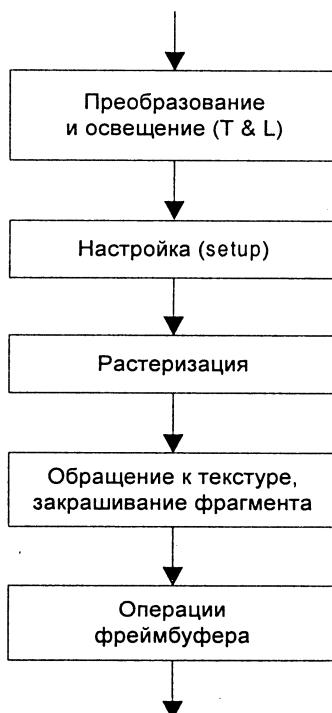


Рис. 2.1. Конвейер рендеринга OpenGL

Однако в целом ряде случаев оказывается желательным иметь возможность явно задавать преобразования данных для отдельных шагов конвейера при помощи специальных программ.

Выделяют два типа таких программ (иногда называемых шейдерами) — вершинные (vertex) и фрагментные (fragment).

Вершинная программа позволяет заменить блок преобразования и освещения (T & L) конвейера OpenGL, т. е. обработку отдельных вершин, на явно задаваемые пользователем действия. Такая программа выполняется независимо для каждой вершины и принимает в качестве входных данных как данные самой вершины (вершинные атрибуты, vertex attributes), так и локальные и глобальные переменные, а также параметры состояния OpenGL. При этом вершинная программа не может ни создавать новых вершин, ни уничтожать существующие вершины (т. е. не может изменять топологию данных). Также вершинная программа не обладает доступом к какой-либо топологической информацией (ребра, грани и т. п.).

Обычно вершинная программа заменяет собой следующую функциональность OpenGL:

- преобразования вершин при помощи модельной матрицы (modelview) и матрицы проектирования (projection);
- смешение (weighting/blending) вершин;
- преобразования нормали;
- вершинное освещение;
- цветные материалы;
- вычисление текстурных координат и их преобразование;
- вычисление степени затуманивания вершины;
- задаваемые пользователем операции.

Фрагментная программа выполняется отдельно для каждого фрагмента, получаемого при растеризации, и заменяет собой блок обращения к текстурам и закрашивания фрагмента в конвейере рендеринга (рис. 2.2).

Простейшим способом применения вершинных и фрагментных программ в OpenGL заключается в использовании расширений ARB_vertex_program и ARB_fragment_program [1, 18]. Эти расширения позволяют использовать в OpenGL-программах вершинные и фрагментные программы, написанные на специальном ассемблере. Хотя системы команд в версиях этого ассемблера для вершинных и фрагментных программ несколько отличаются, в действительности они очень близки. Все программы на этом ассемблере работают с набором регистров, каждый из которых представляет собой четырехмерный вещественный вектор $((x, y, z, w), (r, g, b, a)$ или (s, t, r, q)). В этот набор регистров обычно входят атрибуты (обрабатываемой вершины

или фрагмента), локальные переменные, переменные окружения и временные переменные. Результаты своей работы программа записывает в *выходные регистры*. Вершинная программа также имеет в своем распоряжении набор адресных регистров. Наборы регистров для вершинной и фрагментной программ приводятся на рис. 2.3 и 2.4.

Каждая такая программа представляет собой набор строк, строка может соответствовать не более чем одной команде (могут присутствовать пустые строки и комментарии).

В листингах 2.1 и 2.2 приводятся примеры вершинной и фрагментной программ для вычисления попиксельного диффузного и бликового освещения.

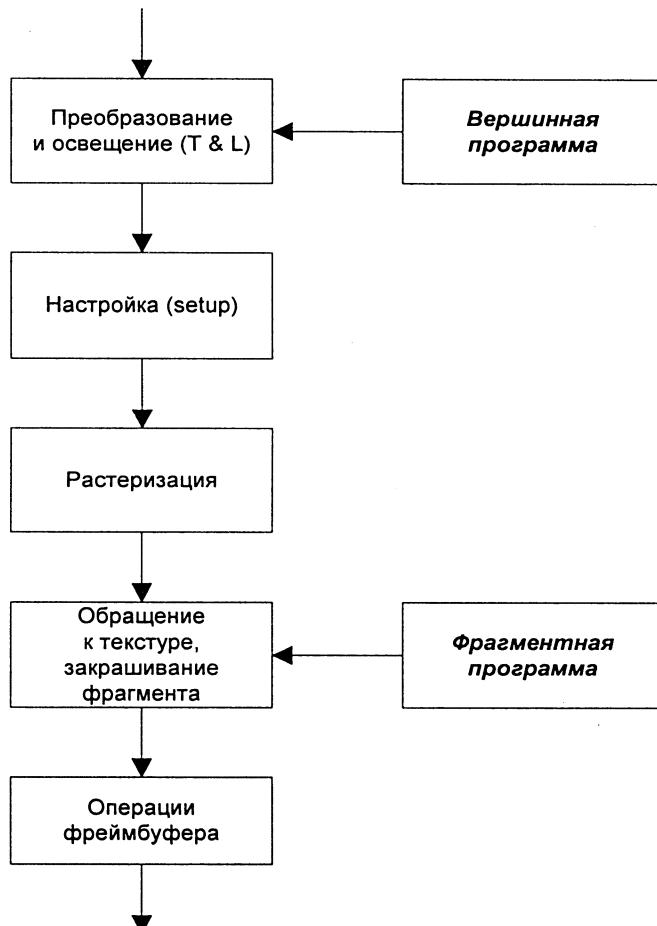


Рис. 2.2. Место вершинной и фрагментной программ в конвейере рендеринга

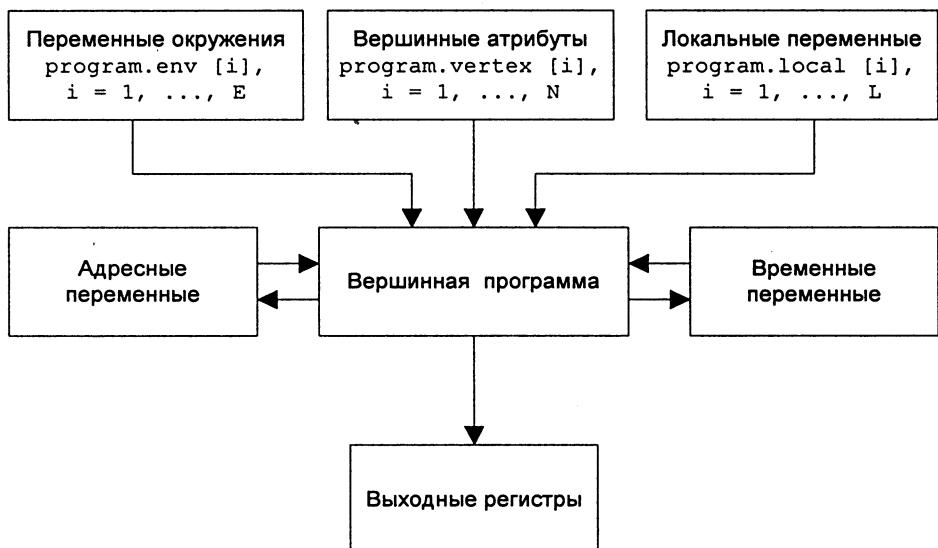


Рис. 2.3. Регистры, доступные вершинным программам

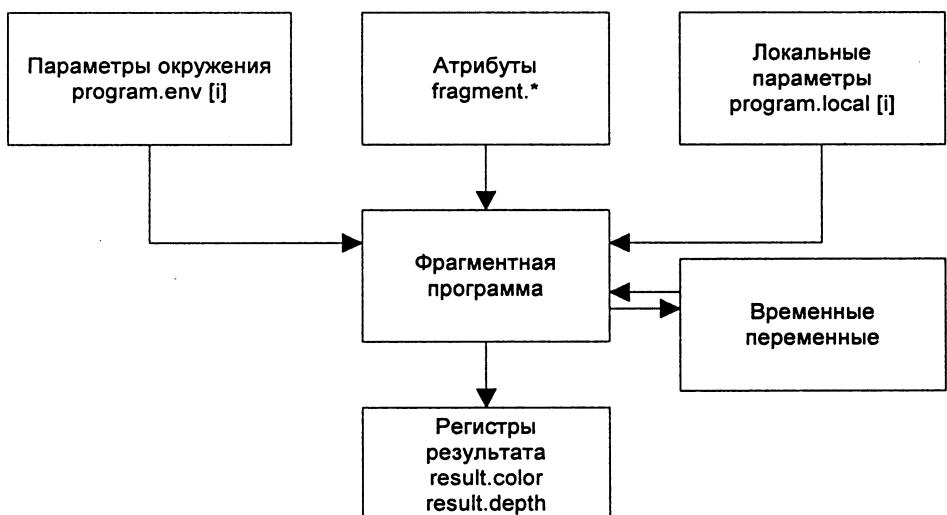


Рис. 2.4. Регистры, доступные фрагментным программам

Листинг 2.1. Пример вершинной программы

```
!!ARBvp1.0
#
# Simple vertex shader to setup data for per-pixel specular lighting
#
# on entry:
#     vertex.position
#     vertex.normal      - normal vector (n) of TBN basic
#     vertex.texcoord [0] - normal texture coordinates
#
#     program.local [0]   - eye position
#     program.local [1]   - light position
#
#     state.matrix.program [0] - rotation matrix for the object
#
# on exit:
#     result.texcoord [0]   - texture coordinates
#     result.texcoord [1].x - diffuse component
#     result.texcoord [1].y - specular component
#
ATTRIB pos      = vertex.position;
ATTRIB n        = vertex.normal;
PARAM eye      = program.local [0];
PARAM light    = program.local [1];
PARAM mvp [4]   = { state.matrix.mvp };
PARAM mv [4]    = { state.matrix.modelview };
PARAM mv0 [4]   = { state.matrix.modelview.invtrans };
PARAM half     = 0.5;

TEMP p, l, v, h, nt, temp, t;

# transform position
DP4 p.x, pos, mv [0];
DP4 p.y, pos, mv [1];
DP4 p.z, pos, mv [2];
DP4 p.w, pos, mv [3];
```

```
# compute l (vector to light)
ADD l, -p, light;      # l = light - p

# normalize it (we need to correctly compute h)
DP3 temp.x, l, l;      # now temp.x = (l,l)
RSQ temp.y, temp.x;    # compute inverse square root of (v,v)
MUL l, l, temp.y;      # normalize

# compute v (vector to viewer)
ADD v, -p, eye;        # v = eye - pos

# normalize it (we need to correctly compute h)
DP3 temp.x, v, v;      # now temp.x = (v,v)
RSQ temp.y, temp.x;    # compute inverse square root of (v,v)
MUL v, v, temp.y;      # normalize

# compute h = l+v
ADD h, l, v;

# normalize it (we need to correctly compute h)
DP3 temp.x, h, h;      # now temp.x = (h,h)
RSQ temp.y, temp.x;    # compute inverse square root of (h,h)
MUL h, h, temp.y;      # normalize

# transform normal by MV.inverse.transpose
DP3 nt.x, n, mv0 [0];
DP3 nt.y, n, mv0 [1];
DP3 nt.z, n, mv0 [2];
MOV nt.w, 1;

# normalize it
DP3 temp.x, nt, nt;    # now temp.x = (nt,nt)
RSQ temp.y, temp.x;    # compute inverse square root of (h,h)
MUL nt, nt, temp.y;   # normalize

# compute diffuse and specular terms
DP3 temp.x, nt, l;      # compute dot products (n,l) and (n,h)
DP3 temp.y, nt, h;
MOV temp.w, 50;
```

```

# compute lighting
LIT t, temp; # now t.y is diffuse
              # t.z is specular

# store it into texcoord [1].xy
MOV result.texcoord [1].x, t.y;
MOV result.texcoord [1].y, t.z;

# copy texcoord [0]
MOV result.texcoord [0], vertex.texcoord [0];

# copy primary and secondary colors
MOV result.color, vertex.color;
MOV result.color.secondary, vertex.color.secondary;

# transform position into clip space
DP4 result.position.x, pos, mvp [0];
DP4 result.position.y, pos, mvp [1];
DP4 result.position.z, pos, mvp [2];
DP4 result.position.w, pos, mvp [3];

# we're done
END

```

Листинг 2.2. Пример фрагментной программы

```

!!ARBfp1.0
#
# simple specular shader
# on entry:
#   fragment.texcoord [0]      - texture coordinates
#   fragment.texcoord [1].xy - diffuse and specular components
#
#   texture [0] - decal map
#
PARAM specColor = { 0, 0, 1 };
TEMP color, temp, diffuse, specular;

# get decal color

```

```

TEX      temp, fragment.texcoord [0], texture [0], 2D;

ADD      diffuse, fragment.texcoord [1].x, 0.4;
MUL      specular, fragment.texcoord [1].y, specColor;

MUL      color, temp, diffuse;
ADD      color, color, specular;
MOV      color.w, 1;

MOV      result.color, color;

END

```

Обратите внимание на то, что каждая из этих программ начинается со строки специального вида, задающей тип и версию соответствующей программы. Все вершинные программы, вводимые расширением `ARB_vertex_program`, должны начинаться со строки `!!ARBvp1.0`, а все фрагментные программы, вводимые расширением `ARB_fragment_program`, — со строки `!!ARBfp1.0`.

Каждая программа заканчивается командой `END`.

Весь текст от символа `#` и до конца строки считается комментарием и игнорируется.

Каждая команда должна заканчиваться точкой с запятой `(;)`.

Программа состоит как из команд, описывающих регистры, так и из команд обработки данных (т. е. операций над регистрами).

Команда `ATTRIB` позволяет задать удобное имя для входного атрибута (вершины или фрагмента). Команда `PARAM` служит для задания параметров — постоянных векторов и параметров состояния OpenGL; все параметры (как и атрибуты) доступны только для чтения.

Команда `OPTION` позволяет задавать специальные опции загрузки и выполнения соответствующей программы.

Команда `TEMP` позволяет создавать временные переменные, доступные как для чтения, так и для записи.

Все остальные команды служат для обработки данных и имеют следующий вид:

```
opCode dest, [-]src0 [,[-]src1 [,[-]src2]];
```

Здесь `opCode` — символьный код инструкции, `dest` — регистр, в который будет помещен результат выполнения команды, а величины `src0`, `src1` и `src2` — регистры с исходными данными. Квадратными скобками обозначены необязательные величины.

Примеры команд:

```
MOV R1, R2;
MAD R1, R2, R3, -R4;
```

Необязательный знак минус (-) позволяет в качестве источника использовать как сам регистр, так и вектор, получающийся из него умножением на -1.

Для входных регистров (*src0*, *src1* и *src2*) существует возможность использовать вместо самого регистра вектор, получающийся из исходного путем перестановки его компонент, например:

```
MOV R1, R2.yzwx;
MOV R2, -R3.yzwx;
```

Для выходного регистра можно задать маску, защищающую отдельные компоненты регистра от изменения. Пример:

```
MOV R2.xw, -R3;
```

Обратите внимание, что следующие две команды эквивалентны:

```
MOV R1, R2.xxxx;
MOV R1, R2.x;
```

В табл. 2.1 и 2.2 приведены команды для вершинных и фрагментных программ.

Таблица 2.1. Команды для вершинной программы

Команда (назначение)	Синтаксис	Комментарий
ABS (модуль)	ABS dest, src;	Покомпонентное вычисление модуля: $dest=fabs(src)$
ADD (сложение)	ADD dest, src0, src1;	Сложение значений двух параметров, сумма записывается в регистр результата: $dest=src0 + src1$
ARL	ARL dest.C1, src.C2;	Загрузка в адресный регистр значения целой части скалярного операнда
DP3 (скалярное произведение по первым трем компонентам)	DP3 dest, src0, src1;	Вычисление скалярного произведения источников как трехмерных векторов: $dest=src0.x*src1.x + src0.y*src1.y + src0.z*src1.z$
DP4 (скалярное произведение)	DP4 dest, src0, src1;	Вычисление скалярного произведения четырехмерных векторов; результат записывается во все компоненты регистра dest: $dest=src0.x*src1.x + src0.y*src1.y + src0.z*src1.z + src0.w*src1.w$

Таблица 2.1 (продолжение)

Команда (назначение)	Синтаксис	Комментарий
DPH (однородное скалярное произведение)	DPH dest, src0, src1;	Вычисление однородного скалярного произведения: $dest = src0.x * src1.x + src0.y * src1.y + src0.z * src1.z + src1.w$
DST (расстояние)	DST dest, src0.C1, src1.C2;	Эффективное вычисление вектора $dest = (1, d, d^2, 1/d)$ по двум скалярным значениям: $src0.C1 = d^2$ $src1.C2 = 1/d$
EX2 (приближенное значение 2 в степени)	EX2 dest, src.C;	Вычисление для заданного скалярного операнда $src.C$ приближенного значения $2^{\text{src}.C}$; результат записывается во все компоненты регистра $dest$
EXP (значение 2 в степени)	EXP dest, src.C;	Более точное вычисление $2^{\text{src}.C}$: $dest.x = 2^{\text{floor}(\text{src}.C)}$ $dest.y = \text{src}.C - \text{floor}(\text{src}.C)$ $dest.z =$ $= \text{roughAppr2ToX}(\text{floor}(\text{src}.C))$ $dest.w = 1$ Функция roughAppr2ToX вычисляет приближенное значение с точностью до 2^{-11}
FLR (вычисление функции floor)	FLR dest, src;	Покомпонентное вычисление целой части (функция floor, наибольшее целое, не превосходящее аргумент) компонент источника
FRC (дробная часть)	FRC dest, src;	Покомпонентное вычисление дробной части: $\text{frac}(x) = x - \text{floor}(x)$
LG2 (приближенное значение логарифма по основанию 2)	LG2 dest, src.C;	Вычисление приближенного значения логарифма по основанию 2 от скалярного аргумента; результат записывается во все компоненты источника
LIT (коэффициенты освещения)	LIT dest, src;	Служит для ускорения вычисления диффузной и бликовой освещенности в вершинах: $\text{src}.x = (n, 1)$ $\text{src}.y = (n, h)$ $\text{src}.w = p$ (степень, приводится к отрезку $[-128, 128]$). На выходе: $dest.x = 1$ $dest.y = \max(0, (n, 1))$ $dest.z = (n, 1) > 0 ? \text{roughAppr2ToX}(\max(0, (n, h))^p) : 0$ $dest.w = 1$

Таблица 2.1 (продолжение)

Команда (назначение)	Синтаксис	Комментарий
LOG (логарифм по основанию 2)	LOG dest, src.C;	Вычисление приближенного значения логарифма по основанию 2 от скалярного операнда: temp=fabs(src.C) dest.x=floor(log2(temp)) dest.y=temp/2^floor(log2(temp)) dest.z=roughAppr2ToX(temp) dest.w=1
MAD (умножение и сложение)	MAD dest, src0, src1, src2;	Вычисление значения: dest=src0*src1 + src2
MAX (максимум)	MAX dest, src0, src1;	Покомпонентное вычисление максимума: dest=max(src0,src1)
MIN (минимум)	MIN dest, src0, src1;	Покомпонентное вычисление минимума: dest=min(src0,src1)
MOV (копирование)	MOV dest, src;	Запись значений компонент источника в компоненты примника: dest=src
MUL (покомпонентное умножение)	MUL dest, src0, src1;	Вычисление покомпонентного произведения: dest=src0*src1
POW (возведение в степень)	POW dest, src0.C1, src1.C2;	Вычисление приближенного значения первого скалярного операнда, возведенного в степень второго скалярного операнда, и запись результата во все компоненты регистра dest: apprPow(a,b)= =apprExp2(b*apprLog2*a))
RCP (обратное значение)	RCP dest, src.C;	Вычисление приближенного значения, обратного к скалярному операнду, и запись во все компоненты регистра dest: dest=1/src.C
RSQ	RSQ dest, src.C;	Возведение скалярного операнда в степень 1/2: dest=1/sqrt(fabs(src.C))
SGE ("больше или равно")	SGE dest, src0, src1;	Покомпонентное сравнение и запись результатов сравнения для каждой компоненты в регистр dest: dest.x=src0.x>=src1.x?1:0 dest.y=src0.y>=src1.y?1:0 dest.z=src0.z>=src1.z?1:0 dest.w=src0.w>=src1.w?1:0

Таблица 2.1 (окончание)

Команда (назначение)	Синтаксис	Комментарий
SLT ("меньше чем")	SLT dest, src0, src1;	Покомпонентное сравнение и запись результатов сравнения для каждой компоненты в регистр dest: dest.x=src0.x<src1.x?1:0 dest.y=src0.y<src1.y?1:0 dest.z=src0.z<src1.z?1:0 dest.w=src0.w<src1.w?1:0
SUB (разность)	SUB dest, src0, src1;	Покомпонентное вычисление разности операндов: dest=src0 - src1
SWZ ("перемешивание" компонент)	SWZ dest, src, <extSwizzle>	Произвольное перемешивание компонент (или их части); источником для компоненты может быть любая компонента исходного регистра (со знаком или без) или значения 0 и 1. Здесь <extSwizzle> ::= <comp>, <comp>, <comp>, <comp> <comp> ::= [-] (0 1 x y z w)
XPD (векторное произведение)	XPD dest, src0, src1;	Вычисление векторного произведения первых трех компонент первого операнда и первых трех компонент второго операнда, результат записывается в первые три компоненты регистра dest

Таблица 2.2. Команды для фрагментной программы

Команда (назначение)	Синтаксис	Комментарий
ABS (модуль)	ABS dest, src;	Покомпонентное вычисление модуля: dest=fabs(src)
ADD (сложение)	ADD dest, src0, src1;	Покомпонентное сложение двух параметров; сумма записывается в регистр результата: dest=src0 + src1
CMP (сравнение)	CMP dest, src0, src1, src2;	Покомпонентно, в зависимости от знака соответствующей компоненты src0, выбирается очередная компонента либо из src1, либо из src2: dest.x = src0.x < 0 ? src1.x : src2.x dest.y = src0.y < 0 ? src1.y : src2.y dest.z = src0.z < 0 ? src1.z : src2.z dest.w = src0.w < 0 ? src1.w : src2.w

Таблица 2.2 (продолжение)

Команда (назначение)	Синтаксис	Комментарий
COS (косинус угла)	COS dest, src.C;	Покомпонентное вычисление косинуса скалярного аргумента и запись его значения во все разрешенные компоненты результата; угол выражается в радианах и не обязательно находится в пределах [-PI, PI]
DP3 (скалярное произведение по первым трем компонентам)	DP3 dest, src0, src1;	Вычисление скалярного произведения источников как трехмерных векторов: $\text{dest} = \text{src0}.x * \text{src1}.x + \text{src0}.y * \text{src1}.y + \text{src0}.z * \text{src1}.z$
DP4 (скалярное произведение)	DP4 dest, src0, src1;	Вычисление скалярного произведения четырехмерных векторов и запись результата во все компоненты регистра dest: $\text{dest} = \text{src0}.x * \text{src1}.x + \text{src0}.y * \text{src1}.y + \text{src0}.z * \text{src1}.z + \text{src0}.w * \text{src1}.w$
DPH (однородное скалярное произведение)	DPH dest, src0, src1;	Вычисление однородного скалярного произведения: $\text{dest} = \text{src0}.x * \text{src1}.x + \text{src0}.y * \text{src1}.y + \text{src0}.z * \text{src1}.z + \text{src1}.w$
DST (расстояние)	DST dest, src0, src1;	Эффективное вычисление вектора $\text{dest} = (1, d, d^2, 1/d)$ по двум значениям: $\text{src0} = (*, d^2, d^2, *)$ $\text{src1} = (*, 1/d, *, 1/d)$ Звездочка (*) означает, что соответствующее значение не важно
EX2 (приближенное значение 2 в степени)	EX2 dest, src.C;	Вычисление для заданного скалярного операнда src.C приближенного значения $2^{\text{src.C}}$ и запись результата во все компоненты регистра dest
FLR (вычисление функции floor)	FLR dest, src;	Покомпонентное вычисление целой части (функция floor — наибольшее целое, не пре-восходящее аргумент) источника
FRC (дробная часть)	FRC dest, src;	Покомпонентное вычисление дробной части операнда: $\text{frac}(x) = x - \text{floor}(x)$
LG2 (приближенное значение логарифма по основанию 2)	LG2 dest, src.C;	Вычисление приближенного значения логарифма по основанию 2 от скалярного аргумента и запись результата во все компоненты источника

Таблица 2.2 (продолжение)

Команда (назначение)	Синтаксис	Комментарий
LIT (коэффициен- ты освещения)	LIT dest, src;	Ускорение вычисления диффузной и бликовой освещенности в вершинах: $src.x=(n, 1)$ $src.y=(n, h)$ $src.w=p$ (степень, приводится к отрезку [−128, 128]). На выходе $dest.x=1$ $dest.y=\max(0, (n, 1))$ $dest.z=(n, 1)>0?roughAppr2ToX(\max(0,$ $(n, h))^p):0$ $dest.w=1$
LRP (линейная интерполяция)	LRP dest, src0, src1, src2;	Покомпонентная линейная интерполяция между значениями src1 и src2 на основе src0: $dest.x=src0.x*src1.x + (1 - src0.x)*src2.x$ $dest.y=src0.y*src1.y + (1 - src0.y)*src2.y$ $dest.z=src0.z*src1.z + (1 - src0.z)*src2.z$ $dest.w=src0.w*src1.w + (1 - src0.w)*src2.w$
MAD (умножение и сложение)	MAD dest, src0, src1, src2;	Вычисление значения: $dest=src0*src1 + src2$
MAX (вычисление максимума)	MAX dest, src0, src1;	Покомпонентное вычисление максимума $dest=\max(src0, src1)$
MIN (вычисление минимума)	MIN dest, src0, src1;	Покомпонентное вычисление минимума $dest=\min(src0, src1)$
MOV (копирование)	MOV dest, src;	$dest=src$
MUL (покомпонент- ное умножение)	MUL dest, src0, src1;	Вычисление покомпонентного произведения: $dest=src0*src1$
POW (возвведение в степень)	POW dest, src0.C1, src1.C2;	Вычисление приближенного значения первого скалярного операнда, введенного в степень второго скалярного операнда, и запись ре- зультата во все компоненты регистра dest: $\text{apprPow}(a, b) = \text{apprExp2}(b * \text{apprLog2}(a))$ Обратите внимание: $0^0 = 1$
RCP (обратное значение)	RCP dest, src.C;	Вычисление приближенного значения, обрат- ного к скалярному операнду, и запись резуль- тата во все компоненты регистра dest: $dest=1/src.C$

Таблица 2.2 (продолжение)

Команда (назначение)	Синтаксис	Комментарий
RSQ	RSQ dest, src.C;	Возведение скалярного операнда в степень -1/2: dest=1/sqrt(fabs(src.C))
SCS (синус и коси- нус)	SCS dest, src.C;	Одновременное вычисление синуса и косинуса и запись этих значений в ху-компоненты реги- стра dest: dest.x=cos (src.C) dest.y=sin(src.C) Значение скалярного операнда должно нахо- диться в пределах [-PI, PI], компоненты z и w результата не определены
SGE ("больше или равно")	SGE dest, src0, src1;	Выполнение покомпонентного сравнения “больше или равно” над аргументами и запись результатов в соответствующие компоненты регистра dest: dest.x=src0.x>=src1.x?1:0 dest.y=src0.y>=src1.y?1:0 dest.z=src0.z>=src1.z?1:0 dest.w=src0.w>=src1.w?1:0
SIN (синус)	SIN dest, src.C;	Вычисление синуса скалярного операнда, вы- раждающего угол в радианах: dest=sin(src.C) Угол не обязательно находится в пределах [-PI, PI]
SLT ("меньше")	SLT dest, src0, src1;	Выполнение покомпонентного сравнения “меньше” над аргументами и запись результа- тов в соответствующие компоненты регистра dest: dest.x=src0.x<src1.x?1:0 dest.y=src0.y<src1.y?1:0 dest.z=src0.z<src1.z?1:0 dest.w=src0.w<src1.w?1:0
SUB (разность)	SUB dest, src0, src1;	Покомпонентное вычисление разностиope- рандов: dest=src0 - src1
SWZ ("перемешива- ние" компо- нент)	SWZ dest, src, <extSwizzle>	Произвольное перемешивание компонент (или их части), источником для компоненты может быть любая компонента исходного регистра (со знаком или без) или значения 0 и 1: <extSwizzle> ::= <comp>, <comp> <comp>, <comp> <comp> ::= [-] (0 1 x y z w r g b a)

Таблица 2.2 (окончание)

Команда (назначение)	Синтаксис	Комментарий
XPD (векторное произведение)	XPD dest, src0, src1;	Вычисление векторного произведения первых трех компонент первого операнда и первых трех компонент второго операнда и запись результата в первые три компонента регистра dest. Значение dest.w не определено
TEX	TEX dest, src0, texture [n], type;	Выборка из текстуры типа type с n-го текстурного блока; в качестве текстурных координат используются первые компоненты параметра src0; результат записывается в регистр dest. Параметр type принимает значение 1D, 2D, 3D, cube или rectangle
TXP	TXP dest, src0, texture [n], type;	Выборка из текстуры типа type с n-го текстурного блока; в качестве текстурных координат используются первые компоненты параметра src0, деленные на src0.w; результат записывается в регистр dest. Параметр type принимает значение 1D, 2D, 3D, cube и rectangle
TXB	TXB dest, src0, texture [n], type;	Выборка из текстуры типа type с n-го текстурного блока; в качестве текстурных координат используются первые компоненты параметра src0; результат записывается в регистр dest. Параметр type принимает значение 1D, 2D, 3D, cube и rectangle. Значение src0.w используется для смещения уровня в пирамидальном фильтровании, с которого будет взято значение
KIL (условное прерывание выполнения фрагментной программы)	KIL src;	В случае, когда хотя бы одна из компонент операнда меньше нуля, выполнение фрагментной программы прерывается для данного фрагмента. При этом дальнейшие шаги конвейера для данного фрагмента будут пропущены

В листинге 2.3 приведен пример программы на языке C++, строящей изображение освещенного тора с использованием вершинной и фрагментной программ из листингов 2.1 и 2.2.

Для удобства работы программы из этого листинга использует специальные классы (`vertexProgram` и `FragmentProgram`) для облегчения работы с вершинными и фрагментными программами (полную реализацию этих классов можно найти на сопроводительном компакт-диске книги).

Листинг 2.3. Построение изображения освещенного тора с использованием вершинной и фрагментной программ из листингов 2.1 и 2.2

```

// 
// Simple example of using ARB vertex/fragment programs in OpenGL
//

#include      <glut.h>
#include      <stdio.h>
#include      <stdlib.h>

#include      "libTexture.h"
#include      "TypeDefs.h"
#include      "Vector3D.h"
#include      "Vector2D.h"
#include      "Vector4D.h"
#include      "VertexProgram.h"
#include      "FragmentProgram.h"

VertexProgram    vertexProgram;
FragmentProgram fragmentProgram;

Vector3D  eye    ( 7, 5, 7 );           // camera position
Vector3D  light   ( 5, 0, 4 );          // light position
Vector3D  rot    ( 0, 0, 0 );
float     angle   = 0;
int       mouseOldX = 0;
int       mouseOldY = 0;
unsigned  decalMap;

void init ()
{
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );
    glEnable      ( GL_DEPTH_TEST );
}

```

```
glEnable      ( GL_TEXTURE_2D );
glDepthFunc   ( GL_LEQUAL. );

glHint       ( GL_POLYGON_SMOOTH_HINT, GL_NICEST );
glHint       ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}

void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

        // draw the light
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();

    glDisable      ( GL_TEXTURE_2D );
    glTranslatef   ( light.x, light.y, light.z );
    glColor4f      ( 1, 1, 1, 1 );
    glutSolidSphere ( 0.1f, 15, 15 );

    glPopMatrix ();
    glEnable      ( GL_TEXTURE_2D );

    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();

    glRotatef     ( rot.x, 1, 0, 0 );
    glRotatef     ( rot.y, 0, 1, 0 );
    glRotatef     ( rot.z, 0, 0, 1 );

vertexProgram.enable ();
vertexProgram.bind   ();

fragmentProgram.enable ();
fragmentProgram.bind   ();

    glBindTexture ( GL_TEXTURE_2D, decalMap );

    glutSolidTeapot ( 1.5 );
}
```

```
vertexProgram.disable ();
fragmentProgram.disable ();

glPopMatrix ();

glutSwapBuffers ();
}

void reshape ( int w, int h )
{
    glViewport ( 0, 0, (GLsizei)w, (GLsizei)h );
    glMatrixMode ( GL_PROJECTION );
        // factor all camera ops into
        // projection matrix
    glLoadIdentity ();
    gluPerspective ( 60.0, (GLfloat)w/(GLfloat)h, 1.0, 60.0 );
    gluLookAt ( eye.x, eye.y, eye.z, // eye
                0, 0, 0,           // center
                0, 0, 1 );          // up

    glMatrixMode ( GL_MODELVIEW );
    glLoadIdentity ();
}

void motion ( int x, int y )
{
    rot.y -= ((mouseOldY - y) * 180.0f) / 200.0f;
    rot.z -= ((mouseOldX - x) * 180.0f) / 200.0f;
    rot.x = 0;

    if ( rot.z > 360 )
        rot.z -= 360;

    if ( rot.z < -360 )
        rot.z += 360;

    if ( rot.y > 360 )
        rot.y -= 360;

    if ( rot.y < -360 )
```

```
    rot.y += 360;

    mouseOldX = x;
    mouseOldY = y;

    glutPostRedisplay ();
}

void mouse ( int button, int state, int x, int y )
{
    if ( state == GLUT_DOWN )
    {
        mouseOldX = x;
        mouseOldY = y;
    }
}

void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' ) // quit requested
        exit ( 0 );
}

void animate ()
{
    angle = 0.004f * glutGet ( GLUT_ELAPSED_TIME );

    light.x = 4*cos ( angle );
    light.y = 4*sin ( angle );
    light.z = 3 + 0.3 * sin ( angle / 3 );

    vertexProgram.enable      ();
    vertexProgram.bind        ();
    vertexProgram.local [0] = eye;
    vertexProgram.local [1] = light;
    vertexProgram.disable     ();

    glutPostRedisplay ();
}
```

```
int main ( int argc, char * argv [] )  
{  
    // initialize glut  
    glutInit ( &argc, argv );  
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );  
    glutInitWindowSize ( 500, 500 );  
  
    // create window  
    glutCreateWindow ( "Example of vertex and fragment assembly shaders" );  
  
    // register handlers  
    glutDisplayFunc ( display );  
    glutReshapeFunc ( reshape );  
    glutKeyboardFunc ( key );  
    glutMouseFunc ( mouse );  
    glutMotionFunc ( motion );  
    glutIdleFunc ( animate );  
  
    init ();  
    initExtensions ();  
  
    assertExtensionsSupported ( "GL_ARB_vertex_program"  
                                "GL_ARB_fragment_program" );  
  
    decalMap = createTexture2D ( true, "../..../Textures/wood1.bmp" );  
  
    if ( !vertexProgram.load ( "example.vp" ) )  
    {  
        printf ( "Error loading vertex program:\n %s\n",  
                 vertexProgram.getErrorMessage ().c_str () );  
  
        return 1;  
    }  
  
    if ( !fragmentProgram.load ( "example.fp" ) )  
    {  
        printf ( "Error loading fragment program:\n %s\n",  
                 fragmentProgram.getErrorMessage ().c_str () );  
    }  
}
```

```
    return 2;
}

glutMainLoop ();

return 0;
}
```

Более подробно о создании и использовании вершинных и фрагментных программ можно прочитать в [1].

Появление вершинных и фрагментных программ заметно расширило возможности OpenGL, однако написание подобных программ затруднялось необходимостью использования языка низкого уровня (т. е. ассемблера). Возникла потребность в языке высокого уровня, на котором можно было бы легко писать вершинные и фрагментные программы. Наиболее успешным образом для такого языка был так называемый RenderMan Shading Language — С-подобный язык, многие годы успешно используемый для написания шейдеров в пакете RenderMan. Однако непосредственно использование языка RenderMan Shading Language было затруднено как отличием моделей рендеринга пакета RenderMan и OpenGL, так и тем, что язык RenderMan Shading Language никогда не был ориентирован на рендеринг в реальном времени с использованием графических ускорителей.

Первой успешной попыткой создания языка высокого уровня для написания вершинных и фрагментных программ стал язык Cg (C for graphics), разработанный компанией NVIDIA в 2002 году (на данный момент доступна версия 1.3 этого языка) [16, 29]. Язык Cg сильно напоминает как язык C, так и RenderMan Shading Language, поэтому писать программы на нем довольно просто. С другой стороны, язык Cg ориентирован на написание вершинных и фрагментных программ (шейдеров) для современных графических ускорителей. Более того, программы на Cg можно использовать не только вместе с OpenGL, но и вместе с Direct3D.

Язык Cg поддерживает векторные (float2, float3 и float4) и матричные (float2x2, float3x3 и float4x4) типы данных. Также программам на Cg доступна большая библиотека функций.

Язык обладает мощными управляющими средствами — поддерживаются условный оператор и различные операторы цикла, введение функций. При этом поддержка Cg реализуется через компиляцию Cg-программы в подходящие для GPU команды. С этой целью вводится понятие так называемых профилей (profile).

Профиль определяет, во что именно (т. е. в какой набор команд) будет компилироваться данная программа на Cg. Правильный выбор профиля позво-

ляет в полной степени использовать возможности имеющегося графического ускорителя.

В табл. 2.3 приведены поддерживаемые вершинные и фрагментные профили.

Таблица 2.3. Профили, поддерживаемые Cg

Профиль	Тип	Используемые расширения
CG_PROFILE_ARBVP1	Вершинный	ARB_vertex_program
CG_PROFILE_ARBFNP1	Фрагментный	ARB_framemnt_program
CG_PROFILE_VP20	Вершинный	NV_vertex_program
CG_PROFILE_FP20	Фрагментный	NV_texture_shader и NV_register_combiners
CG_PROFILE_VP30	Вершинный	NV_vertex_program2
CG_PROFILE_FP30	Фрагментный	NV_fragment_program

В следующих двух листингах (листинг 2.4 и 2.5) приводятся аналоги вершинной и фрагментной программ из листингов 2.1 и 2.2, реализованные на языке Cg.

Листинг 2.4. Вершинная программа на Cg

```
struct InData
{
    float4 pos      : POSITION;
    float4 normal   : NORMAL;
    float2 texCoord : TEXCOORD0;
};

struct OutData
{
    float4 pos      : POSITION;
    float2 texCoord : TEXCOORD0;
    float diffuse;
    float specular;
};

OutData main ( InData IN,
               uniform float4x4 ModelViewProj,
               uniform float4x4 ModelViewIT,
```

```

        uniform float4x4 ModelView,
        uniform float4    lightPos,
        uniform float4    eyePos )
{

OutData OUT;

OUT.pos      = mul ( ModelViewProj, IN.pos );
OUT.texCoord = IN.texCoord;

float4 p = mul ( ModelView, IN.pos );
float3 l = normalize ( lightPos.xyz - p.xyz );
float3 v = normalize ( eyePos.xyz - p.xyz );
float3 h = normalize ( l + v );
float3 n = normalize ( mul ( ModelViewIT, IN.normal ).xyz );

OUT.diffuse = max ( 0.0, dot ( n, l ) );
OUT.specular = pow ( max ( 0.0, dot ( n, h ) ), 50.0 );

return OUT;
}

```

Листинг 2.5. Фрагментная программа на Cg

```

struct OutData
{
    float4 pos      : POSITION;
    float2 texCoord : TEXCOORD0;
    float diffuse;
    float specular;
};

float4 main ( OutData IN,
              uniform sampler2D tex0 ) : COLOR
{
    return (IN.diffuse + 0.4) * tex2D ( tex0, IN.texCoord ) +
           IN.specular * float4 ( 0, 0, 1, 1 );
}

```

Примечание

В каждой из программ функция `main` является точкой входа (т. е. именно она вызывается для обработки данных).

Наборы входных и выходных данных передаются как структура (на самом деле все входные данные можно передавать и по отдельности, но это делает заголовок функции слишком громоздким).

Обратите внимание на описатели `POSITION`, `NORMAL`, `TEXCOORD0` — они фактически задают привязку соответствующего входного параметра к тому или иному атрибуту вершины (или фрагмента).

Ряд величин передается с описателем `uniform`. Он обозначает, что данная величина остается неизменной в пределах примитива; очевидно, что модельная матрица постоянна для каждого отдельного выводимого примитива.

Также обратите внимание на использование типа `sampler2D` — для доступа к текстурам используются специальные типы данных (`sampler1D`, `sampler2D`, `sampler3D` и `samplerCUBE`).

Для доступа к текстуре используются функции `tex1D`, `tex1Dproj`, `tex2D`, `tex2Dproj`, `tex3D`, `tex3Dproj`, `texCUBE` и `texCUBEproj`. Сuffixикс `proj` в имени функции означает, что данная функция перед непосредственным обращением к текстуре осуществляет деление текстурных координат на последнюю координату.

Рассмотрим функцию `tex2Dproj`:

```
float4 tex2Dproj ( sampler2D tex, float3 sq );
```

При ее использовании сперва осуществляется деление значений первых двух компонент вектора `sq` на значение его последней компоненты, результаты деления (`sq.x/sq.z`, `sq.y/sq.z`) используются для доступа к текстуре `tex`.

Примечание

Размеры программ на языке Cg существенно меньше программ на ассемблере, хотя они выполняют точно то же самое. При этом листинги программ на Cg гораздо более понятны, чем ассемблерные.

Рассмотрим теперь, что нужно для практического использования Cg вместе с OpenGL.

В первую очередь потребуется набор средств Cg Toolkit для соответствующей платформы (он доступен как для Windows, так и для Linux), который можно скачать с сайта компании NVIDIA (<http://developer.nvidia.com>).

В программу необходимо включить заголовочный файл `cgL.h` и при сборке программы подключить библиотеки `cglib` и `cgglib` (для платформы Linux следует подключить библиотеки `Cg`, `CgGL` и `pthread`).

Для использования Cg следует создать *контекст* — переменную типа CGcontext. Контекст фактически является контейнером для всех загруженных Cg-программ, поэтому достаточно иметь всего один Cg-контекст на всю программу.

Каждая программа на Cg связывается с переменной типа CGprogram, используемой для работы с этой программой. Кроме того, необходимы два профиля — один для всех вершинных программ и один для всех фрагментных.

```
CGcontext context;
CGprogram vertexProgram;
CGprogram fragmentProgram;
CGprofile vertexProfile;
CGprofile fragmentProfile;
```

Первым шагом (естественно, после инициализации OpenGL) является создание контекста Cg:

```
context = cgCreateContext ();

// check for success
if ( context == NULL )
{
    fprintf ( stderr, "Cannot create Cg context\n" );
    exit ( 1 );
}
```

Вместо явного задания профилей можно воспользоваться функцией cgGLGetLastProfile для получения подходящего профиля:

```
vertexProfile = cgGLGetLatestProfile ( CG_GL_VERTEX );

if ( vertexProfile == CG_PROFILE_UNKNOWN )
{
    fprintf ( stderr, "Invalid vertex profile type" );
    exit ( 1 );
}
```

```
cgGLSetOptimalOptions ( vertexProfile );
```

Функция cgGLGetOptimalOptions устанавливает опции компиляции, наиболее полно соответствующие найденному графическому ускорителю и его драйверу.

Для загрузки программы на Cg из файла предназначена функция `cgCreateProgramFromFile`:

```
CGprogram cgCreateProgramFromFile ( CGcontext context,
                                    CGenum programType,
                                    const char* program,
                                    CGprofile profile,
                                    const char* entry,
                                    const char** args );
```

Параметры: `context` — контекст Cg, к которому будет подключена данная программа; `programType` — определяет содержимое файла (`CG_SOURCE` — исходная программа, `CG_OBJEST` — объектный код предкомпиляции программы); `program` — имя файла, содержащего саму программу; `profile` — профиль, который следует использовать при компиляции данной программы; `entry` — задает точку входа в программу, т. е. функцию, которая будет вызываться для обработки вершины (или фрагмента); `args` — позволяет передать дополнительные опции компилятору (если таких опций нет, то следует передать значение `NULL`).

Данная функция возвращает ссылку на откомпилированную программу или `NULL` в случае ошибки.

Для загрузки программы не из файла, а напрямую из памяти предназначена функция `cgCreateProgram`:

```
CGprogram cgCreateProgram ( CGcontext context, CGenum programType,
                            const char* program, CGprofile profile,
                            const char* entry, const char** args );
```

Смысл параметров этой функции точно такой же, как у функции `cgCreateProgramFromFile`, только параметр `program` — это строка, содержащая текст программы.

При помощи функции `cgGetProgramString` можно получать различную информацию о программе:

```
const char* cgGetProgramString ( CGprogram program, CGenum stringType );
```

Наибольший интерес представляет значение параметра `stringType`, равное `CG_COMPILED_PROGRAM`. В этом случае возвращается откомпилированный текст программы.

В случае ошибки можно обратиться к функции `cgGetError` для получения кода ошибки. Функция `cgGetErrorString` по этому коду возвращает описание ошибки.

В листинге 2.6 приведен пример получения полной информации об ошибке.

Листинг 2.6. Проверка ошибок при использовании Cg

```

void checkCgError ()
{
    CGerror error = cgGetError();

    if ( error != CG_NO_ERROR )
    {
        fprintf ( stderr, "CG error: %s\n",
                  cgGetErrorString ( error ) );

        exit ( 1 );
    }
}

```

Следующим шагом является загрузка программы при помощи функции `cgGLLoadProgram`:

```

cgGLLoadProgram ( vertexProgram );
cgGLLoadProgram ( fragmentProgram );

```

После этого можно воспользоваться функцией `cgGetNamedParameter` для получения ссылок на входные параметры программ по их именам:

`CGparameter cgGetNamedParameter (CGprogram program, const char * name);`
С помощью полученных значений типа `CGparameter` можно устанавливать значения `uniform`-параметров при помощи следующих функций:

```

void cgGLSetParameter1f ( CGparameter parameter, float x );
void cgGLSetParameter1fv( CGparameter parameter, const float* array );
void cgGLSetParameter2f ( CGparameter parameter, float x, float y );
void cgGLSetParameter2fv( CGparameter parameter, const float* array );
void cgGLSetParameter3f ( CGparameter parameter, float x, float y,
                        float z );
void cgGLSetParameter3fv( CGparameter parameter, const float* array );
void cgGLSetParameter4f ( CGparameter parameter, float x, float y,
                        float z, float w );
void cgGLSetParameter4fv( CGparameter parameter, const float* array );

```

Для задания матричных параметров предназначены следующие функции:

```

void cgGLSetMatrixParameterfr ( CGparameter parameter,
                               const float* matrix);
void cgGLSetMatrixParameterfc ( CGparameter parameter,
                               const float* matrix);

```

Суффиксы *r* и *c* в именах функций определяют способ хранения матрицы — по строкам (*r*) или по столбцам (*c*).

Для задания матриц, являющимися параметрами состояния OpenGL, предназначена функция `cgGLSetStateMatrixParameter`:

```
void cgGLSetStateMatrixParameter ( CGparameter parameter,
                                    GLenum stateMatrixType,
                                    GLenum transform );
```

Параметры: `stateMatrixType` — задает матрицу, которую следует поместить в параметр `parameter` (табл. 2.4); `transform` — задает преобразование, которое следует применить к матрице перед записью в параметр `param` (табл. 2.5).

Таблица 2.4. Возможные значения параметра `stateMatrixType`

Значение	Комментарий
<code>CG_GL_MODELVIEW_MATRIX</code>	Матрица модельного преобразования
<code>CG_GL_PROJECTION_MATRIX</code>	Матрица проектирования
<code>CG_GL_TEXTURE_MATRIX</code>	Матрица преобразования текстурных координат
<code>CG_GL_MODELVIEW_PROJECTION_MATRIX</code>	Произведение матрицы модельного преобразования и матрицы проектирования

Таблица 2.5. Возможные значения параметра `transform`

Значение	Комментарий
<code>CG_GL_MATRIX_IDENTITY</code>	Матрица не преобразуется
<code>CG_GL_MATRIX_TRANSPOSE</code>	Матрица транспонируется
<code>CG_GL_MATRIX_INVERSE</code>	Матрица обращается
<code>CG_GL_MATRIX_INVERSE_TRANSPOSE</code>	Матрица транспонируется и обращается

Для непосредственного использования программы следует разрешить соответствующий профиль и выбрать необходимую программу в качестве текущей:

```
cgGLEnableProfile ( vertexProfile );
```

```
cgGLBindProgram ( vertexProgram );
```

После этого можно задать значения необходимых переменных и осуществить вывод объекта с использованием данной программы. По окончании вывода следует запретить данный профиль:

```
cgGLDisableProfile ( vertexProfile );
```

В листинге 2.7 приведен исходный текст на C++ программы, строящей изображение с использованием вершинной и фрагментной программ на Cg (см. листинги 2.4 и 2.5).

Листинг 2.7. Использование шейдеров, написанных на языке Cg

```
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    <Cg/cgGL.h>

#include    "libTexture.h"
#include    "TypeDefs.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Vector4D.h"

CGcontext context;
CGprofile vertexProfile;      // profile for vertex shader
CGprofile fragmentProfile;   // profile for fragment shader
CGprogram vertexProgram;     // handles for programs
CGprogram fragmentProgram;
CGparameter modelViewProjMatrix; // handles for parameters
CGparameter modelViewMatrix;
CGparameter modelViewItMatrix;
CGparameter lightPos, eyePos;
CGparameter tex0;

Vector3D eye  ( 7, 5, 7 );    // camera position
Vector3D light ( 5, 0, 4 );    // light position
Vector3D rot ( 0, 0, 0 );
float     angle = 0;
int      mouseOldX = 0;
int      mouseOldY = 0;
unsigned decalMap;

void checkCgError ()
{
    CGerror error = cgGetError();
```

```
if ( error != CG_NO_ERROR )
{
    fprintf ( stderr, "CG error: %s\n",
              cgGetErrorString ( error ) );

    exit(1);
}

void cgErrorCallback ()
{
    CGerror lastError = cgGetError ();

    if ( lastError )
    {
        fprintf ( stderr, "CG error: %s\n",
                  - cgGetErrorString ( lastError ) );
        exit ( 1 );
    }
}

void init ()
{
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );
    glEnable      ( GL_DEPTH_TEST );
    glEnable      ( GL_TEXTURE_2D );
    glDepthFunc   ( GL_LEQUAL      );

    glHint       ( GL_POLYGON_SMOOTH_HINT,           GL_NICEST );
    glHint       ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );

    // now setup Cg
    // create context
    context = cgCreateContext ();

    // check for success
    if ( context == NULL )
    {
        fprintf ( stderr, "Cannot create Cg context\n" );
    }
}
```

```
    exit ( 1 );
}

        // create vertex profile
vertexProfile = cgGLGetLatestProfile ( CG_GL_VERTEX );

if ( vertexProfile == CG_PROFILE_UNKNOWN )
{
    fprintf ( stderr, "Invalid vertex profile type" );

    exit ( 1 );
}

cgGLSetOptimalOptions ( vertexProfile );

        // create fragment profile
fragmentProfile = cgGLGetLatestProfile ( CG_GL_FRAGMENT );

if ( fragmentProfile == CG_PROFILE_UNKNOWN )
{
    fprintf ( stderr, "Invalid fragment profile type" );

    exit ( 1 );
}

cgGLSetOptimalOptions ( fragmentProfile );

        // now load cg programs
vertexProgram = cgCreateProgramFromFile ( context, CG_SOURCE,
                                         "vertex.cg", vertexProfile, "main", 0 );

checkCgError ();

if ( vertexProgram == NULL )
{
    CGerror error = cgGetError ();

    fprintf ( stderr, "Error loading vertex program:\n%s",
              cgGetString ( error ) );
}
```



```

        CG_GL_MATRIX_IDENTITY );

cgGLSetStateMatrixParameter ( modelViewMatrix,
                            CG_GL_MODELVIEW_MATRIX,
                            CG_GL_MATRIX_IDENTITY );

cgGLSetParameter4f ( lightPos, light.x, light.y, light.z, 1 );
cgGLSetParameter4f ( eyePos, eye.x, eye.y, eye.z, 1 );

glBindTexture ( GL_TEXTURE_2D, decalMap );

glutSolidTeapot ( 1.5 );

cgGLDisableProfile ( vertexProfile );
cgGLDisableProfile ( fragmentProfile );

glPopMatrix ();

glutSwapBuffers ();
}

void reshape ( int w, int h )
{
    glViewport      ( 0, 0, (GLsizei)w, (GLsizei)h );
    glMatrixMode   ( GL_PROJECTION );
        // factor all camera ops into
        // projection matrix
    glLoadIdentity ();
    gluPerspective ( 60.0, (GLfloat)w/(GLfloat)h, 1.0, 60.0 );
    gluLookAt ( eye.x, eye.y, eye.z, // eye
                0, 0, 0,           // center
                0, 0, 1 );         // up

    glMatrixMode   ( GL_MODELVIEW );
    glLoadIdentity ();
}

void motion ( int x, int y )
{

```

```
    rot.y -= ((mouseOldY - y) * 180.0f) / 200.0f;
    rot.z -= ((mouseOldX - x) * 180.0f) / 200.0f;
    rot.x = 0;

    if ( rot.z > 360 )
        rot.z -= 360;

    if ( rot.z < -360 )
        rot.z += 360;

    if ( rot.y > 360 )
        rot.y -= 360;

    if ( rot.y < -360 )
        rot.y += 360;

    mouseOldX = x;
    mouseOldY = y;

    glutPostRedisplay ();
}

void mouse ( int button, int state, int x, int y )
{
    if ( state == GLUT_DOWN )
    {
        mouseOldX = x;
        mouseOldY = y;
    }
}

void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' ) // quit requested
        exit ( 0 );
}

void animate ()
{
```

```
angle = 0.004f * glutGet ( GLUT_ELAPSED_TIME ) ;

light.x = 4*cos ( angle );
light.y = 4*sin ( angle );
light.z = 3 + 0.3 * sin ( angle / 3 );

cgGLEnableProfile ( vertexProfile );
cgGLBindProgram ( vertexProgram );
cgGLSetParameter4f ( lightPos, light.x, light.y, light.z, 1 );
cgGLSetParameter4f ( eyePos, eye.x, eye.y, eye.z, 1 );
cgGLDisableProfile ( vertexProfile );

glutPostRedisplay ();
}

int main ( int argc, char * argv [] )
{
    // initialize glut
    glutInit ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize ( 500, 500 );

    // create window
    glutCreateWindow ( "Example of Cg specular vertex shader" );

    // register handlers
    glutDisplayFunc ( display );
    glutReshapeFunc ( reshape );
    glutKeyboardFunc ( key );
    glutMouseFunc ( mouse );
    glutMotionFunc ( motion );
    glutIdleFunc ( animate );

    init ();

    decalMap = createTexture2D ( true, "../../../Textures/wood1.bmp" );

    // install program object as
```

```
// part of current state
cgGLEnableProfile      ( fragmentProfile );
cgGLBindProgram         ( fragmentProgram );
cgGLSetTextureParameter ( tex0, decalMap );
cgGLDisableProfile      ( fragmentProfile );

glutMainLoop ();

return 0;
}
```

Для удобства работы с программами на Сg удобно создать специальный класс `CgProgram`, инкапсулирующий всю работу с Cg-программой — загрузку, подключение, задание параметров и т. п.

В следующих листингах приведены описание (листинг 2.8) и реализация (листинг 2.9) класса `CgProgram`.

Листинг 2.8. Описание класса `CgProgram`

```
class CgProgram
{
protected:
    static CGcontext context;
    static CGprofile vertexProfile;
    static CGprofile fragmentProfile;

    CGprogram program;
    int type;
    string listing;
    string errorDesc;

public:
    CgProgram ( int theType );
    ~CgProgram ();

    bool load ( const char * fileName );

    const string& getListing () const
    {
        return listing;
```

```
}

const string& getErrorDescription () const
{
    return errorDesc;
}

bool isVertex () const
{
    return type == CgVertexProgram;
}

bool isFragment () const
{
    return type == CgFragmentProgram;
}

CGprofile getProfile () const
{
    return isVertex () ? vertexProfile : fragmentProfile;
}

CGparameter parameterForName ( const char * name ) const
{
    return cgGetNamedParameter ( program, name );
}

bool setTexture ( const char * name, unsigned texId )
{
    CGparameter param = parameterForName ( name );

    if ( !param )
        return false;

    cgGLSetTextureParameter ( param, texId );

    return true;
}

bool setVector ( const char * name, const Vector4D& value )
```

```
{  
    CGparameter param = parameterForName ( name );  
  
    if ( !param )  
        return false;  
  
    cgGLSetParameter4f ( param, value.x, value.y, value.z,  
                        value.w );  
  
    return true;  
}  
  
bool setVector ( CGparameter param, const Vector4D& value )  
{  
    if ( !param )  
        return false;  
  
    cgGLSetParameter4f ( param, value.x, value.y, value.z,  
                        value.w );  
  
    return true;  
}  
  
bool enable ()  
{  
    cgGLEnableProfile ( getProfile () );  
  
    return true;  
}  
  
bool disable ()  
{  
    cgGLDisableProfile ( getProfile () );  
  
    return true;  
}  
  
bool bind ()  
{
```

```

    cgGLBindProgram ( program );

    return true;
}

bool unbind ()
{
    cgGLBindProgram ( 0 );

    return true;
}

void setMatrixParameter ( CGparameter param, CGGLenum matrix,
                           CGGLenum transform )
{
    cgGLSetStateMatrixParameter ( param, matrix, transform );
}

enum CgProgramType
{
    CgVertexProgram = 1,
    CgFragmentProgram = 2
};

protected:
    bool checkCgError ();
};

```

Листинг 2.9. Реализация класса CgProgram

```

// 
// Object wrapper for Cg programs
//

#include "CgProgram.h"

CGcontext CgProgram :: context      = NULL;
CGprofile CgProgram :: vertexProfile = CG_PROFILE_UNKNOWN;

```

```
CGprofile CgProgram :: fragmentProfile = CG_PROFILE_UNKNOWN;

CgProgram :: CgProgram ( int theType )
{
    type = theType;
}

CgProgram :: ~CgProgram ()
{
    cgDestroyProgram ( program );
}

bool CgProgram :: load ( const char * fileName )
{
    // create context
    if ( context == NULL )
        context = cgCreateContext ();

    // check for success
    if ( context == NULL )
    {
        errorDesc = "Cannot create context";

        return false;
    }

    // create vertex profile
    if ( type == CgVertexProgram )
    {
        if ( vertexProfile == CG_PROFILE_UNKNOWN )
        {
            vertexProfile = cgGLGetLatestProfile ( CG_GL_VERTEX );

            if ( vertexProfile == CG_PROFILE_UNKNOWN )
            {
                errorDesc = "Invalid vertex profile type";

                return false;
            }
        }
    }
}
```

```
    cgGLSetOptimalOptions ( vertexProfile );
}

}

else
if ( type == CgFragmentProgram )
{
    // create fragment profile
    if ( fragmentProfile == CG_PROFILE_UNKNOWN )
    {
        fragmentProfile = cgGLGetLatestProfile ( CG_GL_FRAGMENT );

        if ( fragmentProfile == CG_PROFILE_UNKNOWN )
        {
            errorDesc = "Invalid fragment profile type";

            return false;
        }
    }

    cgGLSetOptimalOptions ( fragmentProfile );
}
}

else
{
    errorDesc = "Unknown profile type";

    return false;
}

program = cgCreateProgramFromFile ( context, CG_SOURCE, fileName,
                                    getProfile (), "main", 0 );

if ( !checkCgError () )
    return false;

if ( program == NULL )
{
    CGerror error = cgGetError ();

    errorDesc = "Error loading vertex program:\n";
    errorDesc += cgGetErrorString ( error );
}
```

```
    return false;
}

listing = cgGetProgramString ( program, CG_COMPILED_PROGRAM );

cgGLLoadProgram ( program );

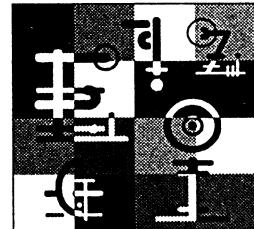
return checkCgError ();
}

bool CgProgram :: checkCgError ()
{
    CGerror error = cgGetError();

    if ( error != CG_NO_ERROR )
        errorDesc = cgGetErrorString ( error );
    else
        errorDesc = "";

    return error == CG_NO_ERROR;
}
```

Глава 3



Появление OpenGL 2.0 и GLSL

Версия OpenGL 2.0 вышла 7 сентября 2004 года. Основным ее новшеством (в частности, определившим номер версии — 2.0, а не 1.6) стало введение программируемого конвейера рендеринга, реализуемого посредством шейдеров на OpenGL Shading Language (далее мы будем употреблять сокращение GLSL).

На рис. 3.1 показано место вершинного и фрагментного процессоров (выполняющих шейдеры) в общем конвейере рендеринга OpenGL.

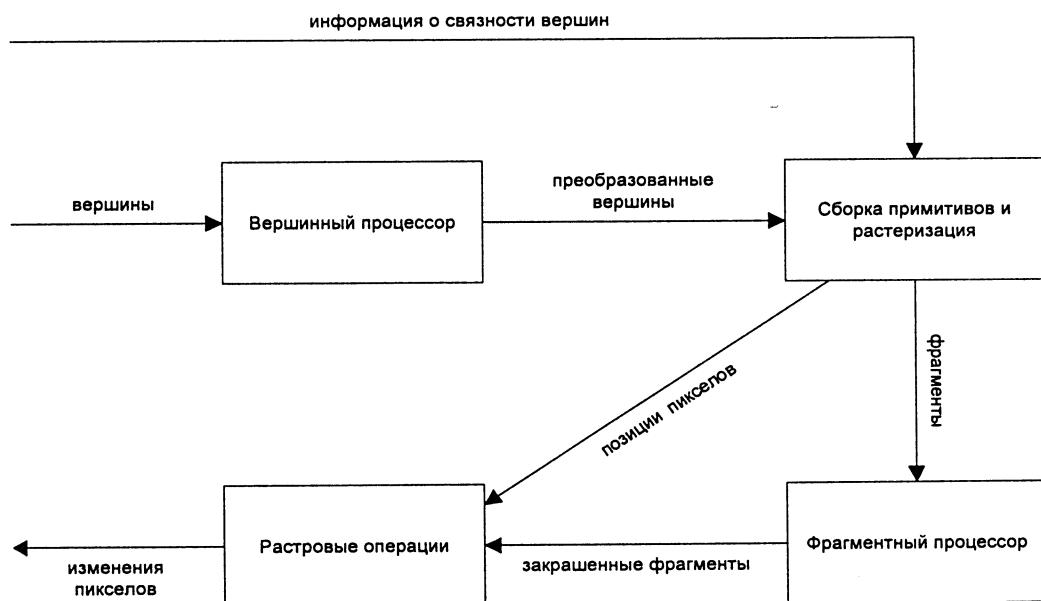


Рис. 3.1. Место шейдеров в конвейере рендеринга OpenGL

В эту версию из расширений `ARB_shader_objects`, `ARB_vertex_shader`, `ARB_fragment_shader` и `ARB_shading_language_100` перешел весь механизм (API) создания и использования шейдеров на GLSL.

Как в данных расширениях, так и в OpenGL 2.0 для работы с шейдерами используются (по аналогии с текстурами, VBO и т. п.) так называемые *объекты-шейдеры* (*shader objects*). Вся работа с шейдерами в OpenGL 2.0 осуществляется через соответствующие объекты-шейдеры.

В OpenGL 2.0 (как и в соответствующих расширениях) поддерживаются два типа шейдеров — вершинные (`vertex`) и фрагментные (`fragment`). Все они пишутся на GLSL, при этом поддержка вершинных шейдеров взята из расширения `ARB_vertex_shader`, поддержка фрагментных шейдеров — из расширения `ARB_fragment_shader`.

Примеры вершинного и фрагментного шейдеров, написанных на GLSL, приведены в листингах 3.1 и 3.2.

Листинг 3.1. Пример вершинного шейдера на GLSL

```
//  
// Simplest GLSL vertex shader  
//  
  
void main (void)  
{  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

Листинг 3.2. Пример фрагментного шейдера на GLSL

```
//  
// Simplest GLSL fragment shader  
//  
  
void main (void)  
{  
    gl_FragColor = vec4 ( 0.0, 1.0, 0.0, 1.0 );  
}
```

Как видно из приведенных листингов, OpenGL Shading Language представляет собой высокоуровневый язык, близкий по синтаксису к языку C (а также

к RenderMan Shading Language) и предназначенный для программирования вершинного и фрагментного конвейеров.

Программам на этом языке доступны как различные параметры состояния OpenGL, так и большой набор функций (подробная информация приведена в гл. 12 и 13).

В процессе переноса API для поддержки шейдеров из расширений в состав OpenGL 2.0 были сделаны некоторые изменения (табл. 3.1), не затрагивающие функциональность API и, скорее, представляющие собой незначительные изменения в именах отдельных функций.

Кроме того, вместо вводимого расширениями типа `GLhandleARB`, используемого для обращения к объектам-шейдерам, было решено использовать уже существующий (и используемый в ряде похожих мест) тип `GLuint`.

Таблица 3.1. Соответствие имен функций поддержки шейдеров

Функция расширения	Соответствующая функция OpenGL 2.0
<code>glCreateShaderObjectARB</code>	<code>glCreateShader</code>
<code>glShaderSourceARB</code>	<code>glShaderSource</code>
<code>glCompileShaderARB</code>	<code>glCompileShader</code>
<code>glDeleteShaderARB</code>	<code>glDeleteShader</code>
<code>glCreateProgramObjectARB</code>	<code>glCreateProgram</code>
<code>glAttachObjectARB</code>	<code>glAttachShader</code>
<code>glDetachObjectARB</code>	<code>glDetachShader</code>
<code>glLinkProgramARB</code>	<code>glLinkProgram</code>
<code>glUseProgramObjectARB</code>	<code>glUseProgram</code>
<code>glDeleteObjectARB</code>	<code>glDeleteProgram</code>
<code>glGetActiveAttribARB</code>	<code>glGetActiveAttrib</code>
<code>glGetAttribLocationARB</code>	<code>glGetAttribLocation</code>
<code>glBindAttribLocationARB</code>	<code>glBindAttribLocation</code>
<code>glGetUniformLocationARB</code>	<code>glGetUniformLocation</code>
<code>glGetActiveUniformARB</code>	<code>glGetActiveUniform</code>
<code>glUniform{1234}{if}vARB</code>	<code>glUniform{1234}{if}v</code>
<code>glUniformMatrix{234}{f}vARB</code>	<code>glUniformMatrix{234}{f}v</code>
<code>glValidateProgramARB</code>	<code>glValidateProgram</code>
<code>glGetObjectParameterivARB</code>	<code>glGetShaderiv</code>
<code>glGetInfoLogARB</code>	<code>glGetShaderInfoLog</code>
<code>glGetAttachedObjectsARB</code>	<code>glGetAttachedShaders</code>
<code>glGetShaderSourceARB</code>	<code>glGetShaderSource</code>

Помимо поддержки шейдеров в OpenGL 2.0 вошел ряд небольших, но полезных добавлений.

Из расширения ARB_draw_buffers в состав OpenGL 2.0 была перенесена возможность для шейдеров за один проход осуществлять вывод сразу в несколько различных цветовых буферов (в том числе, различных значений цвета).

Также именно в этой версии было наконец снято ограничение на то, что все размеры текстур обязательно должны быть степенями двух. В этой версии стало возможно использовать текстуры произвольных размеров, при этом работа с такими текстурами полностью эквивалентна работе с текстурами, чьи размеры являются степенями двух (в частности, все компоненты текстурных координат принимают значения на отрезке [0, 1]). Данная возможность пришла из расширения ARB_texture_non_power_of_two [1].

Из расширения ARB_point_sprite [1] была взята поддержка спрайтов (квадратов с "натянутой на них" текстурой, всегда параллельных экрану) (листинг 3.3). Кроме этого была добавлена возможность управлять направлением, по которому происходит возрастание текстурных координат (через параметр GL_POINT_SPRITE_COORD_ORIGIN).

Листинг 3.3. Пример задания спрайта

```
float quadratic [] = { 1.0f, 0.0f, 0.01f };

glEnable ( GL_POINT_SPRITE_ARB );
glPointParameterfvARB ( GL_POINT_DISTANCE_ATTENUATION_ARB, quadratic );
glPointParameterfARB ( GL_POINT_FADE_THRESHOLD_SIZE_ARB, 20.0f );

glTexEnvf ( GL_POINT_SPRITE_ARB, GL_COORD_REPLACE_ARB, GL_TRUE );
glPointSize ( size );
```

Из расширений ATI_separate_stencil и EXT_stencil_two_side была взята возможность по-разному задавать работу с буфером трафарета отдельно для лицевых и нелицевых граней (листинг 3.4).

Листинг 3.4. Пример раздельного задания работы с буфером трафарета для лицевых и нелицевых граней

```
glActiveStencilFaceEXT ( GL_BACK ); // setup stencil ops for
// back faces
glStencilMask ( ~0 );
```

```
glStencilFunc      ( GL_ALWAYS, 0, ~0 );

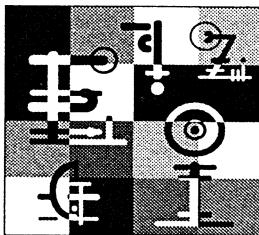
glStencilOp        ( GL_KEEP,           // stencil test failed
                     GL_INCR_WRAP_EXT, // depth test failed
                     GL_KEEP );        // depth test passed

                     // setup stencil for
                     // front faces
glActiveStencilFaceEXT ( GL_FRONT );
glStencilMask       ( ~0 );
glStencilFunc      ( GL_ALWAYS, 0, ~0 );

glStencilOp        ( GL_KEEP,           // stencil test fail
                     GL_DECR_WRAP_EXT, // depth test fail
                     GL_KEEP );        // depth test pass

                     // process all shadow
                     // faces at once
drawShadowVolume ( torus.getFaces (), torus.getNumFaces (),
                   torus.getEdges (), torus.getNumEdges (),
                   torus.getVertices (), torus.getNumVertices () );
```

На данный момент последние версии драйверов как от компании NVIDIA, так и от компании ATI обеспечивают поддержку OpenGL 2.0.



Часть II

Дополнительные библиотеки

Глава 4. Библиотека GLUT

**Глава 5. Общие сведения о библиотеках GLH,
NV_MATH и NV_UTIL**

**Глава 6. Основные классы для работы
с векторами, матрицами и кватернионами**

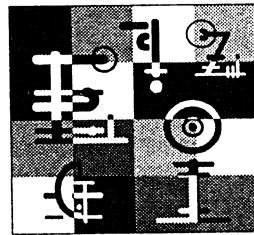
**Глава 7. Работа с расширениями,
библиотека libExt**

Глава 8. Библиотеки libTexture и libTexture3D

**Глава 9. Реализация р-буфера в виде класса,
использование расширения
EXT_framebuffer_object
для создания внеэкранных буферов**

**Глава 10. Понятие камеры. Работа
с библиотекой libCamera**

**Глава 11. Библиотека libMesh. Работа
с полигональными моделями
и их загрузка из популярных форматов**



Глава 4

Библиотека GLUT

Одним из важнейших преимуществ библиотеки OpenGL является полное отделение модели рендеринга от специфики конкретной оконной (и операционной) системы.

Таким образом, целый ряд важных для функционирования приложения операций, связанных с оконной системой (таких как создание и уничтожение окон, изменение параметров окон и обработка сообщений) оказался полностью вынесенным за пределы OpenGL. С одной стороны, это позволило обеспечить полную переносимость кода, непосредственно использующего OpenGL. Однако при этом для получения полноценного приложения программист вынужден влезать в детали конкретной оконной системы (сильно отличающихся для разных платформ).

Задачей библиотеки GLUT (OpenGL Utility Toolkit) является обеспечение унифицированного (т. е. независимого от конкретной операционной/оконной системы) интерфейса к основным возможностям и функциям оконной системы.

Целью создания библиотеки GLUT было, прежде всего, обеспечение крайне простого и удобного интерфейса к оконной системе, поэтому она поддерживает доступ только к важнейшим функциям оконной системы.

Основу интерфейса прикладного программирования (API, Application Program Interface) библиотеки GLUT составляет довольно небольшое число функций, каждая из которых требует небольшого числа параметров. Функции GLUT никогда не возвращают указатели и единственный случай, когда функции принимают адрес в качестве параметра, это передача адресов строк (при этом сами строки копируются внутри GLUT) и передача адресов функций-обработчиков сообщений.

Библиотека GLUT поддерживает работу с окнами и меню. При этом есть понятие *текущего окна* (*current window*) и текущего меню (*current menu*). Большинство операций GLUT действуют только на текущее окно (или меню).

Одной из особенностей библиотеки GLUT является то, что в ней реализован свой цикл обработки сообщений; это затрудняет ее совместное использование с другими библиотеками, также реализующими свой цикл обработки сообщений (или требующих создания специального цикла обработки).

Все функции GLUT объединены в несколько классов в соответствии с их функциональностью (рис. 4.1):

- инициализация* — обработка параметров командной строки, инициализация оконной системы и начального состояния окон;
- начало цикла обработки сообщений* — вызов внутреннего цикла обработки сообщений в GLUT;
- работа с окнами* — функции для создания и работы с окнами;
- работа с меню* — функции для создания и работы со всплывающими меню;
- установка обработчиков сообщений*;
- получение информации о текущем состоянии GLUT*;
- вывод текста*;
- вывод простейших геометрических объектов*.

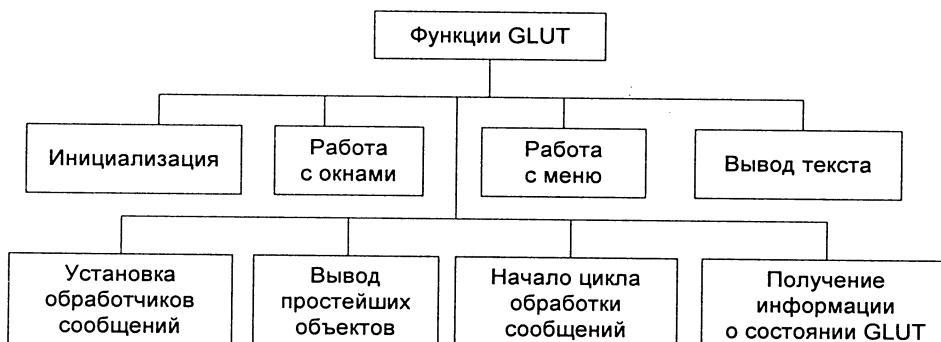


Рис. 4.1. Классы функций GLUT

Инициализация

Имена всех функций инициализации начинаются с префикса `glutInit`. Главной функцией инициализации является функция `glutInit`, которая должна быть вызвана ровно один раз, до всех других вызовов функций библиотеки GLUT.

```
void glutInit ( int * argc, char ** argv );
```

В качестве параметров данная функция принимает указатель на параметр `argc` и параметр `argv` функции `main`.

Примечание

Данная функция может получать параметры из командной строки (это обычно поддерживается в X Window System), в этом случае функция `glutInit` удаляет свои параметры из входных параметров функции `main`, изменяя для этого `argc` и `argv`.

Для задания начальных размеров и положения создаваемых окон предназначены следующие две функции:

```
void glutInitWindowSize ( int width, int height );
```

```
void glutInitWindowPos ( int x, int y );
```

Параметры `width` и `height` задают начальный размер для создаваемых окон (в пикселях), а параметры `x` и `y` — начальное положение верхнего левого угла окна (рис. 4.2).

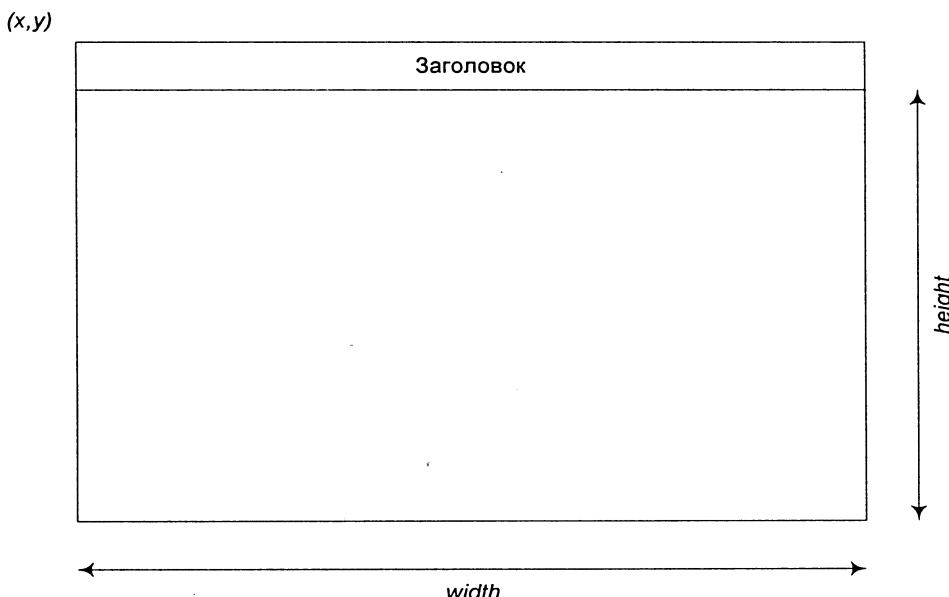


Рис. 4.2. Начальные параметры для создания окон

Еще одним важным начальным параметром является режим, или точнее, строение фреймбуфера. *Режим* определяет, будут ли использоваться буфер глубины, буфер трафарета и другие буфера, будет ли поддерживаться двойная буферизация (ее использование позволяет создавать анимацию без мерцания). Для задания режима предназначена функция `glutInitDisplayMode`:

```
void glutInitDisplayMode ( unsigned int mode );
```

Параметр `mode` является беззнаковым целым числом, отдельные биты которого несут информацию о задаваемом режиме. Этот параметр строится как логическое объединение (операция побитового ИЛИ) из констант, приведенных в табл. 4.1.

Таблица 4.1. Возможные значения битов для параметра `mode`

Константа	Комментарий	Значение по умолчанию
<code>GLUT_RGBA</code>	Выбрать RGBA-режим	Да
<code>GLUT_RGB</code>	Выбрать RGB-режим	Да
<code>GLUT_INDEX</code>	Выбор палитрового режима	Нет
<code>GLUT_SINGLE</code>	Не использовать двойную буферизацию	Да
<code>GLUT_DOUBLE</code>	Использовать двойную буферизацию	Нет
<code>GLUT_ACCUM</code>	Использовать буфер накопления	Нет
<code>GLUT_ALPHA</code>	Отводить в буфере место под альфа-канал	Нет
<code>GLUT_DEPTH</code>	Создавать буфер глубины	Нет
<code>GLUT_STENCIL</code>	Создавать буфер трафарета	Нет
<code>GLUT_MULTISAMPLE</code>	Создавать буфер мультисэмплинга	Нет
<code>GLUT_STEREO</code>	Создавать буфер для стереоизображения	Нет
<code>GLUT_LUMINANCE</code>	Использовать специальный цветовой режим, по функциональности похожий на RGBA, но в буфере цвета отсутствуют зеленый и синий компоненты, а красный компонент содержит индекс для специальной цветовой таблицы. Изначально эта таблица инициализирована значениями серого цвета	Нет

Начало цикла обработки сообщений

Для вызова встроенного цикла обработки сообщений предназначена функция `glutMainLoop` библиотеки GLUT:

```
void glutMainLoop ();
```

Эту функцию обязательно нужно вызывать после инициализации, создания окон, установки обработчиков сообщений. Управление из этой функции никогда не возвращается, однако установленные обработчики сообщений будут вызываться по мере поступления соответствующих сообщений.

Для прерывания выполнения программы следует использовать функцию `exit`, вызываемую из одного из обработчиков сообщений.

Работа с окнами

Библиотека GLUT поддерживает два типа окон — нормальные и дочерние (вложенные в нормальные или в другие дочерние окна) (рис. 4.3). Оба типа поддерживают вывод в них средствами OpenGL и установку обработчиков сообщений. Каждое окно (независимо от его типа) однозначно идентифицируется целым числом.

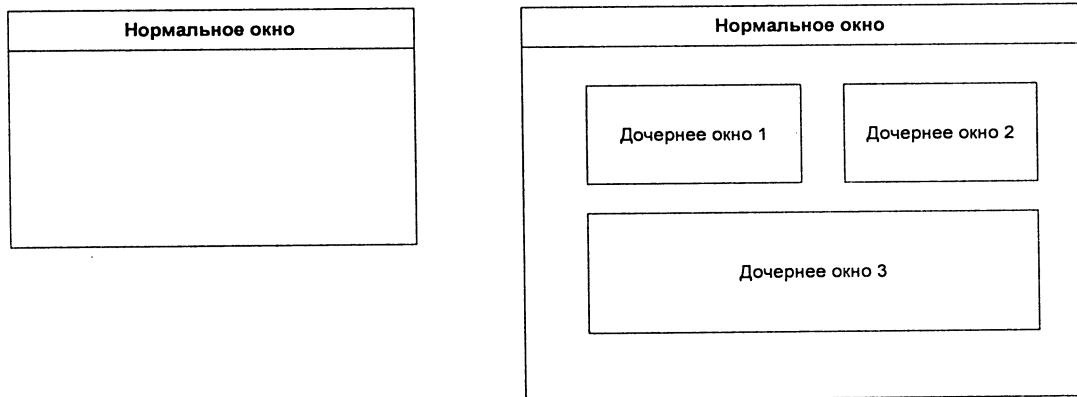


Рис. 4.3. Нормальные и дочерние окна

Каждое создаваемое окно имеет собственный контекст рендеринга для OpenGL.

Для создания нормального (обычного) окна предназначена функция `glutCreateWindow`:

```
int glutCreateWindow ( char * name );
```

Вызов этой функции создает обычное окно с заголовком `name`, положение и размеры окна определяют величины, заданные при помощи функций `glutInitWindowPos` и `glutInitWindowSize`. Созданное окно делается текущим и возвращается его идентификатор.

Для создания дочернего окна предназначена функция `glutCreateSubWindow`:

```
int glutCreateSubWindow ( int win, int x, int y, int width, int height );
```

Параметр `win` является идентификатором родительского окна (окна, в котором будет создано новое окно), а параметры `x`, `y`, `width` и `height` задают положение создаваемого окна относительно родительского и его размеры (все значения задаются в пикселях).

Узнать идентификатор текущего окна можно при помощи функции `glutGetWindow`:

```
int glutGetWindow();
```

Сделать текущим окно с заданным идентификатором `win` можно при помощи функции `glutSetWindow`:

```
void glutSetWindow ( int win );
```

Для уничтожения окна с идентификатором `win` предназначена функция `glutDestroyWindow`:

```
void glutDestroyWindow ( int win );
```

Для того чтобы пометить текущее окно как требующее перерисовки, служит функция `glutPostRedisplay`:

```
void glutPostRedisplay ();
```

В случае использования двойной буферизации сменить буфер (отображаемый в окне на тот, в который осуществляется вывод средствами OpenGL) можно при помощи функции `glutSwapBuffers`:

```
void glutSwapBuffers ();
```

Даже если двойная буферизация не используется, этот вызов все равно можно использовать (таким образом, код не зависит от используемого режима буферизации).

Следующие две функции служат для перемещения текущего окна в точку с координатами (`x`, `y`) и изменения размеров текущего окна на `width`, `height`:

```
void glutPositionWindow ( int x, int y );
```

```
void glutReshapeWindow ( int width, int height );
```

Также библиотека GLUT предоставляет возможность перехода в полноэкранный режим работы при помощи функции `glutFullScreen`:

```
void glutFullScreen ();
```

Вызов этой функции переводит текущее (обязательно нормальное, а не дочернее) окно в полноэкранный режим (способ перевода может зависеть от конкретной оконной системы).

Для перевода окна из полноэкранного в обычный можно воспользоваться любой из функций `glutPositionWindow` или `glutReshapeWindow`.

Также библиотека GLUT предоставляет возможность изменения положения окна по глубине (относительно соседнего). Для этого предназначены следующие две функции:

```
void glutPopWindow ();
```

```
void glutPushWindow ();
```

Следующие функции предназначены соответственно для отображения/скрытия/свертывания текущего окна:

```
void glutShowWindow    ();
void glutHideWindow   ();
void glutIconifyWindow();
```

Можно изменить заголовок текущего окна на строку, задаваемую параметром `name`, при помощи функции `glutSetWindowTitle`:

```
void glutSetWindowTitle ( char * name );
```

Еще библиотека GLUT позволяет задавать форму курсора для текущего окна при помощи функции `glutSetCursor`:

```
void glutSetCursor ( int cursor );
```

Значение параметра `cursor` определяет один из внутренних курсоров GLUT, полный список возможных видов которых приводится в табл. 4.2.

Таблица 4.2. Виды курсоров в GLUT

Константа	Тип курсора
GLUT_CURSOR_RIGHT_ARROW	Стрелка вверх и вправо
GLUT_CURSOR_LEFT_ARROW	Стрелка вверх и влево
GLUT_CURSOR_INFO	Рука
GLUT_CURSOR_DESTROY	Череп с костями
GLUT_CURSOR_HELP	Вопросительный знак
GLUT_CURSOR_CYCLE	Стрелки, врачающиеся по кругу
GLUT_CURSOR_SPRAY	Стрелка в виде распылителя
GLUT_CURSOR_WAIT	Часы
GLUT_CURSOR_TEXT	Текстовый курсор
GLUT_CURSOR_CROSSHAIR	Перекрестье
GLUT_CURSOR_UP_DOWN	Стрелки вверх и вниз
GLUT_CURSOR_LEFT_RIGHT	Стрелки слева направо
GLUT_CURSOR_TOP_SIDE	Стрелка, указывающая на верхнюю сторону
GLUT_CURSOR_BOTTOM_SIDE	Стрелка, указывающая на нижнюю сторону
GLUT_CURSOR_LEFT_SIDE	Стрелка, указывающая на левую сторону
GLUT_CURSOR_RIGHT_SIDE	Стрелка, указывающая на правую сторону
GLUT_CURSOR_TOP_LEFT_CORNER	Стрелка, указывающая в верхний левый угол

Таблица 4.2 (окончание)

Константа	Тип курсора
GLUT_CURSOR_TOP_RIGHT_CORNER	Стрелка, указывающая в верхний правый угол
GLUT_CURSOR_BOTTOM_RIGHT_CORNER	Стрелка, указывающая в нижний правый угол
GLUT_CURSOR_BOTTOM_LEFT_CORNER	Стрелка, указывающая в нижний левый угол
GLUT_CURSOR_FULL_CROSSHAIR	Другое перекрестье
GLUT_CURSOR_NONE	Невидимый курсор
GLUT_CURSOR_INHERIT	Курсор родительского окна

Работа с меню

Библиотека GLUT поддерживает простейшие всплывающие (popup) меню.

Примечание

Во время использования меню его нельзя подвергать изменениям (добавлять, удалять или изменять его элементы).

Новое меню создает функция `glutCreateMenu`:

```
int glutCreateMenu ( void (*func)( int value ) );
```

Эта функция возвращает идентификатор созданного меню (положительное целое число). В качестве параметра `func` выступает функция, которая будет вызываться при выборе элемента меню (получая в качестве параметра номер выбранного элемента). Созданное функцией меню становится текущим.

Следующие две функции позволяют соответственно получить идентификатор текущего меню и установить заданное меню текущим:

```
int glutGetMenu ();
void glutSetMenu ( int menu );
```

Для удаления меню, заданного его идентификатором, предназначена функция `glutDestroyMenu`:

```
void glutDestroyMenu ( int menu );
```

При помощи функции `glutAddMenuEntry` к концу текущего меню можно добавить новый пункт:

```
void glutAddMenuEntry ( char * name, int value );
```

Параметр `name` задает строку, соответствующую добавляемому пункту меню, а параметр `value` — значение, передаваемое функции-обработчику событий меню при выборе данного пункта.

Библиотека GLUT также поддерживает вложенные (иерархические) меню, когда пункт меню может иметь подменю.

Для добавления подменю в качестве нового пункта к текущему меню предназначена функция `glutAddSubMenu`:

```
void glutAddSubMenu ( char * name, int menu );
```

Параметр `name` (как и в предыдущей функции) задает строку, отображаемую на экране для этого пункта меню, а параметр `menu` определяет меню, подключаемое к этому пункту.

Для изменения уже существующих пунктов текущего меню предназначены функции `glutChangeToMenuItem` и `glutChangeToSubMenu`:

```
void glutChangeToMenuItem ( int entry, char * name, int value );
```

```
void glutChangeToSubMenu ( int entry, char * name, int menu );
```

Изменяемый пункт задается номером `entry` (верхнему пункту меню соответствует единица).

Уничтожить заданный пункт (как обычный, так и имеющий подменю) текущего меню можно при помощи функции `glutRemoveItem`:

```
void glutRemoveItem ( int entry );
```

Поскольку все создаваемые GLUT меню являются всплывающими (popup), необходимо задать для текущего меню кнопку мыши, нажатие которой в текущем окне вызывает это меню. Для этого предназначены следующие функции:

```
void glutAttachMenu ( int button );
```

```
void glutDetachMenu ( int button );
```

Первая функция определяет, что при нажатии кнопки мыши, задаваемой параметром `button`, для текущего окна следует вызвать текущее меню. Вторая функция "отсоединяет" текущее меню от заданной кнопки мыши в текущем окне. Возможные значения параметра `button` приведены в табл. 4.3.

Таблица 4.3. Возможные значения параметра `button`

Константа	Кнопка мыши
<code>GLUT_LEFT_BUTTON</code>	Левая
<code>GLUT_MIDDLE_BUTTON</code>	Средняя
<code>GLUT_RIGHT_BUTTON</code>	Правая

Установка обработчиков сообщений

Для обработки сообщений GLUT использует так называемые callback-функции (или функции обратного вызова), т. е. приложение передает библиотеке GLUT адреса функций, которые должны вызываться при наступлении определенных событий. Выделяют три типа callback-функций — оконные, меню и глобальные (рис. 4.4).

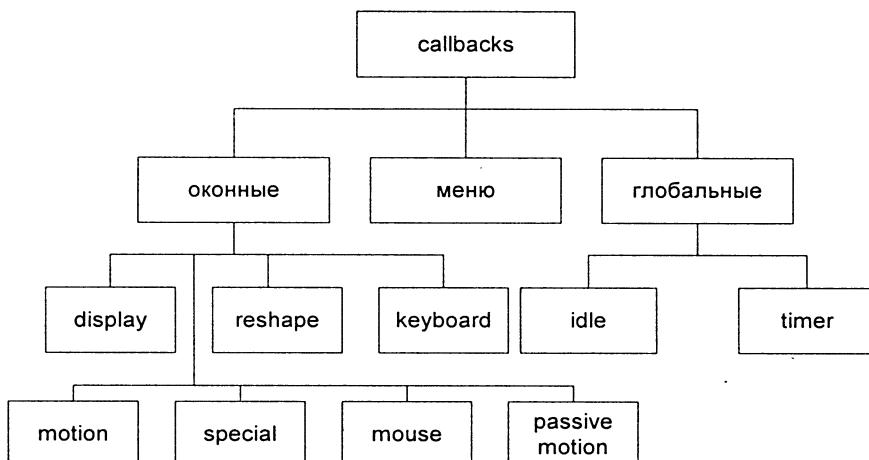


Рис. 4.4. Классификация обработчиков сообщений в GLUT

Оконные callback-функции позволяют приложению получать сообщения о необходимости перерисовки содержимого окна, об изменении его размеров, о получении ввода от пользователя. При этом обработчики сообщений ввода от пользователей получают сообщения от того окна, в котором ввод имел место.

Функции, служащие для обработки сообщений от *меню*, нами уже рассмотрены.

Глобальные callback-функции связаны с течением времени.

Одним из главных обработчиков сообщений является функция отображения содержимого окна. Для текущего окна ее можно задать при помощи функции `glutDisplayFunc`:

```
void glutDisplayFunc ( void (*func)() );
```

Параметр `func` является адресом функции, которая будет вызываться каждый раз, когда возникнет необходимость в перерисовке содержимого окна.

Установить обработчик изменения размеров для текущего окна можно с помощью функции `glutReshapeFunc`:

```
void glutReshapeFunc ( void (*func)( int width, int height ) );
```

Параметр `func` является адресом функции, которая будет вызываться каждый раз, когда требуется изменить размеры окна. Входными параметрами для нее служат новые размеры окна в пикселях (`width` и `height`).

Установить для текущего окна обработчик события, вызванного нажатием клавиши на клавиатуре, при котором генерируется ASCII-код, можно при помощи функции `glutKeyboardFunc`:

```
void glutKeyboardFunc ( void (*func)(unsigned char key, int x, int y) );
```

Передаваемая функция `func` будет вызываться каждый раз, когда происходит нажатие клавиши (клавиатурное событие), генерирующее ASCII-код. В качестве параметров в функцию-обработчик передаются собственно ASCII-код (`key`) и координаты курсора мыши (`x`, `y`) в момент возникновения данного клавиатурного события.

Для установки обработчика клавиатурных событий, не порождающих ASCII-кодов, предназначена функция `glutSpecialFunc`:

```
void glutSpecialFunc ( void (*func)( int key, int x, int y) );
```

Параметрами, передаваемыми функции-обработчику, являются код клавиши `key` (константы в табл. 4.4) и координаты курсора мыши в момент наступления данного события (`x` и `y`).

Таблица 4.4. Коды специальных клавиш

Константа	Клавиша
<code>GLUT_KEY_F1</code>	<code><F1></code>
<code>GLUT_KEY_F2</code>	<code><F2></code>
<code>GLUT_KEY_F3</code>	<code><F3></code>
<code>GLUT_KEY_F4</code>	<code><F4></code>
<code>GLUT_KEY_F5</code>	<code><F5></code>
<code>GLUT_KEY_F6</code>	<code><F6></code>
<code>GLUT_KEY_F7</code>	<code><F7></code>
<code>GLUT_KEY_F8</code>	<code><F8></code>
<code>GLUT_KEY_F9</code>	<code><F9></code>
<code>GLUT_KEY_F10</code>	<code><F10></code>
<code>GLUT_KEY_F11</code>	<code><F11></code>
<code>GLUT_KEY_F12</code>	<code><F12></code>
<code>GLUT_KEY_LEFT</code>	<code><↔></code>
<code>GLUT_KEY_UP</code>	<code><↑></code>

Таблица 4.4 (окончание)

Константа	Клавиша
GLUT_KEY_RIGHT	<→>
GLUT_KEY_DOWN	<↓>
GLUT_KEY_PAGE_UP	<Page Up>
GLUT_KEY_PAGE_DOWN	<Page Down>
GLUT_KEY_HOME	<Home>
GLUT_KEY_END	<End>
GLUT_KEY_INSERT	<Insert>

Для снятия клавиатурного обработчика (как обычных клавиатурных событий, так и нажатия специальных клавиш) достаточно вызвать соответствующую функцию установки обработчика с параметром `func`, равным `NULL`.

Для установки обработчика нажатий и отпусканьй кнопок мыши предназначена функция `glutMouseFunc`:

```
void glutMouseFunc ( void ( *func)( int button, int state,
                                         int x, int y ) );
```

Параметр `button`, передаваемый функции-обработчику, определяет кнопку мыши, вызвавшую данное событие (табл. 4.3). Параметр `state` принимает одно из двух значений в зависимости от того, была кнопка мыши нажата (`GLUT_DOWN`) или отпущена (`GLUT_UP`). Параметры `x` и `y` содержат координаты курсора мыши относительно окна в момент нажатия/отпускания клавиши мыши. Чтобы снять обработчик данного события для текущего окна, следует вызвать функцию `glutMouseFunc` с параметром `func`, равным `NULL`.

Также можно установить обработчик события передвижения курсора мыши для текущего окна. Для этого предназначены функции `glutMotionFunc` и `glutPassiveMotionFunc`:

```
void glutMotionFunc      ( void ( *func)( int x, int y ) );
void glutPassiveMotionFunc ( void ( *func)( int x, int y ) );
```

Первая из этих функций устанавливает обработчик, который будет вызываться при передвижении мыши, когда хотя бы одна из ее кнопок нажата. Вторая функция устанавливает обработчик для случая, когда происходит перемещение мыши и ни одна из кнопок мыши не нажата. В качестве входных параметров функция-обработчик получает координаты курсора мыши в системе координат окна.

Еще одним событием, связанным с мышью, для которого можно установить обработчик, является вхождение курсора мыши в область окна и выход

из нее. Для установки этого обработчика предназначена функция `glutEntryFunc`:

```
void glutEntryFunc ( void (*func)( int state ) );
```

Функция-обработчик получает в параметре `state` информацию о событии — курсор мыши покинул область окна (`GLUT_LEFT`) или вошел в область окна (`GLUT_ENTER`).

Также можно установить обработчик сообщений об изменении видимости окна.

```
void glutVisibilityFunc ( void (*func)( int state ) );
```

Функция-обработчик получает в параметре `state` текущее состояние видимости окна — отображается (`GLUT_VISIBLE`) или скрыто (`GLUT_NOT_VISIBLE`).

Примечание

Окно считается видимым, если виден хотя бы один из пикселов окна.

Еще одним обработчиком, который можно установить в GLUT, является так называемый `idle`-обработчик — обработчик, который вызывается каждый раз, когда нет других событий:

```
void glutIdleFunc ( void (*func)() );
```

Данный обработчик особенно удобен для создания анимации — в нем вычисляются новые параметры анимируемых объектов, после чего вызывается функция `glutPostRedisplay` для перерисовки содержимого окна.

Также можно установить обработчик, который будет вызван через определенный интервал времени. Для этого предназначена функция `glutTimerFunc`:

```
void glutTimerFunc ( unsigned int msecs, void (*func)( int value ),
                     int value );
```

Входными параметрами для функции `glutTimerFunc` являются интервал времени в миллисекундах (`msecs`), по истечении которого следует вызвать функцию-обработчик (`func`), и значение (`value`), передаваемое функции-обработчику. Наличие параметра `value` позволяет использовать одну и ту же функцию-обработчик для нескольких событий сразу.

Примечание

Функции `glutIdleFunc` и `glutTimerFunc` устанавливают глобальные обработчики событий, не привязанные ни к одному окну.

Получение информации о текущем состоянии GLUT

Одной из самых часто используемых функций для получения информации в GLUT является функция `glutGet`:

```
int glutGet ( GLenum state );
```

Параметр `state` определяет, какую именно информацию следует вернуть. Возможные значения параметра `state` приведены в табл. 4.5.

Таблица 4.5. Возможные значения параметра `state` функции `glutGet`

Константа	Возвращаемое значение
<code>GLUT_WINDOW_X</code>	Положение текущего окна в пикселях относительно верхнего левого угла экрана (координата <i>x</i>)
<code>GLUT_WINDOW_Y</code>	Положение текущего окна в пикселях относительно верхнего левого угла экрана (координата <i>y</i>)
<code>GLUT_WINDOW_WIDTH</code>	Ширина текущего окна в пикселях
<code>GLUT_WINDOW_HEIGHT</code>	Высота текущего окна в пикселях
<code>GLUT_WINDOW_BUFFER_SIZE</code>	Полное количество бит на пикセル
<code>GLUT_WINDOW_STENCIL_SIZE</code>	Количество бит на один пикセル в буфере трафарета
<code>GLUT_WINDOW_DEPTH_SIZE</code>	Количество бит на один пикセル в буфере глубины
<code>GLUT_WINDOW_RED_SIZE</code>	Количество бит на красный компонент одного пикселя
<code>GLUT_WINDOW_GREEN_SIZE</code>	Количество бит на зеленый компонент одного пикселя
<code>GLUT_WINDOW_BLUE_SIZE</code>	Количество бит на синий компонент одного пикселя
<code>GLUT_WINDOW_ALPHA_SIZE</code>	Количество бит на альфа-компонент одного пикселя
<code>GLUT_WINDOW_ACCUM_RED_SIZE</code>	Количество бит на красный компонент для одного пикселя в буфере накопления
<code>GLUT_WINDOW_ACCUM_GREEN_SIZE</code>	Количество бит на зеленый компонент для одного пикселя в буфере накопления
<code>GLUT_WINDOW_ACCUM_BLUE_SIZE</code>	Количество бит на синий компонент для одного пикселя в буфере накопления

Таблица 4.5 (окончание)

Константа	Возвращаемое значение
GLUT_WINDOW_ACCUM_ALPHA_SIZE	Количество бит на альфа-компонент для одного пикселя в буфере накопления
GLUT_WINDOW_DOUBLEBUFFER	1 — текущее окно использует двойную буфферизацию, 0 — в противном случае
GLUT_WINDOW_RGBA	1 — текущее окно в RGBA-режиме, 0 — текущее окно в палитровом режиме
GLUT_WINDOW_PARENT	Идентификатор родительского окна (0 для нормального окна)
GLUT_WINDOW_NUM_CHILDREN	Количество непосредственных дочерних окон текущего окна
GLUT_WINDOW_COLORMAP_SIZE	0 для RGBA-режимов, иначе — размер палитры
GLUT_WINDOW_NUM_SAMPLES	Количество образцов для одного пикселя для текущего окна
GLUT_WINDOW_STEREO	1 — текущее окно использует стереорежим, 0 — в противном случае
GLUT_WINDOW_CURSOR	Курсор для текущего окна
GLUT_SCREEN_WIDTH	Ширина экрана в пикселях
GLUT_SCREEN_HEIGHT	Высота экрана в пикселях
GLUT_SCREEN_WIDTH_MM	Ширина экрана в миллиметрах
GLUT_SCREEN_HEIGHT_MM	Высота экрана в миллиметрах
GLUT_MENU_NUM_ITEMS	Количество пунктов в текущем меню
GLUT_DISPLAY_MODE_POSSIBLE	Поддерживается ли текущий режим окна
GLUT_INIT_DISPLAY_MODE	Начальный режим для окна
GLUT_INIT_WINDOW_X	Начальная координата x окна
GLUT_INIT_WINDOW_Y	Начальная координата y окна
GLUT_INIT_WINDOW_WIDTH	Начальная ширина окна
GLUT_INIT_WINDOW_HEIGHT	Начальная высота окна
GLUT_ELAPSED_TIME	Количество миллисекунд с момента вызова функции <code>glutInit</code>

Для получения информации об устройствах предназначена функция `glutDeviceGet`:

```
int glutDeviceGet ( GLenum info );
```

Параметр `info`, возможные значения которого приведены в табл. 4.6, определяет информацию, которую следует вернуть.

Таблица 4.6. Возможные значения параметра `info`

Константа	Возвращаемое значение
<code>GLUT_HAS_KEYBOARD</code>	Ненулевое значение, если присутствует клавиатура
<code>GLUT_HAS_MOUSE</code>	Ненулевое значение, если присутствует мышь
<code>GLUT_HAS_TABLET</code>	Ненулевое значение, если присутствует планшет
<code>GLUT_NUM_MOUSE_BUTTONS</code>	Количество кнопок мыши (0 — мышь отсутствует)
<code>GLUT_NUM_SPACEBALL_BUTTONS</code>	Количество кнопок на устройстве Spaceball
<code>GLUT_NUM_TABLET_BUTTONS</code>	Количество кнопок на планшете

Также существует возможность получения информации о состоянии специальных клавиш в момент вызова обработчика сообщения. Для этого предназначена функция `glutGetModifiers`:

```
int glutGetModifiers ();
```

В возвращаемом целом числе могут быть установлены следующие биты:

- `GLUT_ACTIVE_SHIFT` — нажата клавиша <Shift> или <Caps Lock>;
- `GLUT_ACTIVE_CTRL` — нажата одна из клавиш <Ctrl>;
- `GLUT_ACTIVE_ALT` — нажата одна из клавиш <Alt>.

Данную функцию можно вызвать только из обработчиков событий от клавиатуры и мыши.

С помощью библиотеки GLUT можно получить информацию о поддержке заданных расширений (проверяются только общие расширения, GL):

```
int glutExtensionSupported ( char * extName );
```

Данная функция возвращает ненулевое значение в случае, если расширение с именем `extName` поддерживается. Для успешного выполнения этой функции необходимо, чтобы к моменту ее вызова существовало хотя бы одно окно, созданное при помощи GLUT.

Примечание

В главе 7 рассмотрен более общий способ проверки поддержки расширений при помощи библиотеки `libExt`.

Вывод текста

Библиотека GLUT поддерживает вывод текста с использованием двух типов шрифтов — векторных (каждый символ представлен в виде набора отрезков прямых) и растровых (каждый символ задается битовым изображением).

Для вывода одного символа растрового шрифта средствами OpenGL предназначена функция `glutBitmapCharacter`:

```
void glutBitmapCharacter ( void * font, int ch );
```

Параметр `font` задает используемый растровый шрифт. Растровые шрифты, поддерживаемые библиотекой GLUT, приведены в табл. 4.7.

Таблица 4.7. Значения параметра `font`

Константа	Описание шрифта
GLUT_BITMAP_8_BY_13	Шрифт с размером каждого символа 8 на 13 пикселов
GLUT_BITMAP_9_BY_15	Шрифт с размером каждого символа 9 на 15 пикселов
GLUT_BITMAP_TIMES_ROMAN_10	Пропорциональный шрифт Times Roman размером 10 пунктов
GLUT_BITMAP_TIMES_ROMAN_24	Пропорциональный шрифт Times Roman размером 24 пункта
GLUT_BITMAP_HELVETICA_10	Пропорциональный шрифт Helvetica размером 10 пунктов
GLUT_BITMAP_HELVETICA_12	Пропорциональный шрифт Helvetica размером 12 пунктов
GLUT_BITMAP_HELVETICA_18	Пропорциональный шрифт Helvetica размером 18 пунктов

Ширину символа, заданного параметром параметр `ch`, для растрового шрифта `font` можно узнать при помощи функции `glutBitmapWidth`:

```
int glutBitmapWidth ( GLUTbitmapFont font, int ch );
```

Для вывода символа векторного шрифта предназначена функция `glutStrokeCharacter`:

```
void glutStrokeCharacter ( void * font, int ch );
```

Параметр `font` задает векторный шрифт, который следует использовать для вывода символа `ch`, и может принимать одно из двух значений — `GLUT_STROKE_ROMAN` или `GLUT_STROKE_MONO_ROMAN`.

Аналогично, при помощи функции `glutStrokeWidth` можно получить ширину символа, заданного параметром `font` векторного шрифта:

```
int glutStrokeWidth ( GLUTstrokeFont font, int ch );
```

Вывод простейших объектов

Библиотека GLUT содержит ряд функций, предназначенных для вывода простейших геометрических объектов.

Примечание

Все эти функции задают правильные нормали в вершинах, но не задают текстурных координат. Единственным исключением является объект "Чайник" (`teapot`) — для него задаются и нормали и текстурные координаты.

Каждый из примитивов может быть выведен в одном из двух режимов — каркасном (`wire`) и сплошном (`solid`). Параметры функций вывода в каркасном и сплошном режиме одинаковы, а их имена отличаются только наличием слова `Wire` или `Solid`.

Примечание

Все примитивы выводятся в начале координат.

Сфера

Для вывода сферы предназначены функции:

```
void glutSolidSphere ( GLdouble r, GLint slices, GLint stacks );
void glutWireSphere ( GLdouble r, GLint slices, GLint stacks );
```

Параметр `r` задает радиус сферы, а параметры `slices` и `stacks` задают разбиение сферы на многоугольники.

Куб

Для вывода куба предназначены функции:

```
void glutSolidCube ( GLdouble size );
void glutWireCube ( GLdouble size );
```

Параметр `size` задает размер ребра куба.

Конус

Для вывода конуса предназначены функции `glutSolidCone` и `glutWireCone`. Основание конуса располагается в плоскости $z = 0$, а вершина — в точке $(0, 0, \text{height})$.

```
void glutSolidCone ( GLdouble base, GLdouble height,
                     GLint slices, GLint stacks );
void glutWireCone   ( GLdouble base, GLdouble height,
                     GLint slices, GLint stacks );
```

Параметр `base` задает радиус основания конуса, параметр `height` — его высоту (рис. 4.5). Параметры `slices` и `stacks` определяют разбиение конуса на полигоны (вдоль основания конуса и вдоль оси Oz).

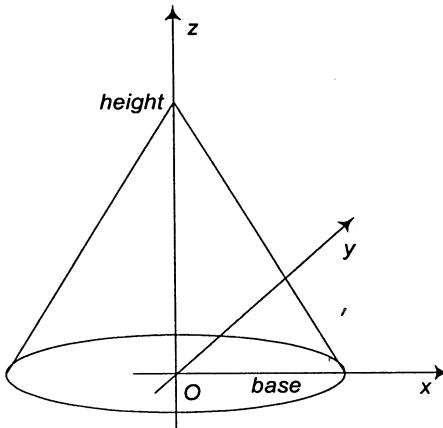


Рис. 4.5. Вывод конуса

Тор

Для вывода тора предназначены функции:

```
void glutSolidTorus ( GLdouble inner, GLdouble outer,
                     GLint sides, GLint rings );
void glutWireTorus   ( GLdouble inner, GLdouble outer,
                     GLint sides, GLint rings );
```

Параметры `inner` и `outer` задают внутренний и внешний радиусы тора (рис. 4.6), а параметры `sides` и `rings` — разбиение тора (для каждой радиальной секции и количество радиальных секций).

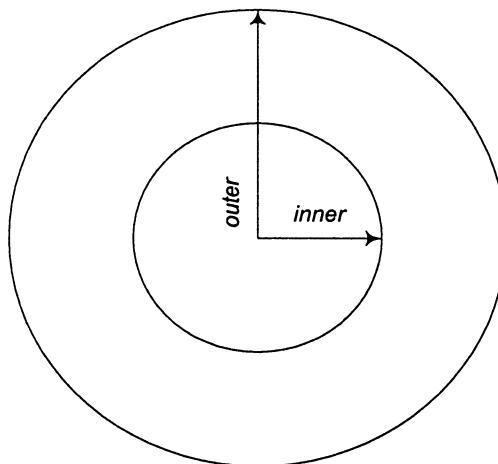


Рис. 4.6. Внутренний и внешний радиусы тора

Додекаэдр

Для вывода додекаэдра (правильного двенадцатигранника) предназначены функции:

```
void glutSolidDodecahedron ();
void glutWireDodecahedron ();
```

Эти функции осуществляют вывод додекаэдра с центром в начале координат и радиусом $\sqrt{3}$.

Октаэдр

Для вывода октаэдра (правильного восьмигранника) предназначены функции:

```
void glutSolidOctahedron ();
void glutWireOctahedron ();
```

Эти функции осуществляют вывод октаэдра с центром в начале координат и радиусом, равным единице.

Тетраэдр

Для вывода тетраэдра (правильного четырехгранника) предназначены функции:

```
void glutSolidTetrahedron ();
void glutWireTetrahedron ();
```

Эти функции осуществляют вывод тетраэдра с центром в начале координат и радиусом $\sqrt{3}$.

Икосаэдр

Для вывода икосаэдра (правильного двадцатигранника) предназначены функции:

```
void glutSolidIcosahedron ();
void glutWireIcosahedron ();
```

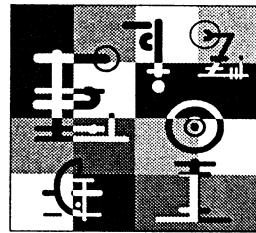
Эти функции осуществляют вывод икосаэдра с центром в начале координат и радиусом, равным единице.

"Чайник"

Библиотека GLUT поддерживает вывод ставшего классическим в компьютерной графике "Чайника," впервые использованного М. Ньюеллом (M. Newell). Для этого предназначены функции:

```
void glutSolidTeapot ( GLdouble size );
void glutWireTeapot ( GLdouble size );
```

Параметр `size` задает относительный размер "Чайника".



Глава 5

Общие сведения о библиотеках GLH, NV_MATH и NV_UTIL

Для облегчения работы с OpenGL и GLUT компания NVIDIA создала несколько библиотек на языке C++, которые входят в состав NVIDIA SDK.

В этой главе мы рассмотрим основные библиотеки — GLH, NV_MATH и NV_UTIL.

Библиотека GLH

Библиотека GLH (OpenGL Helper Library) представляет собой набор классов, облегчающих работу с библиотеками OpenGL и GLUT. Эти классы разделяются на три основные группы:

- инкапсуляция основных математических объектов;
- объектно-ориентированная надстройка над GLUT;
- объектно-ориентированная надстройка над OpenGL.

Все классы, вводимые в GLH, помещаются в пространство имен `glh` или в специальное внутреннее пространство имен `GLH_REAL_NAMESPACE`. Поэтому необходимо либо при каждом использовании класса ставить префикс (`glh::`), либо подключить данное пространство имен к текущему при помощи следующей команды:

```
using namespace glh;
```

Инкапсуляция основных математических классов

К основным математическим классам относятся векторы, матрицы, кватернионы, прямые и плоскости.

Векторы

Одной из самых важных математических абстракций является вектор. Основой всех реализаций вектора является следующий шаблонный класс:

```
template <int N, class T>
class vec
```

Данный класс представляет собой абстракцию n-мерного вектора, все компоненты которого имеют тип т. В табл. 5.1 приведены основные методы данного класса.

Таблица 5.1. Основные методы класса `vec<N, T>`

Метод	Описание
<code>int size() const</code>	Возвращает количество компонент (размерность) вектора
<code>vec(const T & t = T())</code>	Строит вектор по ссылке на массив чисел
<code>vec(const T * tp)</code>	Строит вектор по указателю на массив чисел
<code>const T * get_value() const</code>	Возвращает указатель на компоненты вектора как на массив
<code>T dot(const vec<N,T> & rhs) const</code>	Вычисляет скалярное произведение
<code>T length() const</code>	Вычисляет длину вектора
<code>T square_norm() const</code>	Вычисляет квадрат длины вектора
<code>void negate()</code>	Инвертирует вектор — изменяет знак каждого его компонента
<code>T normalize()</code>	Нормирует вектор, возвращает его исходную длину
<code>vec<N,T> & set_value(const T * rhs)</code>	Инициализирует вектор при помощи указателя на массив чисел
<code>T & operator [] (int i)</code>	Предоставляет доступ к компоненту массива (с возможностью его изменения)
<code>const T & operator [] (int i) const</code>	Предоставляет доступ к компоненту массива (только для чтения)

Для данного класса определены операторы *=, /= (покомпонентные операции умножения и деления вектора на другой вектор или скаляр), +=, -= (покомпонентные операции сложения и вычитания векторов), унарный минус, а также операторы +, - и * (поэлементные операции).

Действие этих операторов полностью соответствует традиционным покомпонентным операциям над n -мерными векторами.

Примечание

Операторы + и – обозначают покомпонентное сложение векторов. Операторы * и / обозначают покомпонентное умножение и деление вектора на вектор или скаляр. Операторы += и -= обозначают покомпонентное сложение и вычитание заданного вектора и другого вектора с записью результата в заданный вектор. Операторы *= и /= обозначают покомпонентное умножение и деление заданного вектора на другой вектор или скаляр с записью результата в заданный вектор.

Также для этого класса определены бинарные операции сложения (+), вычитания (-), умножения (*) (как на вектор, так и на скаляр) и деления (/) на скаляр и операторы сравнения == (равно) и != (не равно).

На основе данного шаблонного класса строится ряд классов, реализующих обычные двух-, трех- и четырехмерные вектора.

Библиотека GLH определяет тип `real` как `float` (в самом начале файла `glh_linear.h`) и использует его в основных классах.

Внутри пространства имен `GLH_REAL_NAMESPACE` (вложенного в пространство имен `glh`) вводятся классы `vec2`, `vec3` и `vec4` путем расширения классов `vec<2,real>`, `vec<3,real>` и `vec<4,real>`.

При этом к каждому такому классу добавляется конструктор, берущий на вход 2, 3 или 4 (в зависимости от класса) числа типа `real` и метод `get_value`, принимающий ссылки (`reference`) на соответствующее число переменных типа `real`.

В класс `vec3` добавляется метод `cross` для вычисления векторного произведения:

```
vec3 cross( const vec3 &rhs ) const
```

В класс `vec4` добавляется еще один конструктор, принимающий на вход трехмерный вектор и скалярное значение (в качестве четвертого компонента).

В самом пространстве имен `glh` следующим образом определяются классы `vec2f`, `vec3f` и `vec4f`:

```
typedef GLH_REAL_NAMESPACE::vec2 vec2f;
typedef GLH_REAL_NAMESPACE::vec3 vec3f;
typedef GLH_REAL_NAMESPACE::vec4 vec4f;
```

Матрицы

Еще одним крайне важным классом является абстракция матрицы 4×4 , вводимая в пространстве имен `GLH_REAL_NAMESPACE` как `matrix4` (и позже

определяется класс `matrix4f` как равный данному). В табл. 5.2 перечислены основные методы этого класса.

Таблица 5.2. Методы класса `matrix4`

Метод	Описание
<code>matrix4()</code>	Инициализирует матрицу единичной матрицей, т. е. присваивает ее диагональным элементам единичные значения, а всем остальным — нулевые значения
<code>matrix4(real r)</code>	Инициализирует все элементы матрицы значением r
<code>matrix4(real * m)</code>	Инициализирует матрицу массивом из 16 чисел
<code>matrix4 (real a00, real a01, real a02, real a03, real a10, real a11, real a12, real a13, real a20, real a21, real a22, real a23, real a30, real a31, real a32, real a33)</code>	Инициализирует матрицу путем явного задания значений всех ее элементов
<code>void get_value(real * mp) const</code>	Заполняет массив значениями элементов матрицы
<code>const real * get_value() const</code>	Возвращает указатель на массив элементов матрицы
<code>void set_value(real * mp)</code>	Инициализирует матрицу массивом чисел
<code>void set_value(real r)</code>	Инициализирует матрицу (все ее элементы) числом r
<code>void make_identity()</code>	Инициализирует матрицу значениями элементов единичной матрицы
<code>static matrix4 identity()</code>	Возвращает единичную матрицу
<code>void set_scale(real s)</code>	Заносит в первые три элемента по диагонали значение s
<code>void set_scale(const vec3 & s)</code>	Заносит в первые три элемента по диагонали значения соответствующих компонентов вектора s
<code>void set_translate(const vec3 & t)</code>	Заносит в последний столбец матрицы значения вектора t
<code>void set_row(int r, const vec4 & t)</code>	Заносит в строку матрицы r вектор t
<code>void set_column(int c, const vec4 & t)</code>	Заносит в столбец матрицы c вектор t

Таблица 5.2 (продолжение)

Метод	Описание
void get_row(int r, vec4 & t) const	Записывает заданную строку матрицы в передаваемый вектор
vec4 get_row(int r) const	Возвращает строку матрицы в виде вектора
void get_column(int c, vec4 & t) const	Записывает заданный столбец матрицы в передаваемый вектор
vec4 get_column(int c) const	Возвращает столбец матрицы в виде вектора
matrix4 inverse() const	Возвращает матрицу, обратную данной (т. е. матрицу, умножение которой на данную дает единичную матрицу)
matrix4 transpose() const	Возвращает транспонированную матрицу
matrix4 & mult_right(const matrix4 & b)	Умножает матрицу справа на b
matrix4 & mult_left(const matrix4 & b)	Умножает матрицу слева на b
void mult_matrix_vec(const vec3 &src, vec3 &dst) const	Умножает матрицу на вектор: dst = M * src
void mult_matrix_vec(vec3 & src_and_dst) const	Умножает матрицу на вектор: src_and_dst = M * src_and_dst
void mult_vec_matrix(const vec3 &src, vec3 &dst) const	Умножает вектор на матрицу: dst = src * M
void mult_vec_matrix(vec3 & src_and_dst) const	Умножает вектор на матрицу: src_and_dst = src_and_dst * M
void mult_matrix_vec(const vec4 &src, vec4 &dst) const	Умножает матрицу на вектор: dst = M * src
void mult_matrix_vec(vec4 & src_and_dst) const	Умножает матрицу на вектор: src_and_dst = M * src_and_dst
void mult_vec_matrix(const vec4 &src, vec4 &dst) const	Умножает вектор на матрицу: dst = src * M
void mult_vec_matrix(vec4 & src_and_dst) const	Умножает вектор на матрицу: src_and_dst = M * src_and_dst
void mult_matrix_dir(const vec3 &src, vec3 &dst) const	Умножает верхнюю левую подматрицу 3 × 3 на вектор
void mult_matrix_dir(vec3 & src_and_dst) const	Умножает верхнюю левую подматрицу 3 × 3 на вектор

Таблица 5.2 (окончание)

Метод	Описание
<code>void mult_dir_matrix(const vec3 &src, vec3 &dst) const</code>	Умножает вектор на верхнюю левую подматрицу 3×3
<code>void mult_dir_matrix(vec3 &src_and_dst) const</code>	Умножает вектор на верхнюю левую подматрицу 3×3
<code>real & operator () (int row, int col)</code>	Предоставляет доступ к элементу матрицы по столбцу и строке при помощи оператора ()
<code>const real & operator () (int row, int col) const</code>	Предоставляет доступ к элементу матрицы по столбцу и строке (только для чтения) при помощи оператора ()
<code>real & element (int row, int col)</code>	Предоставляет доступ к элементу матрицы по столбцу и строке
<code>const real & element (int row, int col) const</code>	Предоставляет доступ к элементу матрицы по столбцу и строке

Также для объектов класса `matrix4` определены операторы `*=` (в качестве параметра может выступать как матрица, так и скаляр), `+=, *, ==, !=`.

Кватернионы

Еще одним распространенным классом в компьютерной графике являются кватернионы [2, 3]. Кватернион представляет из себя четырехмерный вектор со специальным образом введенными операциями умножения и деления, является расширением комплексных чисел (при этом обычно компоненты x, y, z считаются мнимой частью кватерниона, а компонент w — действительной частью). В библиотеке GLH кватернионы представлены классами `quaternion` (или равным ему классом `quaternionf` в пространстве имен `glh`) и `rotation` (или равным ему классом `rotationf` в пространстве имен `glh`).

Примечание

Кватернионы задают повороты в трехмерном пространстве и в ряде случаев с ними работать гораздо удобнее, чем с матрицами.

В табл. 5.3 приведены основные методы класса `quaternion`.

Таблица 5.3. Основные методы класса *quaternion*

Метод	Описание
<code>quaternion()</code>	Создает кватернион (0, 0, 0, 1)
<code>quaternion(const real v[4])</code>	Инициализирует кватернион элементами массива компонентов
<code>quaternion(real q0, real q1, real q2, real q3)</code>	Инициализирует кватернион явным заданием всех компонентов
<code>quaternion(const matrix4 & m)</code>	Инициализирует кватернион матрицей поворота
<code>quaternion(const vec3 &axis, real radians)</code>	Инициализирует кватернион осью поворота и углом поворота
<code>quaternion(const vec3 &rotateFrom, const vec3 &rotateTo)</code>	Инициализирует кватернион поворотом, переводящим вектор <code>rotateFrom</code> в вектор <code>rotateTo</code>
<code>quaternion(const vec3 & from_look, const vec3 & from_up, const vec3 & to_look, const vec3& to_up)</code>	Инициализирует кватернион поворотом, переводящим одну ориентацию камеры в другую
<code>const real * get_value() const</code>	Возвращает указатель на массив компонентов кватерниона
<code>void get_value(real &q0, real &q1, real &q2, real &q3) const</code>	Возвращает все компоненты кватерниона
<code>quaternion & set_value(real q0, real q1, real q2, real q3)</code>	Устанавливает значения компонентов кватерниона
<code>void get_value(vec3 &axis, real &radians) const</code>	Возвращает ось и угол поворота, задаваемого данным кватернионом
<code>void get_value(matrix4 & m) const</code>	Возвращает матрицу поворота, соответствующую данному кватерниону
<code>quaternion & set_value(const real * qp)</code>	Устанавливает значения всех компонентов кватерниона равными элементам массива
<code>quaternion & set_value(const matrix4 & m)</code>	Устанавливает значения компонентов кватерниона при помощи матрицы поворота. При этом значения кватерниона устанавливаются таким образом, чтобы он соответствовал тому же повороту, что и заданная матрица
<code>quaternion & set_value(const vec3 &axis, real theta)</code>	Устанавливает значения компонентов кватерниона при помощи поворота вокруг оси на заданный угол

Таблица 5.3 (окончание)

Метод	Описание
<code>quaternion & set_value(const vec3 & rotateFrom, const vec3 & rotateTo)</code>	Устанавливает значения компонентов кватерниона таким образом, чтобы он соответствовал повороту, переводящему вектор <code>rotateFrom</code> в вектор <code>rotateTo</code>
<code>quaternion & set_value(const vec3 & from_look, const vec3 & from_up, const vec3 & to_look, const vec3 & to_up)</code>	Устанавливает значения компонентов кватерниона таким образом, чтобы он соответствовал повороту, переводящему одну ориентацию камеры в другую
<code>void normalize()</code>	Нормирует кватернион
<code>quaternion & conjugate()</code>	Выполняет сопряжение кватерниона, т. е. меняет знаки его x-, y- и z-компонентов
<code>quaternion & invert()</code>	Выполняет обращение кватерниона
<code>quaternion inverse() const</code>	Возвращает кватернион, обратный данному
<code>void mult_vec(const vec3 &src, vec3 &dst) const</code>	Выполняет поворот вектора <code>src</code> при помощи данного кватерниона, результат записывается в вектор <code>dst</code>
<code>void mult_vec(vec3 & src_and_dst) const</code>	Выполняет поворот вектора <code>src_and_dst</code> при помощи данного кватерниона
<code>void scale_angle(real scaleFactor)</code>	Умножает угол поворота, соответствующего данному кватерниону, на <code>scaleFactor</code>
<code>static quaternion slerp(const quaternion & p, const quaternion & q, real alpha)</code>	Выполняет сферическую линейную интерполяцию между кватернионами <code>p</code> и <code>q</code> с параметром <code>alpha</code> (принимающим значения от 0 до 1)
<code>static quaternion identity()</code>	Возвращает единичный кватернион (0, 0, 0, 1)
<code>real & operator [](int i)</code>	Предоставляет доступ к компонентам кватерниона по индексу
<code>const real & operator [](int i) const</code>	Предоставляет доступ к компонентам кватерниона по индексу (только для чтения)

Для класса `quaternion` определены операторы `*`, `*=`, `==` и `!=`.

Прямые

Еще одним из основных математических классов, вводимым библиотекой GLH, является класс `line` (класс `linef` в пространстве имен `glh`), инкапсулирующий прямые. Каждая прямая определяется точкой, через которую она проходит (`position`) и единичным вектором направления (`direction`). В табл. 5.4 приведены методы класса `line`.

Таблица 5.4. Методы класса `line`

Метод	Описание
<code>line()</code>	Инициализирует прямую с помощью оси Oz
<code>line(const vec3 &p0, const vec3 &p1)</code>	Инициализирует прямую с помощью двух точек, через которые она проходит
<code>void set_value(const vec3 &p0, const vec3 &p1)</code>	Позволяет установить данную прямую равной прямой, проходящей через точки $p0$ и $p1$
<code>bool get_closest_points(const line &line2, vec3 &pointOnThis, vec3 &pointOnThat)</code>	Позволяет найти ближайшие точки на двух прямых — в <code>pointOnThis</code> записывается точка, ближайшая к прямой <code>line2</code> , в <code>pointOnThat</code> записывается точка прямой <code>line2</code> , ближайшая к данной прямой.
<code>vec3 get_closest_point(const vec3 &point)</code>	Если прямые параллельны, то возвращается значение <code>false</code>
<code>const vec3 &get_position()</code>	Возвращает точку данной прямой, ближайшую к точке <code>point</code>
<code>const vec3 &get_direction()</code>	Возвращает точку, через которую проходит прямая
<code>const</code>	Возвращает единичный направляющий вектор прямой

Плоскости

Еще одним вводимым классом является абстракция плоскости в трехмерном пространстве — класс `plane` (`planef`).

Плоскость определяется единичным вектором нормали (`normal`) и расстоянием вдоль нормали до начала координат (`distance`), т. е. плоскость задается уравнением $(p, \text{normal}) = distance$.

В табл. 5.5 приведены основные методы класса `plane`.

Таблица 5.5. Основные методы класса plane

Метод	Описание
plane()	Инициализирует плоскость плоскостью Oxy
plane(const vec3 &p0, const vec3 &p1, const vec3 &p2)	Инициализирует плоскость плоскостью, проходящей через точки p0, p1 и p2
plane(const vec3 &normal, real distance)	Инициализирует плоскость нормалью и расстоянием до начала координат
plane(const vec3 &normal, const vec3 &point)	Инициализирует плоскость нормалью и точкой на плоскости
void offset(real d)	Сдвигает плоскость в направлении нормали на расстояние d
bool intersect(const line &l, vec3 &intersection) const	Находит пересечение плоскости с прямой l и возвращает точку пересечения в параметре intersection.
	Если пересечения нет, то возвращается значение false
void transform(const matrix4 &matrix)	Позволяет применить заданное матрицей аффинное преобразование к данной плоскости
bool is_in_half_space(const vec3 &point) const	Если точка point лежит в том полупространстве, которое указывает вектор нормали, то возвращается значение false, иначе — значение true
real distance(const vec3 & point) const	Возвращает расстояние от точки point до данной плоскости
const vec3 &get_normal() const	Возвращает единичный вектор нормали к плоскости
real get_distance_from_origin() const	Возвращает расстояние от данной плоскости до начала координат

Для данного класса определены операторы сравнения == и !=.

Объектная надстройка над GLUT

При всей своей простоте библиотека GLUT обладает рядом недостатков. Одним из них является ее процедурная ориентированность, что в частности приводит к необходимости для каждой программы заново писать требуемый набор функций (хотя для многих программ многие функции просто совпадают). В объектном подходе такая проблема решается простым наследованием.

Еще одним следствием процедурной ориентированности библиотеки является то, что все данные, которые требуются обработчикам сообщений,

должны быть глобальными (по крайней мере в пределах файла, в котором они используются).

Кроме того, можно установить не более одного обработчика сообщений для каждого типа сообщений для окна.

В библиотеке GLH предлагается довольно простое решение всех этих проблем, основанное на так называемых *интеракторах* (класс `glut_interactor`, листинг 5.1).

Листинг 5.1. Определение класса `glut_interactor`

```
class glut_interactor
{
public:
    glut_interactor() { enabled = true; }

    virtual void display() {}
    virtual void idle() {}
    virtual void keyboard(unsigned char key, int x, int y) {}
    virtual void menu_status(int status, int x, int y) {}
    virtual void motion(int x, int y) {}
    virtual void mouse(int button, int state, int x, int y) {}
    virtual void passive_motion(int x, int y) {}
    virtual void reshape(int w, int h) {}
    virtual void special(int key, int x, int y) {}
    virtual void timer(int value) {}
    virtual void visibility(int v) {}

    virtual void enable() { enabled = true; }
    virtual void disable() { enabled = false; }

    bool enabled;
};
```

Как видно из листинга 5.1, для каждого поддерживаемого GLUT сообщения в классе `glut_interactor` вводится свой виртуальный метод (ничего не делающий). Кроме этого каждый интерактор содержит флаг `enabled`, позволяющий простой установкой его в значение `false` выключить данный объект из цикла обработки сообщений.

Если объединить набор интеракторов в список и установить для каждого возможного сообщения обработчик, по очереди вызывающий все разрешенные

интеракторы, то мы получаем простую и удобную объектную обертку для GLUT. Необходимые определения и методы содержатся в файле glh_glut.h.

Функция `glut_helpers_initialize ()` предназначена для установки всех обработчиков сообщений в специальные функции, обеспечивающие работу с интеракторами, т. е. делегирующими им все поступающие запросы. Приведем только один из таких обработчиков (листинг 5.2).

Листинг 5.2. Обработчик сообщения отрисовки для работы с интеракторами

```
void glut_display_function()
{
    propagate = true;
    for(std::list<glut_interactor *>::iterator it=interactors.begin();
        it != interactors.end() && propagate; it++)
        (*it)->display();
}
```

Обратите внимание на использование глобальной переменной `propagate` — ее установка в значение `false` в одном из интеракторов прекращает дальнейший перебор интеракторов из списка.

Для добавления нового интерактора к списку предназначена функция `glut_add_interactor`:

```
void glut_add_interactor ( glut_interactor * gi, bool append=true );
```

Параметр `gi` является указателем на добавляемый интерактор, а параметр `append` определяет место списка, в которое следует поместить добавляемый интерактор, — в начало (`append=false`) или в конец (`append=true`).

Удалить интерактор из списка можно с помощью функции `glut_remove_interactor`:

```
void glut_remove_interactor ( glut_interactor * gi )
```

В общем случае можно каждый раз писать свой интерактор, определяя в нем все необходимые методы. Можно также создать некоторый базовый класс, содержащий постоянно используемые функции, и менять в нем всего один-два метода.

Есть, однако, и другой подход — можно разнести всю необходимую функциональность на большое количество отдельных интеракторов, каждый из которых выполняет лишь одно, крайне простое действие. Тогда в программе появится возможность получать необходимую функциональность просто путем комбинирования уже существующих интеракторов с нужными свойствами. Именно таким путем пошли создатели библиотеки GLH, предоставив набор готовых интеракторов, реализующих часто встречаемые действия.

На рис. 5.1 приведена диаграмма основных интеракторов, входящих в библиотеку GLH.

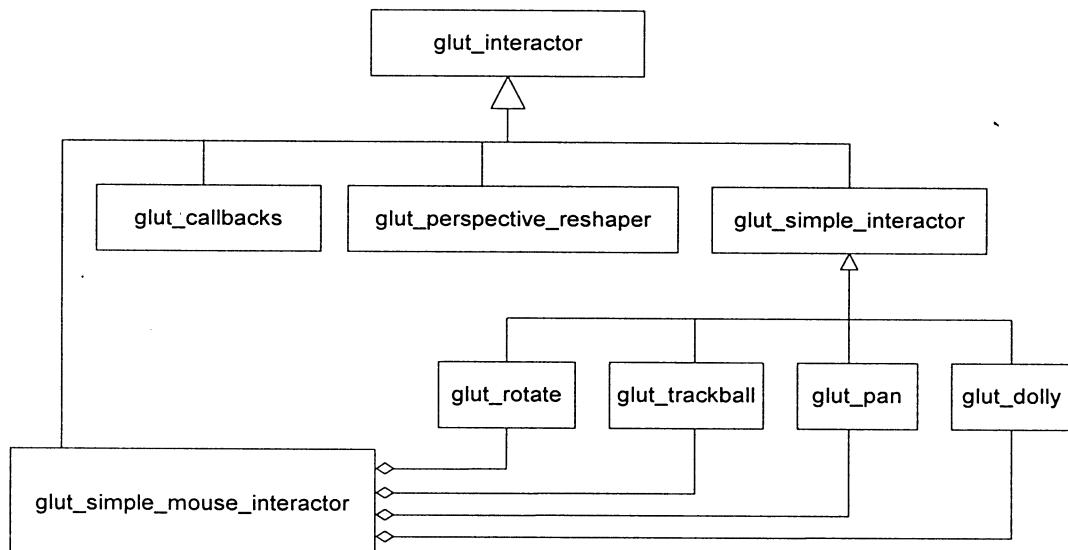


Рис. 5.1. Диаграмма основных интеракторов в GLH

Простейшим из них является интерактор `glut_callbacks` — он просто содержит для каждого обработчика указатель на соответствующую функцию (изначально равный `NULL`). Данный интерактор для каждого события вызывает по указателю соответствующую функцию (если значение указателя не равно `NULL`). Для записи открыты все эти указатели (`display_function`, `idle_function`, `keyboard_function`, `menu_status_function`, `motion_function`, `mouse_function`, `passive_motion_function`, `reshape_function`, `special_function`, `timer_function`, `visibility_function`).

Фактически данный интерактор позволяет использовать обычные функции в интеракторах.

Более интересным является интерактор `glut_perspective_reshaper`. Он реализует часть метода `reshape` — настройку перспективной проекции в соответствии с переданными в конструктор параметрами:

```
glut_perspective_reshaper ( float infovy = 60.f, float inzNear = .1f,
                           float inzFar = 10.f);
```

Класс `glut_simple_interactor` позволяет отслеживать только определенные события (нажатие кнопки мыши, клавиатурного модификатора). Данный интерактор отслеживает перемещения мыши с момента наступления последнего подобного события и предоставляет возможность отреагировать на событие через виртуальный метод `update`. Поскольку сам этот метод в данном

классе неопределен, класс `glut_simple_interactor` является *абстрактным* (т. е. нельзя создавать его экземпляры). Зато этот класс предоставляет базу для целого ряда обработчиков сообщений мыши, обеспечивающих возможность вращения объекта мышью, передвижение объекта как в стороны, так и вперед/назад. Вся эта функциональность реализуется интеракторами `glut_pan`, `glut_dolly`, `glut_trackball` и `glut_rotate`.

Класс `glut_simple_mouse_interactor` обеспечивает всю описанную функциональность путем агрегирования — он содержит в себе интеракторы `glut_pan`, `glut_dolly`, `glut_trackball` и `glut_rotate` и делегирует им необходимые запросы.

Объектная надстройка над OpenGL

В библиотеке GLH содержится несколько объектных "оберток" для ряда объектов OpenGL — дисплейных списков, текстурных объектов, вершинных и фрагментных программ. Мы рассмотрим лишь первые две из них.

Для инкапсуляции дисплейного списка в класс в библиотеке предназначен класс `display_list`, исходный текст которого приводится на листинге 5.3.

Листинг 5.3. Класс `display_list`

```
class display_list
{
public:
    // set managed to true if you want the class to cleanup
    // objects in the destructor
    display_list(bool managed = false)
        : valid(false), manageObjects(managed) {}

    virtual ~display_list()
    {
        if (manageObjects)
            del();
    }

    void call_list()
    {
        if(valid)
            glCallList(dlist);
    }
}
```

```
void new_list(GLenum mode)
{
    if(!valid)
        gen();

    glGenLists(dlist, mode);
}

void end_list()
{
    glEndList();
}

void del()
{
    if(valid)
        glDeleteLists(dlist, 1);

    valid = false;
}

bool is_valid() const { return valid; }

private:

void gen()
{
    dlist = glGenLists(1);
    valid=true;
}

bool valid;
bool manageObjects;
GLuint dlist;
};
```

Для создания самого списка следует, создав объект `display_list`, вызвать метод `new_list`, выполнить все команды OpenGL, которые должны быть запомнены в списке, после чего следует вызвать метод `end_list`.

Метод `call_list` осуществляет вызов списка, а метод `is_valid` определяет, существует ли соответствующий дисплейный список OpenGL.

Параметр конструктора определяет, должен ли при уничтожении данного объекта C++ быть уничтожен и сам дисплейный список OpenGL (т. е. нужно ли освобождать связанные с ним ресурсы).

Объектная "обертка" текстурного объекта реализована в виде класса `tex_object` и унаследованных от него классов `tex_object_1D`, `tex_object_2D`, `tex_object_3D` и `tex_object_cube_map`. В листинге 5.4 приведены описания этих классов.

Листинг 5.4. Описание текстурных классов библиотеки GLH

```
class tex_object
{
public:
    // set managed to true if you want the class to cleanup
    // objects in the destructor
    tex_object(GLenum tgt, bool managed)
        : target(tgt), valid(false), manageObjects(managed) {}

    virtual ~tex_object()
    {
        if (manageObjects)
            del();
    }

    void bind()
    {
        if(!valid)
            gen();

        glBindTexture(target, texture);
    }

    // convenience methods
    void parameter(GLenum pname, GLint i)
    {
        glTexParameteri(target, pname, i);
    }
}
```

```
void parameter(GLenum pname, GLfloat f)
{
    glTexParameterf(target, pname, f);
}

void parameter(GLenum pname, GLint * ip)
{
    glTexParameteriv(target, pname, ip);
}

void parameter(GLenum pname, GLfloat * fp)
{
    glTexParameterfv(target, pname, fp);
}

void enable()
{
    glEnable(target);
}

void disable()
{
    glDisable(target);
}

void del()
{
    if(valid)
        glDeleteTextures(1, &texture);

    valid = false;
}

bool is_valid() const
{
    return valid;
}

void gen()
```

```
{  
    glGenTextures(1, &texture);  
    valid=true;  
}  
  
Glenum    target;  
Bool      valid;  
bool      manageObjects;  
GLuint    texture;  
};  
  
class tex_object_1D : public tex_object  
{  
public:  
    tex_object_1D(bool managed = false) : tex_object(GL_TEXTURE_1D,  
                                                    managed) {}  
};  
  
class tex_object_2D : public tex_object  
{  
public:  
    tex_object_2D(bool managed = false) : tex_object(GL_TEXTURE_2D,  
                                                    managed) {}  
};  
  
class tex_object_3D : public tex_object  
{  
public:  
    tex_object_3D(bool managed = false) : tex_object(GL_TEXTURE_3D,  
                                                    managed) {}  
};  
  
class tex_object_cube_map : public tex_object  
{  
public:  
    tex_object_cube_map(bool managed = false) :  
        tex_object(GL_TEXTURE_CUBE_MAP_ARB, managed) {}  
};
```

Функциональность всех этих классов крайне проста, и основным удобством, которое они предоставляют, является более удобный способ управления параметрами и автоматическое освобождение соответствующих ресурсов.

Библиотека NV_MATH

Библиотека NV_MATH представляет собой еще одну реализацию основных математических объектов и нескольких полезных функций. В отличие от библиотеки GLH, данная библиотека не помещает свои классы в специальные пространства имен. Для использования библиотеки NV_MATH необходимо подключить файлы NV_MATH.H, NV_ALGEBRA.H и NV_MATH.LIB.

В качестве базового типа для всех вводимых классов используется тип `nv_scalar`, определенный как `float`.

Для работы с векторами библиотека NV_MATH вводит классы `vec2`, `vec3`, `vec4` и их транспонированные аналоги `vec2t`, `vec3t` и `vec4t`. Для каждого из этих классов есть большой набор конструкторов, позволяющих инициализировать вектор набором чисел, указателем на массив чисел, другими векторами (в том числе с другой размерностью). Для каждого из этих классов определены операции сравнения (`==` и `!=`). Также определены покомпонентные операции `+`, `-`, `*`, `/`, `+=`, `-=` и `*=`.

Для доступа к отдельным компонентам можно использовать как перегруженный оператор `[]`, так и имена компонентов (x, y, z, w) и (s, t, r, q) .

Метод `sq_norm` возвращает значение квадрата длины вектора, а метод `norm` возвращает саму длину.

Для класса `vec3` также определены следующие методы:

```
nv_scalar normalize      ();
void      orthogonalize ( const vec3& v );
void      orthonormalize( const vec3& v );
```

Первый из них нормирует вектор, возвращая его первоначальную длину; метод `orthogonalize` делает данный вектор ортогональным другому; метод `orthonormalize` сперва делает вектор ортогональным другому, а потом нормирует.

Кроме собственно методов классов библиотека предоставляет много различных функций для работы с ними. В табл. 5.6 приведена краткая сводка таких функций, в случае когда существуют версии функций для всех классов (`vec2`, `vec3` и `vec4`), вместо типа указывается `T`.

Таблица 5.6. Основные функции библиотеки NV_MATH для работы с векторами

Функция	Описание
<code>T& normalize(T & u);</code>	Нормирует вектор
<code>nv_scalar nv_sq_norm(const vec3 & n)</code>	Вычисляет квадрат длины вектора
<code>nv_scalar nv_norm(const vec4 & n)</code>	Вычисляет длину вектора
<code>nv_scalar nv_norm(const vec3 & n)</code>	Вычисляет векторное произведение
<code>vec3 & cross(vec3 & u, const vec3 & v, const vec3 & w);</code>	
<code>nv_scalar dot(const vec3 & v, const vec3 & w);</code>	Вычисляет скалярное произведение
<code>nv_scalar dot(const vec4 & v, const vec4 & w);</code>	Вычисляет скалярное произведение
<code>vec3 & reflect(vec3 & r, const vec3 & n, const vec3 & l);</code>	Вычисляет отражение вектора <code>r</code> относительно нормали <code>n</code> и записывает результат в <code>r</code>
<code>madd(vec3 & u, const vec3 & v, const nv_scalar & lambda);</code>	Вычисляет значение <code>u = v*lambda + u</code>
<code>T& scale(T& u, const nv_scalar s);</code>	Умножает вектор на скаляр и возвращает измененный вектор
<code>T& mult(T& u, const M& m, const T& v);</code>	Вычисляет значение <code>u=M*v</code> (<code>T</code> — vec3 или vec4, <code>M</code> — mat3 или mat4)
<code>T& mult(T& u, const T& v, const M& m);</code>	Вычисляет значение <code>u=v*M</code>
<code>inline void nv_max(vec3 & vOut, const vec3 & vFirst, const vec3 & vSecond)</code>	Находит покомпонентный максимум для векторов <code>vFirst</code> и <code>vSecond</code> и записывает его значение в <code>vOut</code>
<code>inline void nv_min(vec3 & vOut, const vec3 & vFirst, const vec3 & vSecond)</code>	Находит покомпонентный минимум для векторов <code>vFirst</code> и <code>vSecond</code> и записывает его значение в <code>vOut</code>
<code>nv_scalar nv_area(const vec3 & v1, const vec3 & v2, const vec3 & v3);</code>	Вычисляет площадь треугольника <code>v1v2v3</code>
<code>nv_scalar nv_perimeter(const vec3 & v1, const vec3 & v2, const vec3 & v3);</code>	Вычисляет периметр треугольника <code>v1v2v3</code>
<code>nv_scalar nv_find_in_circle(vec3 & center, const vec3 & v1, const vec3 & v2, const vec3 & v3);</code>	Находит центр окружности, вписанной в треугольник <code>v1v2v3</code> , возвращает радиус этой окружности

Таблица 5.6 (окончание)

Функция	Описание
<code>nv_scalar nv_find_circ_circle(vec3 & center, const vec3 & v1, const vec3 & v2, const vec3 &v3);</code>	Находит центр окружности, описанной вокруг треугольника $v1v2v3$, возвращает радиус этой окружности
<code>nv_scalar fast_cos(const nv_scalar x);</code>	Выполняет быстрое приближенное вычисление косинуса
<code>nv_scalar ffast_cos(const nv_scalar x);</code>	

Для работы с матрицами 3×3 и 4×4 в библиотеке NV_MATH вводятся классы `mat3` и `mat4`. Каждый из этих классов имеет пустой конструктор, конструктор, инициализирующий матрицу массивом чисел, и соруконструктор (конструктор копирования). Основные методы этих классов приведены в табл. 5.7.

Таблица 5.7. Основные методы классов `mat3` и `mat4`

Метод	Описание
<code>const vec4 col(const int i) const</code>	Возвращает столбец матрицы как вектор
<code>const vec4 operator[](const int& i) const</code>	Возвращает строку матрицы в виде вектора
<code>const nv_scalar& operator()(const int& i, const int& j) const</code>	Возвращает константную (т. е. доступную только для чтения) ссылку на элемент матрицы (i, j)
<code>nv_scalar& operator()(const int& i, const int& j)</code>	Возвращает ссылку на элемент матрицы (i, j), для которой разрешены и чтение и запись
<code>void set_col(int i, const vec4 & v)</code>	Записывает вектор как столбец матрицы
<code>void set_row(int i, const vec4 & v)</code>	Записывает вектор как строку матрицы
<code>mat3 & get_rot(mat3 & M) const</code>	Возвращает матрицу M (метод определен только для класса <code>mat4</code>) — извлекает верхнюю левую подматрицу 3×3 для данной матрицы и записывает ее в матрицу M
<code>quat & get_rot(quat & q) const</code>	Возвращает параметр q (метод определен только для класса <code>mat4</code>) — находит кватернион, соответствующий повороту, задаваемому данной матрицей, и записывает его в параметр q

Таблица 5.7 (окончание)

Метод	Описание
void set_rot(const quat & q)	Устанавливает данную матрицу равной матрице поворота, определяемого квaternionом q
void set_rot(const mat3 & M)	Устанавливает данную матрицу равной матрице поворота M
void set_rot(const nv_scalar & theta, const vec3 & v)	Устанавливает данную матрицу равной матрице поворота вокруг вектора v на угол θ (в радианах)
void set_rot(const vec3 & u, const vec3 & v)	Устанавливает данную матрицу равной матрице поворота, переводящей вектор u в вектор v
void set_scale(const vec3& s)	Устанавливает матрицу равной матрице масштабирования, коэффициенты масштабирования для каждой из осей определяются соответствующими компонентами вектора s
vec3& get_scale(vec3& s) const	Возвращает вектор из первых 3 диагональных элементов данной матрицы, записанных в параметр s (метод определен только для класса <code>mat4</code>)
void set_translation(const vec3 & t)	Устанавливает все элементы данной матрицы равными матрице переноса на вектор t
vec3 & get_translation(vec3 & t) const	Возвращает вектор из первых 3 элементов крайнего правого столбца данной матрицы, записанных в параметр t (метод определен только для класса <code>mat4</code>)

Для матриц определены операторы умножения на матрицу и вектор (справа и слева). Кроме этого для матриц определен целый ряд функций, приведенных в табл. 5.8.

Таблица 5.8. Функции для работы с матрицами

Функция	Описание
mat4 & add(mat4 & A, const mat4 & B)	Вычисляет значение $A += B$
mat3 & add(mat3 & A, const mat3 & B)	
mat4 & add(mat4 & C, const mat4 & A, const mat4 & B)	Вычисляет значение $C = A + B$
mat3 & add(mat3 & C, const mat3 & A, const mat3 & B)	

Таблица 5.8 (окончание)

Функция	Описание
mat4 & mult(mat4 & C, const mat4 & A, const mat4 & B)	Вычисляет значение $C = A * B$
mat3 & mult(mat3 & C, const mat3 & A, const mat3 & B)	
extern mat4 & negate(mat4 & M); extern mat3 & negate(mat3 & M);	Вычисляет значение $M = -M$
mat3 & transpose(mat3 & B, const mat3 & A) mat4 & transpose(mat4 & B, const mat4 & A)	Транспонирует матрицу
mat3 & transpose(mat3 & B) mat4 & transpose(mat4 & B)	Транспонирует матрицу
mat4 & invert(mat4 & B, const mat4 & A) mat3 & invert(mat3 & B, const mat3 & A)	Записывает в B матрицу, обратную к A
mat4 & look_at(mat4 & M, const vec3 & eye, const vec3 & center, const vec3 & up)	Возвращает матрицу, соответствующую команде gluLookAt
mat4 & frustum(mat4 & M, const nv_scalar l, const nv_scalar r, const nv_scalar b, const nv_scalar t, const nv_scalar n, const nv_scalar f)	Возвращает матрицу, соответствующую команде glFrustum
nv_scalar det(const mat3 & A)	Вычисляет определитель матрицы

Важной особенностью классов `mat3` и `mat4` является то, что они хранят элементы в том порядке, который принят в OpenGL, что позволяет легко использовать эти матрицы при работе с OpenGL.

Также библиотека NV_MATH поддерживает работу с кватернионами, введя для этого класс `quat`. Основные методы этого класса приведены в табл. 5.9.

Таблица 5.9. Основные методы класса `quat`

Метод	Описание
quat()	Пустой конструктор
quat(nv_scalar x, nv_scalar y, nv_scalar z, nv_scalar w)	Инициализирует кватернион значениями всех компонент
quat(const quat& quat)	Copy-конструктор
quat(const vec3& axis, nv_scalar angle)	Инициализирует кватернион поворотом на заданный угол вокруг оси
quat(const mat3& rot)	Инициализирует кватернион матрицей поворота

Таблица 5.9 (окончание)

Метод	Описание
quat Inverse()	Обращает кватернион
void Normalize()	Нормирует кватернион
void FromMatrix(const mat3& mat)	Строит кватернион по матрице поворота
void ToMatrix(mat3& mat) const	Инициализирует матрицу поворотом, задаваемым кватернионом
nv_scalar& operator[](int i) const nv_scalar operator[](int i)	Предоставляет покомпонентный доступ к компонентам кватерниона

Для членов этого класса определены операторы `=`, `-`, `*` и `*=`.

В табл. 5.10 приведены основные функции над кватернионами.

Таблица 5.10. Основные функции для работы с кватернионами

Функция	Описание
quat& normalize(quat & p)	Нормирует кватернион
quat& conj(quat & p)	Применяет к кватерниону операцию со-пряжения
quat& conj(quat & p, const quat & q)	Применяет к кватерниону операцию со-пряжения
quat& add_quats(quat& p, const quat& q1, const quat& q2)	Складывает кватернионы
quat& axis_to_quat(quat & q, const vec3 & a, const nv_scalar phi)	Строит кватернион по повороту вокруг оси на заданный угол
mat3& quat_2_mat(mat3 &M, const quat &q)	Строит матрицу поворота по кватерниону
quat& mat_2_quat(quat &q, const mat3 &M)	Строит кватернион по матрице поворота

Кроме того в библиотеку NV_MATH входит функция `nv_random`, возвращающая случайное вещественное число из промежутка $[-1, 1]$.

Библиотека NV_UTIL

Библиотека NV_UTIL предназначена для работы с данными нескольких распространенных форматов — TGA, JPG (текстуры), ASE (модели), ZIP (архивы).

Для использования этой библиотеки следует подключить файлы NV_UTIL.H и NV_UTIL.LIB.

Чтение текстур в формате TGA

Для работы с TGA-файлами библиотека NV_UTIL в пространстве имен `tga` вводит функцию `read`:

```
tgaImage * read(const char * filename);
```

Эта функция читает текстурные данные из файла с именем `filename` и возвращает либо `NULL` (если изображение не удалось прочитать), либо указатель структуры типа `tgaImage`, описанной в листинге 5.5.

Листинг 5.5. Описание структуры `tgaImage`

```
typedef struct
{
    GLsizei width;
    GLsizei height;
    GLint components;
    GLenum format;
    GLsizei cmapEntries;
    GLenum cmapFormat;
    GLubyte *cmap;

    GLubyte *pixels;
} tgaImage;
```

Фактически поля этой структуры несут всю информацию, необходимую для загрузки изображения в качестве текстуры. В листинге 5.6 приведен фрагмент кода, осуществляющий загрузку текстуры типа TGA при помощи библиотеки NV_UTIL.

Листинг 5.6. Загрузка текстуры типа TGA

```
unsigned loadTga ( const char * fileName )
{
    tgaImage * im = read ( fileName );

    if ( im == NULL )
```

```

    return 0;

    unsigned textureId;

    glGenTextures ( 1, &textureId );
    glBindTexture ( GL_TEXTURE_2D, textureId );
    glPixelStorei ( GL_UNPACK_ALIGNMENT, 1 );
    glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
    glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );

    gluBuild2DMipmaps ( GL_TEXTURE_2D, im->components, im->width,
                        im->height, im->format, GL_UNSIGNED_BYTE,
                        im->pixels );

    glTexParameteri ( GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
                      GL_LINEAR );
    glTexParameteri ( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                      GL_LINEAR_MIPMAP_LINEAR );

    return textureId;
}

```

Чтение текстур в формате JPG

Для загрузки изображений в формате JPG в библиотеке NV_UTIL в пространстве имен jpeg вводится следующая функция:

```
int read ( const char * filename, int * width, int * height,
           unsigned char ** pixels, int * components );
```

Данная функция пытается загрузить текстуру из файла с именем `filename`, при этом параметры `width`, `height` и `components` указывают на переменные, в которые будут записаны ширина и высота изображения в пикселях и количество компонентов цвета на пикセル. Параметр `pixels` содержит адрес переменной, в которую будет записан указатель выделенной области памяти с пикселями изображения (выделение осуществляется при помощи оператора `new`). В случае успешной загрузки возвращается ноль.

В листинге 5.7 приведен пример загрузки изображения с использованием этой функции.

Листинг 5.7. Пример загрузки изображения типа JPG

```

unsigned loadJpg ( const char * fileName )
{
    int             width, height, components;
    unsigned char * pixels;

    if ( read ( fileName, &width, &height, &pixels, &components ) )
        return 0;

    unsigned textureId;

    glGenTextures      ( 1, &textureId );
    glBindTexture      ( GL_TEXTURE_2D, textureId );
    glPixelStorei     ( GL_UNPACK_ALIGNMENT, 1 );
    glTexParameteri   ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
    glTexParameteri   ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );

    gluBuild2DMipmaps ( GL_TEXTURE_2D, components, width,
                        height, GL_RGB, GL_UNSIGNED_BYTE, pixels );

    glTexParameteri   ( GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
                        GL_LINEAR );
    glTexParameteri   ( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR_MIPMAP_LINEAR );

    delete pixels;

    return textureId;
}

```

Чтение данных из ZIP-архивов

Для чтения данных из ZIP-архива предназначена функция `open`, объявленная в пространстве имен `unzip` библиотеки `NV_UTIL`:

```

unsigned char * open ( const char * filename, const char * inzipfile,
                      unsigned int * size);

```

Параметры: `filename` — имя ZIP-архива, `inzipfile` — имя файла внутри архива, который требуется извлечь, `size` — адрес переменной, в которую будет помещена длина извлеченного из архива файла.

Функция возвращает указатель на область памяти (выделенный при помощи оператора new), содержащую образ требуемого файла. В случае ошибки возвращается значение NULL.

Чтение моделей в формате ASE

Одним из простых и достаточно распространенных форматов хранения трехмерных моделей является текстовый формат ASE, применяемый в пакете 3D Studio MAX. Достоинствами этого формата являются его простая организация и поддержка рядом пакетов.

Для поддержки формата ASE в библиотеке NV_UTIL в пространстве имен ase вводятся ряд классов и функция для загрузки моделей из файлов этого формата:

```
model * load ( const char * filename, float scale );
model * load ( const char * buf, unsigned int size, float scale );
geomobj * get_geomobj ( model * m, const char * name );
bool load_tex ( model * m );
```

Прочитанные из файла данные возвращаются в виде указателей на специальные классы, наиболее важными из которых являются `model` и `geomobj`.

Первая из функций `load` загружает модель из файла с именем `filename`, применяя к загруженной модели масштабирование с коэффициентом `scale`. Вторая функция `load` загружает данные не из файла модели, а из буфера `buf` в памяти, содержащего `size` байт данных.

Функция `get_geomobj` возвращает указатель на объект с заданным именем, содержащийся внутри загруженной модели.

Функция `load_tex` предназначена для загрузки всех используемых моделью текстур.

Класс `model` служит для представления всех данных, прочитанных из ASE-файла, и описывается следующей структурой:

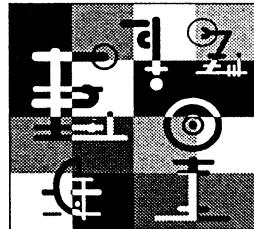
```
struct model
{
    model() : name(0), time(0) {}
    ~model();

    char *      name;
    geom_array  root_geom;
    geom_array  geom;
    mat_array   mat;
```

```
tex_map      tex;
int          time;
};
```

Поля `root_geom` и `geom` представляют собой списки отдельных объектов, являющихся экземплярами класса `geomobj`. К сожалению, сам класс `geomobj` достаточно сложен, поскольку в нем хранится вся информация, которая только может быть извлечена из ASE-файла, включая полную информацию об используемых материалах, анимации и т. п.

Более подробное описание этого класса здесь не проводится, поскольку в главе 11 будут рассмотрены загрузчики для целого ряда форматов моделей, включая ASE (однако при этом для некоторых из таких файлов извлекается не вся содержащаяся в них информация, а только ее подмножество, достаточное для рендеринга модели). Полное описание класса `geomobj` и всех используемых им структур вы можете найти в файле `NV_ASE.H`, входящем в состав библиотеки `NV_UTIL`.



Глава 6

Основные классы для работы с векторами, матрицами и кватернионами

В дальнейшем в книге для работы с векторами, матрицами и кватернионами будут использоваться написанные автором классы `Vector2D`, `Vector3D`, `Vector4D`, `Matrix3D`, `Matrix4x4` и `Quaternion`. По своей функциональности они примерно аналогичны соответствующим классам из NVIDIA SDK.

Работа с векторами

Векторы размерности 2, 3 и 4 представлены классами `Vector2D`, `Vector3D` и `Vector4D`. Все эти классы очень близки друг к другу, поэтому в книге будет разобран лишь один из — `Vector3D` как обладающий наибольшей функциональностью (листинг 6.1).

Листинг 6.1. Описание класса `Vector3D` и функций для работы с ним

```
#include <math.h>

#ifndef EPS
#define EPS 0.0001f
#endif

#ifndef M_PI // since not all compilers define it
#define M_PI 3.14159265358979323846f
#endif

class Vector3D
{
```

```
public:
    float    x, y, z;

    Vector3D () {}
    Vector3D ( float v ) : x ( v ), y ( v ), z ( v ) {}
    Vector3D ( float px, float py, float pz ) : x ( px ), y ( py ),
                                                z ( pz ) {}
    Vector3D ( const Vector3D& v ) : x ( v.x ), y ( v.y ),
                                       z ( v.z ) {}

    Vector3D& operator = ( const Vector3D& v )
    {
        x = v.x;
        y = v.y;
        z = v.z;

        return *this;
    }

    Vector3D operator + () const
    {
        return *this;
    }

    Vector3D operator - () const
    {
        return Vector3D ( -x, -y, -z );
    }

    Vector3D& operator += ( const Vector3D& v )
    {
        x += v.x;
        y += v.y;
        z += v.z;

        return *this;
    }

    Vector3D& operator -= ( const Vector3D& v )
```

```
{  
    x -= v.x;  
    y -= v.y;  
    z -= v.z;  
  
    return *this;  
}  
  
Vector3D& operator *= ( const Vector3D& v )  
{  
    x *= v.x;  
    y *= v.y;  
    z *= v.z;  
  
    return *this;  
}  
  
Vector3D& operator *= ( float f )  
{  
    x *= f;  
    y *= f;  
    z *= f;  
  
    return *this;  
}  
  
vector3D& operator /= ( const Vector3D& v )  
{  
    x /= v.x;  
    y /= v.y;  
    z /= v.z;  
  
    return *this;  
}  
  
Vector3D& operator /= ( float f )  
{  
    x /= f;  
    y /= f;
```

```
z /= f;

    return *this;
}

float& operator [] ( int index )
{
    return * ( index + &x );
}

float operator [] ( int index ) const
{
    return * ( index + &x );
}

int      operator == ( const Vector3D& v ) const
{
    return x == v.x && y == v.y && z == v.z;
}

int      operator != ( const Vector3D& v ) const
{
    return x != v.x || y != v.y || z != v.z;
}

operator float * ()
{
    return &x;
}

operator const float * () const
{
    return &x;
}

float      length () const
{
    return (float) sqrt ( x * x + y * y + z * z );
}
```

```
float    lengthSq () const
{
return x * x + y * y + z * z;
}

Vector3D&    normalize ()
{
    return (*this) /= length ();
}

float    maxLength () const
{
return max3 ( (float)fabs (x), (float)fabs (y),
              (float)fabs (z) );
}

float    distanceToSq ( const Vector3D& p ) const
{
return sqr ( x - p.x ) + sqr ( y - p.y ) + sqr ( z - p.z );
}

float    distanceTo ( const Vector3D& p ) const
{
return (float)sqrt ( sqr ( x - p.x ) + sqr ( y - p.y ) +
                     sqr ( z - p.z ) );
}

float    distanceToAlongAxis ( const Vector3D& p, int axis ) const
{
    return (float)fabs ( operator [] ( axis ) - p [axis] );
}

int     getMainAxis () const
{
    int     axis = 0;
    float  val   = (float) fabs ( x );

    for ( register int i = 1; i < 3; i++ )
    {
```

```
float vNew = (float) fabs ( operator [] ( i ) );

if ( vNew > val )
{
    val = vNew;
    axis = i;
}

return axis;
}

Vector3D& clamp ( float lower, float upper );

static Vector3D getRandomVector ( float len = 1 );

static inline Vector3D vmin ( const Vector3D& v1,
                             const Vector3D& v2 )

{
    return Vector3D ( v1.x < v2.x ? v1.x : v2.x,
                      v1.y < v2.y ? v1.y : v2.y,
                      v1.z < v2.z ? v1.z : v2.z );
}

static inline Vector3D vmax ( const Vector3D& v1,
                             const Vector3D& v2 )

{
    return Vector3D ( v1.x > v2.x ? v1.x : v2.x,
                      v1.y > v2.y ? v1.y : v2.y,
                      v1.z > v2.z ? v1.z : v2.z );
}

friend Vector3D operator + ( const Vector3D&, const Vector3D& );
friend Vector3D operator - ( const Vector3D&, const Vector3D& );
friend Vector3D operator * ( const Vector3D&, const Vector3D& );
friend Vector3D operator * ( float,           const Vector3D& );
friend Vector3D operator * ( const Vector3D&, float );
friend Vector3D operator / ( const Vector3D&, float );
friend Vector3D operator / ( const Vector3D&, const Vector3D& );
```

```
friend float operator & ( const Vector3D&, const Vector3D& );
friend Vector3D operator ^ ( const Vector3D&, const Vector3D& );

private:
float max3 ( float a, float b, float c ) const
{
return a > b ? (a > c ? a : (b > c ? b : c)) :
(b > c ? b : (a > c ? a : c));
}

float min3 ( float a, float b, float c ) const
{
return a < b ? (a < c ? a : (b < c ? b : c)) :
(b < c ? b : (a < c ? a : c));
}

float sqr ( float x ) const
{
return x*x;
}

};

inline Vector3D operator + ( const Vector3D& u, const Vector3D& v )
{
return Vector3D ( u.x + v.x, u.y + v.y, u.z + v.z );
}

inline Vector3D operator - ( const Vector3D& u, const Vector3D& v )
{
return Vector3D ( u.x - v.x, u.y - v.y, u.z - v.z );
}

inline Vector3D operator * ( const Vector3D& u, const Vector3D& v )
{
return Vector3D ( u.x*v.x, u.y*v.y, u.z * v.z );
}

inline Vector3D operator * ( const Vector3D& v, float a )
{
```

```
return Vector3D ( v.x*a, v.y*a, v.z*a );
}

inline Vector3D operator * ( float a, const Vector3D& v )
{
    return Vector3D ( v.x*a, v.y*a, v.z*a );
}

inline Vector3D operator / ( const Vector3D& u, const Vector3D& v )
{
    return Vector3D ( u.x/v.x, u.y/v.y, u.z/v.z );
}

inline Vector3D operator / ( const Vector3D& v, float a )
{
    return Vector3D ( v.x/a, v.y/a, v.z/a );
}

inline Vector3D operator / ( float a, const Vector3D& v )
{
    return Vector3D ( a / v.x, a / v.y, a / v.z );
}

inline float operator & ( const Vector3D& u, const Vector3D& v )
{
    return u.x*v.x + u.y*v.y + u.z*v.z;
}

inline Vector3D operator ^ ( const Vector3D& u,
                           const Vector3D& v )
{
    return Vector3D ( u.y*v.z-u.z*v.y, u.z*v.x-u.x*v.z,
                      u.x*v.y-u.y*v.x );
}

inline Vector3D lerp ( const Vector3D& a, const Vector3D& b, float t )
{
    return a + t * (b - a);
}
```

```
inline float mixedProduct ( const Vector3D& a, const Vector3D& b,
                           const Vector3D& c )
{
    return ( a & ( b ^ c ) );
}

inline bool areCollinear ( const Vector3D& a, const Vector3D& b,
                           const Vector3D& c )
{
    return ((b - a) ^ (c - a)).lengthSq () < EPS * EPS;
}

inline bool areComplanar ( const Vector3D& a, const Vector3D& b,
                           const Vector3D& c, const Vector3D& d )
{
    return fabs ( mixedProduct ( b - a, c - a, d - a ) ) < EPS * EPS * EPS;
}
```

Как видно из листинга, для этого класса переопределены все основные операторы унарные + и -, бинарные +, -, *, /, а также операторы +=, -=, *= и /=. Определены операторы сравнения == и !=.

Для доступа к компонентам можно использовать как имена компонентов, так и оператор [] .

Наличие в этом классе оператора, приводящего его к типам `float *` и `const float *`, позволяет легко использовать экземпляры этого класса в функциях, принимающих указатели на массивы компонентов.

Рассмотрим пример (листинг 6.2).

Листинг 6.2. Пример использования объектов класса `Vector3D` в командах OpenGL

```
Vector3D pos ( 1, 1, 0 );
Vector3D offs ( 1, 0, 0 );

glVertex3fv ( pos );
glVertex3fv ( pos + offs );
```

В табл. 6.1 приведены основные методы класса `Vector3D`.

Таблица 6.1. Методы класса Vector3D

Метод	Описание
float length () const	Вычисляет длину вектора
float lengthSq () const	Вычисляет квадрат длины вектора
Vector3D& normalize ()	Нормирует вектор и возвращает ссылку на него
float maxLength () const	Вычисляет максимум из модулей компонентов
float distanceToSq (const Vector3D& p) const	Вычисляет квадрат расстояния до заданного вектора
float distanceTo (const Vector3D& p) const	Вычисляет расстояние до заданного вектора
float distanceToAlongAxis (const Vector3D& p, int axis) const	Вычисляет расстояние до заданного вектора вдоль одной из координатной оси (расстояние между проекциями на эту ось)
int getMainAxis () const	Возвращает ось, для которой соответствующий компонент имеет наибольшее по модулю значение
Vector3D& clamp (float lower, float upper)	Приводит все компоненты вектора к отрезку [lower, upper], т. е. каждый компонент (<i>v</i>) преобразуется по правилу: Min(max(lower, v), upper)
static Vector3D getRandomVector (float len = 1)	Возвращает случайный вектор заданной длины <i>len</i>
Vector3D vmin (const Vector3D& v1, const Vector3D& v2)	Возвращает покомпонентный минимум из двух векторов
Vector3D vmax (const Vector3D& v1, const Vector3D& v2)	Возвращает покомпонентный максимум из двух векторов

Также определен ряд функций, работающих с экземплярами класса Vector3D (табл. 6.2). К ним относятся перегруженные бинарные операторы & и ^, служащие для обозначения скалярного и векторного произведений двух векторов (векторное произведение определено только для трехмерных векторов).

Таблица 6.2. Методы для работы с экземплярами класса *Vector3D*

Метод	Описание
<code>Vector3D lerp (</code> <code>const Vector3D& a,</code> <code>const Vector3D& b, float t)</code>	Выполняет линейную интерполяцию между двумя векторами. Возвращает значение $a + t * (b - a)$
<code>float mixedProduct (</code> <code>const Vector3D& a,</code> <code>const Vector3D& b,</code> <code>const Vector3D& c)</code>	Вычисляет смешанное произведение трех векторов. Возвращает значение $(a, [b, c])$
<code>bool areCollinear (</code> <code>const Vector3D& a,</code> <code>const Vector3D& b,</code> <code>const Vector3D& c)</code>	Проверяет, являются ли два вектора коллинеарными (т. е. параллельными)
<code>bool areComplanar (</code> <code>const Vector3D& a,</code> <code>const Vector3D& b,</code> <code>const Vector3D& c,</code> <code>const Vector3D& d)</code>	Проверяет, являются ли три вектора компланарными (т. е. лежащими в одной плоскости)

Работа с матрицами

Для работы с трехмерными матрицами используется класс *Matrix3D*, а для работы с четырехмерными матрицами — класс *Matrix4x4*.

У каждого из этих классов имеется пустой конструктор, не выполняющий никакой инициализации, сору-конструктор, конструктор, принимающий на вход одно вещественное число (в этом случае диагональные элементы матрицы инициализируются этим числом, все остальные — нулем), набором из 3 или 4 вещественных чисел, служащих для инициализации диагональных элементов (остальные элементы инициализируются нулем).

У класса *Matrix3D* есть конструктор, строящий ее по трем столбцам.

Для этих классов определены операторы сложения и вычитания (+ и -), умножения (на число и на матрицу), деления (на число), присваивания.

Определены операторы [], возвращающие указатель на строку с заданным номером, что позволяет обращаться к элементам матрицы в обычной форме `m[i][j]` — сначала срабатывает оператор [], возвращающий указатель на массив, к которому применяется [j].

Основные методы этих классов приведены в табл. 6.3.

Таблица 6.3. Основные методы классов *Matrix3D* и *Matrix4x4*

Метод	Описание
M& invert ()	Обращает матрицу и возвращает ссылку на нее
M& transpose ()	Транспонирует матрицу и возвращает ссылку на нее
M inverse () const	Возвращает матрицу, обратную к данной (сама матрица не изменяется)
void getHomMatrix (float * matrix) const	Записывает данную матрицу в используемую в OpenGL матрицу 4×4
static M identity ()	Возвращает единичную матрицу
static M scale (const Vector3D&)	Возвращает матрицу, соответствующую масштабированию с коэффициентами, задаваемыми компонентами вектора
static M rotateX (float)	Возвращает матрицу поворота вокруг оси <i>Ox</i> на заданный угол
static M rotateY (float)	Возвращает матрицу поворота вокруг оси <i>Oy</i> на заданный угол
static M rotateZ (float)	Возвращает матрицу поворота вокруг оси <i>Oz</i> на заданный угол
static M rotate (const Vector3D&, float)	Возвращает матрицу поворота вокруг оси, задаваемой вектором направления, на заданный угол
static M rotate (float yaw, float pitch, float roll)	Возвращает матрицу поворота, соответствующую трем углам Эйлера
static M mirrorX ()	Возвращает матрицу отражения относительно плоскости <i>Oyz</i>
static M mirrorY ()	Возвращает матрицу поворота относительно плоскости <i>Oxz</i>
static M mirrorZ ()	Возвращает матрицу поворота относительно плоскости <i>Oxy</i>

У класса *Matrix4x4* есть два метода для преобразования трехмерных векторов — как направления в пространстве и как точки в пространстве (точки и направления преобразуются по-разному, так матрица переноса изменяет точку, но не меняет направления):

```
Vector3D transformPoint ( const Vector3D& b ) const
Vector3D transformDirection ( const Vector3D& b ) const
```

Для преобразования трехмерных векторов при помощи объектов класса Matrix3D используется перегруженный оператор умножения:

```
Matrix3D m ( Matrix3D :: rotateX ( 0.7 ) );
Vector3D v ( 1, 0, 1 );
Vector3D w;

w = m*v;
```

Работа с кватернионами

Для работы с кватернионами предназначен класс Quaternion. Для этого класса определены все стандартные математические операторы. Методы класса Quaternion описаны в табл. 6.4.

Таблица 6.4. Методы класса Quaternion

Метод	Описание
Quaternion ()	Строит кватернион без инициализации компонентов
Quaternion (float theX, float theY = 0, float theZ = 0, float theW = 0)	Строит кватернион по четырем значениям его компонентов
Quaternion (const Quaternion& q)	Copy-конструктор
Quaternion (const Vector3D& v)	Первые три компонента берутся из переданного вектора, последний задается равным нулю
Quaternion (float angle, const Vector3D& axis)	Строит кватернион, соответствующий повороту вокруг оси, задаваемой вектором, на заданный угол
Quaternion (const Matrix3D& mat)	Строит кватернион, соответствующий переданной матрице поворота
Quaternion (float mat [3][3])	Строит кватернион, соответствующий переданной матрице поворота
Quaternion& conj ()	Производит сопряжение кватерниона и возвращает ссылку на него
Vector3D rotate(const Vector3D& v) const	Выполняет поворот трехмерного вектора при помощи данного кватерниона
Quaternion& normalize ()	Нормирует кватернион и возвращает ссылку на него

Таблица 6.4 (окончание)

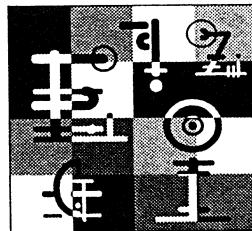
Метод	Описание
Quaternion& initWithAngles (float yaw, float pitch, float roll)	Инициализирует кватернион поворотом по заданным углам Эйлера
Matrix3D getMatrix () const	Возвращает матрицу поворота, соответствующего данному кватерниону
Void getHomMatrix (float * m) const	Возвращает однородную матрицу 4×4 , используемую в OpenGL, соответствующую данному кватерниону

Для интерполяции кватернионов используется так называемая *сферическая линейная интерполяция* (slerp). Для вводимого класса она реализуется функцией slerp:

Quaternion slerp (const Quaternion& q1, const Quaternion& q2, float t);

Здесь кватернионы q1 и q2 задают начальную и конечную ориентацию, а параметр t, принимающий значения из промежутка [0, 1], является параметром интерполяции.

Метод slerp позволяет осуществлять плавный переход от одной ориентации к другой. Использовать линейную интерполяцию матриц поворотов для этой цели нельзя, т. к. получающиеся при интерполяции матрицы могут уже не являться матрицами поворота (более того, они могут даже быть вырожденными).



Глава 7

Работа с расширениями, библиотека libExt

В главе 1 уже рассматривалась роль расширений OpenGL. Здесь будет подробно рассмотрена работа с расширениями для различных платформ и показано использование для этой цели библиотеки libExt. Дело в том, что работа с расширениями имеет свою специфику для различных платформ. Связано это с необходимостью получать адреса функций (предоставляемых драйвером). Получение информации о платформо-зависимых расширениях тоже имеет определенную специфику.

Как уже говорилось, каждое расширение однозначно идентифицируется его именем. Полный список всех известных расширений с описанием можно найти в Интернете по адресу <http://oss.sgi.com/projects/ogl-sample/registry>. Каждое расширение имеет свое описание, в котором в частности перечислены все вводимые данным расширением константы и функции. Для удобства работы все константы и прототипы вводимых расширениями функций содержатся в постоянно обновляемых файлах glext.h (общие расширения), wglext.h (расширения Windows) и glxext.h (расширения для X Window System). Последние версии всех этих файлов можно скачать в Интернете по адресу <http://oss.sgi.com/projects/ogl-sample/registry/>.

Если наличие этих файлов позволяет сразу же использовать вводимые расширениями константы, то с вводимыми функциями несколько сложнее. В этих заголовочных файлах для каждой вводимой функции содержится описание типа — указателя на эту функцию. Так, например, для функции glFogCoordEXT, вводимой расширением GL_EXT_fog_coord, определяется следующий тип:

```
typedef void (APIENTRY * PFNGLFOGCOORDFEXTPROC) (GLfloat coord);
```

Таким образом, тип PFNGLFOGCOORDFEXTPROC является типом адреса указателя на функцию glFogCoordEXT.

Если завести переменную glFogCoordEXT, определив ее как

```
PFNGLFOGCOORDFEXTPROC glFogCoordfEXT;
```

и потом присвоить ей значение адреса этой функции, то обращение

```
glFogCoordfEXT (0.5f);
```

и будет корректным обращением к функции `glFogCoordfEXT`, вводимой данным расширением. При этом подобное обращение к этой переменной выглядит как работа с любой стандартной функцией OpenGL.

К сожалению, механизм получения адресов вводимых расширениями функций по их именам сильно зависит от конкретной платформы. Так, в Windows для этого используется функция `wglGetProcAddress`, а в X Window System — функция `glXGetProcAddressARB`.

Аналогично, хотя для получения списка всех поддерживаемых платформо-независимых расширений (GL) можно использовать вызов `glGetString (GL_EXTENSIONS)`, получение списка расширений для конкретной платформы также сильно зависит от самой платформы (листинг 7.1).

Листинг 7.1. Получение списков общих и платформо-зависимых расширений

```
const char * glExts = (const char *) glGetString ( GL_EXTENSIONS );
const char * plExts = «»;

#ifndef _WIN32           // check Windoze extensions
    plExts = wglGetExtensionsStringARB ( wglGetCurrentDC () );
#else                   // check GLX extensions
    Display * display = glXGetCurrentDisplay ();
    int      screen   = DefaultScreen      ( display );
    plExts = glXQueryExtensionsString ( display, screen );
#endif
```

Для упрощения работы с расширениями есть различные библиотеки (например GLEW, libExt), одну из которых — libExt — мы и рассмотрим далее.

Эта написанная автором библиотека работает как под Microsoft Windows, так и под Linux, предоставляя удобный и простой доступ к большому набору расширений. В листинге 7.2 описаны функции, вводимые этой библиотекой.

Листинг 7.2. Функции, вводимые библиотекой libExt

```
bool    isExtensionSupported  (const char * ext);
void    assertExtensionsSupported (const char * extList);
void    initExtensions        ();
void    printfInfo            ();
const char * getGeneralExtensions ();
const char * getPlatformExtensions ();
```

Функция `isExtensionSupported` проверяет, поддерживает ли расширение с заданным именем. При этом проверяются как общие, так и платформо-зависимые расширения. В случае наличия поддержки расширения функция возвращает значение `true`. Также эта функция (как впрочем и все остальные функции этой библиотеки) инициализирует библиотеку (вызовом функции `initExtensions`), если это еще не было сделано.

Функция `assertExtensionsSupported` (листинг 7.3) проверяет поддержку сразу нескольких расширений. В случае, если хотя бы одно из расширений из списка не поддерживается, на стандартное устройство вывода сообщений об ошибках `stderr` выводится диагностическое сообщение с именем не поддерживаемого расширения и выполнение программы прерывается. На вход функции поступает строка, состоящая из имен расширений, разделенных пробелами, запятыми (,), точками с запятой (;) или символами табуляции.

Листинг 7.3. Пример использования функции `assertExtensionsSupported`

```
init          ();
initExtensions ();
printfInfo   ();

assertExtensionsSupported ( "GL_ARB_shading_language_100 \
                           GL_ARB_shader_objects" );
```

Функция `printfInfo` печатает основную информацию об OpenGL, используемом графическом ускорителе и его драйвере, а также об основных расширениях.

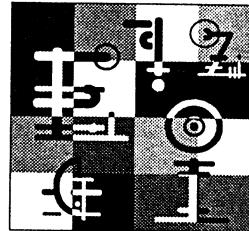
Функция `getGeneralExtensions` возвращает полный список всех поддерживаемых платформо-независимых расширений в виде строки (в которой имена отдельных расширений разделены пробелами).

Функция `getPlatformExtensions` возвращает полный список всех поддерживаемых расширений, специфичных для данной платформы (для Windows возвращается список всех поддерживаемых WGL-расширений, для X Window System — список всех GLX-расширений).

Функция `initExtensions` предназначена для явной инициализации библиотеки (она также вызывается при обращении к любой из рассмотренных функций). Обратите внимание, что поскольку именно эта функция отвечает за получение адресов вводимых поддерживаемыми расширениями функций, то перед использованием вводимых расширением функций обязательно следует ее вызвать (если она еще не была вызвана).

Примечание

Во избежание проблем при обращении к функциям не поддерживаемых расширений рекомендуется всегда в начале программы проверять поддержку необходимых расширений.



Глава 8

Библиотеки libTexture и libTexture3D

К сожалению, ни сама библиотека OpenGL, ни библиотека GLUT не поддерживают загрузку текстур из файлов. Только в библиотеке GLAUX есть возможность загрузки некоторых типов BMP-файлов.

Для облегчения работы с загрузкой текстур из файлов есть различные библиотеки (например, библиотека Devil, доступная в Интернете по адресу <http://openil.sourceforge.net>). Далее будут рассмотрены написанные автором библиотеки libTexture и libTexture3D, предназначенные для загрузки текстур (в том числе, трехмерных и кубических текстурных карт) из файлов разных типов. Кроме того, эти библиотеки позволяют сохранять как содержимое текущего окна (в файле типа TGA), так и трехмерные текстуры (в файлах типа DDS).

Библиотека libTexture поддерживает загрузку текстур из таких форматов, как BMP, TGA, JPG (JPEG), PNG и DDS. Для DDS-файлов поддерживаются сжатые текстуры, кубические текстурные карты (cube texture maps) и трехмерные текстуры. Кроме того, библиотека libTexture позволяет загружать текстуры (и другие данные) из ZIP- и RAR-архивов. Также библиотека может осуществлять поиск заданного файла в нескольких местах (включая содержимое архивов).

И библиотека libTexture и библиотека libTexture3D переносимы — они компилируются и работают под управлением как Microsoft Windows, так и Linux.

К ним легко добавить как поддержку новых форматов текстур, так и новые источники данных, даже не изменяя при этом исходные тексты библиотек.

Для обеспечения гибкости библиотеки понятия *источника данных* (файл, архив и т. п.) и *декодирования данных* разделяются друг от друга.

Внутри библиотека поддерживает список источников данных, т. е. объектов, способных по имени файла дать его образ (в виде объекта класса `Data`). Каждый такой источник данных представляется экземпляром класса, унаследованного от абстрактного класса `FileSystem` (рис. 8.1, листинг 8.1).

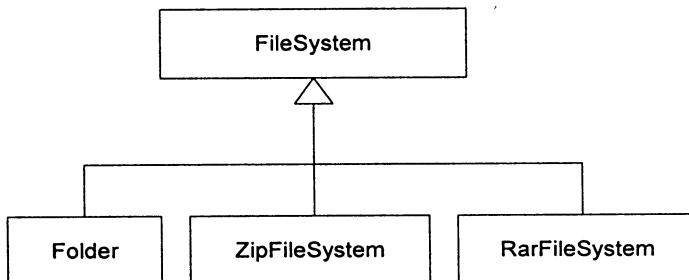


Рис. 8.1. Иерархия источников данных в библиотеке *libTexture*

Листинг 8.1. Описание класса **FileSystem**

```

class FileSystem
{
public:
    FileSystem () {}
    virtual ~FileSystem () {}

    virtual bool isOk () const
    {
        return false;
    }

    virtual Data * getFile ( const string& name ) = 0;
};

  
```

Основной метод этого класса (*getFile*) по имени файла возвращает либо указатель на объект класса *Data*, содержащий образ этого файла, либо значение *NULL*, если этот файл не найден.

Класс *Folder* позволяет получить доступ к файлу относительно заданного каталога (например, внутри него).

Классы *ZipFileSystem* и *RarFileSystem* позволяют получать доступ к файлам, содержащимся в ZIP- и RAR-архивах.

Для добавления новых источников данных (по умолчанию всегда присутствует один — относительно текущего каталога) предназначены следующие функции:

```

bool addFileSystem ( FileSystem * fileSystem );
bool addZipFileSystem ( const char * fileName );
bool addSearchPath ( const char * path );
  
```

Функция `addFileSystem` позволяет добавить к списку источников данных новый источник (в том числе, созданный пользователем).

Функция `addZipFileSystem` добавляет в качестве источника данных ZIP-архив.

Функция `addRARFileSystem` добавляет в качестве источника данных RAR-архив.

Функция `addSearchPath` добавляет новый путь для поиска данных (добавляя экземпляр класса `Folder`).

Точно такая же схема используется и для декодирования текстур из конкретного формата (в котором они содержатся в файле данных) в образ, пригодный для загрузки средствами OpenGL. Такой образ представляется экземплярами классов `Texture` (листинг 8.2) и `CompressedTexture` (рис. 8.2).

Листинг 8.2. Описание класса `Texture`

```
class Texture
{
protected:
    int      width;
    int      height;
    int      numComponents;
    int      format;
    byte    * data;
    int      levels;
    bool     compressed;

public:
    Texture ();
    Texture ( int theWidth, int theHeight, int theNumComponents );
    virtual ~Texture ();

    int getWidth () const
    {
        return width;
    }

    int getHeight () const
    {
        return height;
    }
}
```

```
}

int getNumComponents () const
{
    return numComponents;
}

int getFormat () const
{
    return format;
}

int getLevels () const
{
    return levels;
}

bool    isCompressed () const
{
    return compressed;
}

int getBytesPerLine () const
{
    return width * numComponents;
}

byte * getData () const
{
    return data;
}

void    setFormat ( int newFormat )
{
    format = newFormat;
}

void    putLine    ( int y, dword * bits );

virtual bool    upload ( int target, bool mipmap = true );
};
```

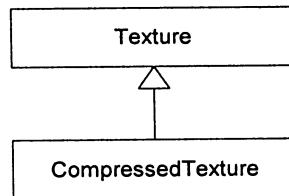


Рис. 8.2. Иерархия классов Texture и CompressedTexture

Класс CompressedTexture предназначен для хранения сжатых (с помощью метода s3tc) текстур.

Абстрактный класс TextureLoader (рис. 8.3, листинг 8.3) предназначен для построения загружаемого образа (в виде экземпляра класса Texture или CompressedTexture) по данным из файла (в виде экземпляра класса Data).

Листинг 8.3. Описание класса TextureLoader

```

class TextureLoader
{
public:
    TextureLoader () {}
    virtual ~TextureLoader () {}

    virtual Texture * load ( Data * data ) = 0;
};

  
```

Метод load и служит для построения загружаемого образа текстуры по данным из файла.

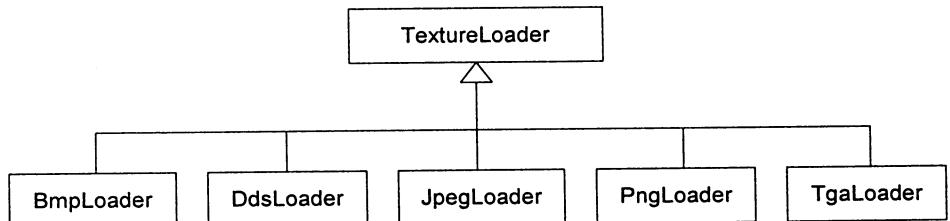


Рис. 8.3. Иерархия классов для загрузки текстур

Библиотека libTexture поддерживает набор декодеров, идентифицируемых по расширению имени файла (BMP, JPG, PNG и т. д.).

Если требуется добавить поддержку нового формата текстуры, достаточно создать соответствующий класс для его загрузки, унаследовав этот класс от TextureLoader, и добавить в список загрузчиков при помощи функции addDecoder:

```
bool addDecoder ( const char * ext, TextureLoader * decoder );
```

Параметр `ext` задает расширение имени файла, для которого устанавливается данный загрузчик, параметр `decoder` указывает на загрузчик.

Следующая группа функций библиотеки libTexture предназначена для создания текстур различных типов по имени файла. Возвращается идентификатор загруженной текстуры в OpenGL или нуль (0), если загрузить текстуру не удалось.

```
unsigned createNormalizationCubemap ( int size,
                                      bool mipmap = false );
unsigned createCubeMap   ( bool mipmap, const char * maps [] );
unsigned createCubeMap   ( bool mipmap, const char * fileName );
unsigned createTexture2D ( bool mipmap, const char * fileName );
unsigned createTexture1D ( bool mipmap, const char * fileName );

unsigned createNormalMapFromHeightMap ( bool mipmap,
                                         const char * fileName, float scale );
unsigned createNormalMap ( bool mipmap,
                           const char * fileName );
```

Функции `createTexture1D` и `createTexture2D` загружают одно- и двухмерные текстуры из файла с заданным именем.

Функции `createCubeMap` загружают кубическую текстурную карту либо по списку из 6 имен файлов (по одному для каждой грани), либо по имени DDS-файла (из поддерживаемых форматов только эти файлы могут содержать кубические текстурные карты).

Функция `createNormalizationCubemap` предназначена для создания нормирующей кубической текстурной карты по заданному размеру грани `size`.

Функция `createNormalMap` загружает карту нормалей, помещая в ее альфа-канал значение, равное единице.

Функция `createNormalMapFromHeightMap` позволяет по имени файла с картой высот построить соответствующую этой карте высот карту нормалей. Параметр `scale` управляет степенью неровности карты — большим значениям соответствуют более неровные карты нормалей, меньшим значениям — более гладкие карты.

Кроме этого, библиотека libTexture вводит еще целый ряд полезных функций:

```
Data      * getFile     ( const char * fileName );
Texture  * getTexture  ( const char * fileName );
```

```

bool      saveScreenShot ( const char * fileName );
bool      fileExist      ( const char * fileName );
string    getPath        ( const string& fullName );
string    getFileName     ( const string& fullName );
string    getName         ( const string& fullName );
string    buildFileName   ( const string& path, const string& name );

```

Функция `getFileName` позволяет по имени получить доступ к содержимому файла, при этом для поиска файла использует список зарегистрированных источников данных.

Функция `getTexture` позволяет по имени файла получить образ текстуры, пригодный для загрузки средствами OpenGL.

Функция `saveScreenshot` позволяет сохранить содержимое текущего активного окна в виде TGA-файла с заданным именем `fileName`.

Следующая группа функций предназначена для работы с именами файлов.

Функция `fileExist` проверяет, существует ли файл с заданным именем `fileName`. При проверке на существование не используются зарегистрированные источники данных.

Функция `getPath` по полному имени файла `fullName` возвращает путь к нему.

Функция `getFileName` по полному имени файла `fullName` возвращает его имя (с расширением).

Функция `getName` по полному имени файла `fullName` возвращает его имя (без пути и расширения).

Функция `buildFileName` по пути к файлу `path` и его имени `name` строит полное имя (с путем).

В листинге 8.4 приведено описание функций, вводимых библиотекой `libTexture3D`.

Листинг 8.4. Функции, вводимые библиотекой `libTexture3D`

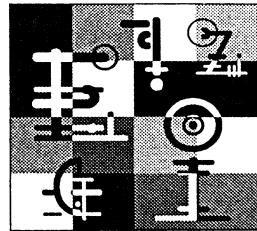
```

bool      saveTexture3D   ( int width, int height, int depth,
                           int components, const byte * data,
                           const char * fileName );
unsigned createTexture3D ( bool mipmap, const char * fileName );
unsigned createTexture3D ( bool mipmap, Data * data );

```

Функция `saveTexture3D` сохраняет трехмерную текстуру размером `width*height*depth` с `components` компонентов на тексел в файле с заданным именем в формате DDS.

Функции `createTexture3D` предназначены для загрузки в память графического ускорителя трехмерной текстуры из файла формата DDS. Входным параметром может быть как имя файла, так и объект класса `Data`.



Глава 9

Реализация *p*-буфера в виде класса, использование расширения `EXT_framebuffer_object` для создания внеэкранных буферов

Традиционные средства OpenGL позволяют осуществлять рендеринг сцены (построение ее изображения) с непосредственным выводом на экран (т. е. в основной фреймбуфер). Однако во многих случаях возникает необходимость записать результаты рендеринга не в основной фреймбуфер (соответствующий какому-либо окну или всему экрану), а в специальную текстуру, которую можно будет использовать в дальнейшем именно как текстуру (например, если надо имитировать отражение сцены в воде с мелкой рябью или искажения, вызванные горячим воздухом, применить различные пост-эффекты и создать специальные текстуры для их использования в дальнейшем рендеринге).

Примечание

Фреймбуфер, главный/основной фреймбуфер (буфер кадра, framebuffer) — часть памяти графического процессора, выделенная под хранение строящегося изображения. Содержит в себе значения цветов для всех пикселов изображения.

Текстура — отдельное изображение, заданное при помощи матрицы текстелов (элемент текстуры называется текстелом), используемое для наложения на поверхность какого-либо объекта при его рендеринге.

Рендеринг в текстуру (Render-To-Texture, RTT) — рендеринг объектов, осуществляемый не в фреймбуфер, а в специальную область памяти графического процессора, которая в дальнейшем используется как обычная текстура.

Есть несколько приемов, позволяющих получить результаты рендеринга в виде текстуры. Простейшим из них является использование функции `glReadPixels`, позволяющей прочитать изображение из фреймбуфера непосредственно в память CPU. После этого данное изображение можно будет

загрузить в текстуру (например, при помощи функции `glTexImage2D`). Однако такой подход требует копирования сначала из фреймбуфера в память центрального процессора, а затем — из памяти центрального процессора в память графического процессора (рис. 9.1).

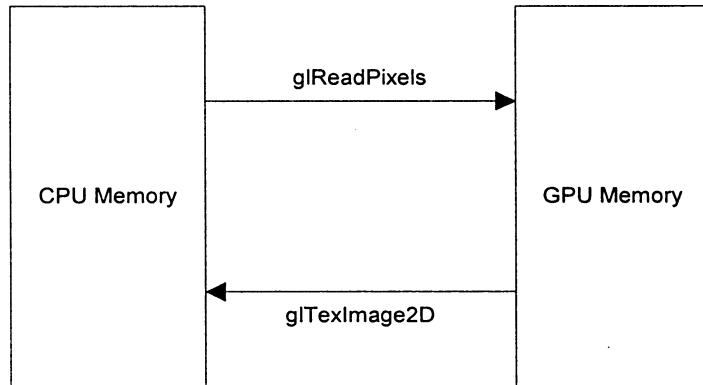


Рис. 9.1. Рендеринг в текстуру посредством функции `glReadPixels`

Подобные операции копирования являются довольно длительными, поэтому этот путь малоприемлем из-за больших временных затрат на копирование данных между графическим ускорителем и CPU.

Гораздо удобнее при помощи функции `glCopyTexSubImage2D` переписать прямоугольный фрагмент изображения из фреймбуфера сразу в текстуру, т. е. копирование происходит из видеопамяти в видеопамять минуя при этом центральный процессор (рис. 9.2).

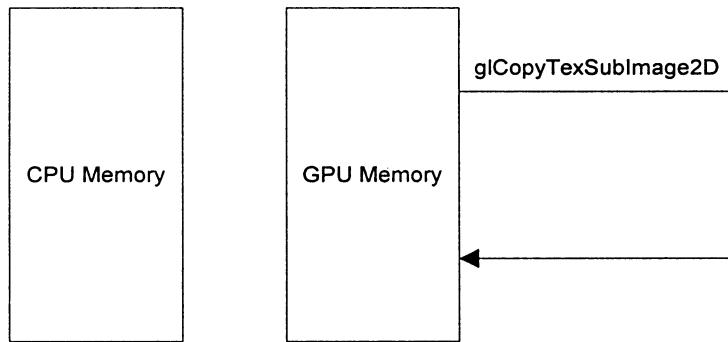


Рис. 9.2. Рендеринг в текстуру через `glCopyTexSubImage2D`

Несмотря на то что этот подход гораздо эффективнее предыдущего, он все равно требует явного копирования памяти, а также использования того же самого фреймбуфера, что и для построения главного изображения.

В ряде случаев было бы гораздо удобнее создать себе "виртуальный" буфер, в который можно было бы записывать результаты рендеринга, однако обладающий другими размером и атрибутами и никак не связанный с основным фреймбуфером. Также было бы очень удобно, если бы этот буфер можно было бы непосредственно использовать в качестве данных для текстуры, вообще избегая тем самым копирования памяти.

Все эти возможности предоставляет использование так называемых *p*-буферов (*p-buffer*). При этом на современных графических ускорителях *p*-буфера хранятся непосредственно в видеопамяти и процесс рендеринга в них аппаратно ускорен.

К сожалению, работа с *p*-буфером на платформах Windows и Linux заметно отличается. Поэтому мы сначала рассмотрим каждую из них отдельно, а затем приведем пример кроссплатформенного использования *p*-буфера.

Также подобную возможность предоставляет расширение `EXT_framebuffer_object`. Его применение во многих случаях оказывается более удобным и выгодным, чем использование *p*-буферов, однако это расширение пока еще поддерживается не всеми картами.

В этой главе мы сначала рассмотрим работу с *p*-буфером (отдельно для Windows и Linux) и построим его реализацию в виде класса, а затем рассмотрим расширение `EXT_framebuffer_object` и для него также построим соответствующую реализацию в виде класса на языке C++.

Работа с *p*-буфером в Microsoft Windows

Для создания и работы с *p*-буферами под управлением ОС Microsoft Windows необходима поддержка сразу двух расширений `WGL_ARB_pixel_format` и `WGL_ARB_pbuffer`. Для использования *p*-буфера в качестве источника данных для текстуры также требуется поддержка расширения `WGL_ARB_render_texture`.

Процесс работы с *p*-буфером можно разделить на следующие шаги:

1. Непосредственное создание *p*-буфера.
2. Выбор *p*-буфера как текущей цели для рендеринга.
3. Уничтожение *p*-буфера.

Непосредственное создание *p*-буфера

Для создания *p*-буфера в первую очередь следует получить допустимый контекст текущего устройства:

```
HDC hDc = wglGetCurrentDC();
```

Далее необходимо получить подходящий формат пикселов для создания *p*-буфера. Для этого предназначена вводимая расширением WGL_ARB_pixel_format функция wglChoosePixelFormat:

```
BOOL wglChoosePixelFormatARB ( HDC hDc, const int * intAttrs,
                           const float * floatAttrs,
                           UINT maxFormats, int * formats,
                           UINT * numFormats );
```

В параметрах *intAttrs* и *floatAttrs* передаются списки целочисленных и вещественных атрибутов. Каждый такой атрибут задается парой значений (имя, значение), где сначала идет идентификатор (имя) атрибута, а затем – его значение; список завершается нулевым значением (идентификатором атрибута, равным нулю). Параметр *maxFormats* сообщает, какое количество целочисленных идентификаторов формата можно записать в массив *formats*. Последний параметр *numFormats* является указателем целочисленной переменной, в которой будет возвращено количество подходящих форматов (это значение может быть меньше, чем *maxFormats*).

Таблица 9.1. Основные атрибуты для функции *wglChoosePixelFormatARB*

Имя	Значение
WGL_COLOR_BITS_ARB	Минимальное общее количество бит для цвета
WGL_STENCIL_BITS_ARB	Минимальное количество бит для значения в буфере трафарета
WGL_AUX_BUFFERS_ARB	Минимальное количество дополнительных буферов
WGL_SUPPORT_OPENGL_ARB	Должен ли формат поддерживать вывод средствами OpenGL
WGL_DRAW_TO_PBUFFER_ARB	Ненулевое значение данного атрибута говорит о том, что вывод будет направляться в <i>p</i> -буфер
WGL_ACCUM_BITS_ARB	Общее количество бит под значение в буфере накопления
WGL_DOUBLE_BUFFER_ARB	Ненулевое значение говорит о необходимости создания второго буфера
WGL_ALPHA_BITS_ARB	Минимальное количество для значения в альфа-канале
WGL_DEPTH_BITS_ARB	Минимальное количество бит для значения в буфере глубины
WGL_PIXEL_TYPE_ARB	Тип значений пикселов, принимает одно из двух значений – WGL_TYPE_RGBA_ARB (для RGBA-режима) или WGL_TYPE_COLOR_INDEX_ARB (для палитрового режима)

Таблица 9.1 (окончание)

Имя	Значение
WGL_RED_BITS_ARB	Минимальное количество бит для значения в красном канале
WGL_GREEN_BITS_ARB	Минимальное количество бит для значения в зеленом канале
WGL_BLUE_BITS_ARB	Минимальное количество бит для значения в синем канале

Данная функция возвращает нулевое значение в случае ошибки и ненулевое, если вызов прошел успешно. В случае, если получен хотя бы один идентификатор формата, его можно использовать в дальнейшем для создания *p*-буфера.

Для этого предназначена функция `wglCreatePbufferARB`:

```
HPBUFFER wglCreatePbufferARB ( HDC hDC, int format, int width,
                               int height, const int * attrs );
```

Параметры: `hDC` — контекст устройства, полученный на предыдущем шаге; `format` — идентификатор формата пикселов для создаваемого *p*-буфера; `width` и `height` — задают желаемый размер *p*-буфера в пикселях; `attrs` — передает дополнительные целочисленные параметры, аналогично функции `wglChoosePixelFormatARB`.

Данная функция возвращает указатель (`handle`) созданного *p*-буфера или значение `NULL` в случае ошибки.

Размер созданного *p*-буфера может отличаться от запрашиваемого, поэтому после создания *p*-буфера следует узнать его истинные размеры при помощи функции `wglQueryPbufferARB`:

```
int width;
int height;
wglQueryPbufferARB ( hBuffer, WGL_PBUFFER_WIDTH_ARB, &width );
wglQueryPbufferARB ( hBuffer, WGL_PBUFFER_HEIGHT_ARB, &height );
```

После непосредственного создания *p*-буфера нужно получить контекст устройства для него при помощи функции `wglGetPbufferDCARB`:

```
HDC wglGetPbufferDCARB ( HPBUFFER hBuffer );
```

Заключительным шагом создания *p*-буфера является создание контекста рендеринга для него. Это можно сделать при помощи функции `wglCreateContext`:

```
HGLRC wglCreateContext ( HDC hBufDC );
```

Выбор *p*-буфера как текущей цели для рендеринга

Направление вывода в заданный *p*-буфер выполняется с помощью функции `wglMakeCurrent`:

```
BOOL wglMakeCurrent ( HDC hBufDc, HGLRC hBufRc );
```

Параметры `hBufDc` и `hBufRc` — это контекст устройства и контекст рендеринга для данного *p*-буфера.

Для того чтобы снова направить вывод в окно, следует вызывать функцию `wglMakeCurrent` с параметрами, соответствующими окну, в которое надо перенаправить вывод. Работая с библиотекой GLUT, можно для этой цели использовать функцию `glutSetWindow`, передав ей целочисленный идентификатор окна.

Уничтожение *p*-буфера

Для того чтобы уничтожить *p*-буфер (т. е. освободить связанные с ним ресурсы), необходимо выполнить следующие три шага:

1. Уничтожить контекст рендеринга для *p*-буфера.
2. Освободить контекст устройства *p*-буфера.
3. Уничтожить сам *p*-буфер.

Все эти действия выполняет следующий фрагмент кода:

```
wglDeleteContext      ( hGlRc );
wglReleasePbufferDCARB ( hBuffer, hDc );
wglDestroyPbufferARB   ( hBuffer );
```

Обработка переключения видеорежима

В случае переключения видеорежима может произойти нарушение целостности *p*-буфера, т. е. он фактически становится негодным для дальнейшего использования.

Для того чтобы определить, пригоден данный *p*-буфер для использования по-прежнему или нет, можно использовать следующий фрагмент кода:

```
int flag;
wglQueryPbufferARB ( hBuffer, WGL_PBUFFER_LOST_ARB, &flag );
```

Если после выполнения этого фрагмента кода значение переменной `flag` окажется отличным от нуля, это значит, что *p*-буфер был поврежден и больше не является пригодным к использованию. В этом случае его следует уничтожить и создать заново рассмотренным ранее способом.

Копирование данных из *p*-буфера в текстуру

После того как *p*-буфер был успешно создан и в него был произведен вывод, можно использовать функцию `glCopyTexSubImage2D` для копирования содержимого *p*-буфера (или только его части) в текстуру.

Однако не забывайте, что на платформе Windows различные контексты рендеринга имеют свои наборы атрибутов, в том числе, свои наборы дисплейных списков, текстур и т. п. Поэтому если вы хотите записывать в *p*-буфер результаты рендеринга с использованием какой-либо текстуры, то ее следует явно загрузить в этом контексте.

Впрочем, вместо этого можно воспользоваться функцией `wglShareLists`, позволяющей перенести дисплейные списки и текстуры из одного контекста рендеринга в другой:

```
BOOL wglShareLists( HGLRC hG1RcFrom, HGLRC hG1RcTo );
```

Параметры `hG1RcFrom` и `hG1RcTo` задают исходный контекст (из которого надо перенести дисплейные списки и текстуры) и контекст, в который надо осуществить перенос.

Связывание *p*-буфера с текстурой

При наличии поддержки расширения `WGL_ARB_render_texture` можно непосредственно использовать *p*-буфер как источник данных для текстуры, не прибегая к копированию пикселов.

В этом случае сам *p*-буфер привязывается (`bind`) к соответствующей текстуре и во время этой привязки текстура содержит содержимое цветового буфера данного *p*-буфера, а сам *p*-буфер не доступен для рендеринга. Когда требуется произвести вывод в *p*-буфер, его сперва нужно "освободить" от данной текстуры.

Можно также связать *p*-буфер с кубической картой, тогда при рендеринге *p*-буферу сообщается, в какую грань куба идет вывод.

Можно также использовать пирамидальное фильтрование, однако для этого надо либо явно задавать все промежуточные уровни, либо воспользоваться рассмотренным ранее расширением `GL_SGIS_generate_mipmap`, обеспечивающим автоматическое построение всех промежуточных уровней для заданной текстуры.

Если *p*-буфер привязан к текстуре и в это время из-за переключения видео режима сам *p*-буфер перестал быть пригодным для использования, то несмотря на это содержимое текстуры по-прежнему остается пригодным для использования и не разрушается.

При освобождении *p*-буфера от текстуры содержимое его цветового буфера может стать неопределенным.

Чтобы можно было использовать данный *p*-буфер для привязки к текстуре, требуется задать дополнительные атрибуты в функциях `wglChoosePixelFormatARB` и `wglCreatePbufferARB`.

- Дополнительным атрибутом для функции `wglChoosePixelFormatARB` является `WGL_BIND_TO_TEXTURE_RGB_ARB` или `WGL_BIND_TO_TEXTURE_RGBA_ARB`. Посредством задания для одного из этих атрибутов ненулевого значения сообщается, что данный *p*-буфер должен поддерживать привязку к RGB- (или RGBA-) текстуре.
- Для функции `wglCreatePbufferARB` вводятся два дополнительных атрибута:
 - `WGL_TEXTURE_TARGET_ARB` — принимает одно из трех значений `WGL_TEXTURE_1D_ARB`, `WGL_TEXTURE_2D_ARB` или `WGL_TEXTURE_CUBE_MAP_ARB` (задание одно-, двух- или трехмерной текстуры);
 - `WGL_MIPMAP_TEXTURE_ARB` — ненулевое значение обеспечивает выделение памяти под все промежуточные уровни пирамидального фильтрования.

Для привязки *p*-буфера к текущей текстуре предназначена функция `wglBindTexImageARB`:

```
BOOL wglBindTexImageARB ( HPBUFFER hBuffer, int buf );
```

Параметр `buf` определяет, какой именно из буферов данного *p*-буфера следует связать с текущей текстурой. Он принимает одно из значений — `WGL_FRONT_LEFT_ARB`, `WGL_FRON_RIGHT_ARB`, `WGL_BACK_LEFT_ARB`, `WGL_BACK_RIGHT_ARB`, `WGL_AUX0_ARB`, `WGL_AUX1_ARB` и т. д.

После выполнения этой команды данный *p*-буфер привязывается к текущей текстуре и в него нельзя производить рендеринг.

"Отвязать" *p*-буфер от текстуры можно с помощью функции `wglReleaseTexImageARB`:

```
BOOL wglReleaseTexImageARB ( HPBUFFER hBuffer, int buf );
```

При привязывании *p*-буфера к кубической карте необходимо вызовом функции `wglSetPbufferAttribARB` установить грань куба, в которую идет вывод. Атрибут `WGL_CUBE_MAP_FACE_ARB` принимает одно из значений:

```
WGL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB, WGL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB,  
WGL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, WGL_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB,  
WGL_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, WGL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB.
```

Обратите внимание, что перед выбором очередной грани кубической карты следует вызвать функцию `glFlush ()`.

Реализация *p*-буфера для платформы Windows

Для удобства работы можно *p*-буфер "завернуть" в класс, инкапсулирующий все операции с ним, включая его создание и уничтожение. В листинге 9.1 приведено описание такого класса.

Листинг 9.1. Описание класса `PBuffer` для платформы Windows

```
#ifndef __P_BUFFER__
#define __P_BUFFER__

#include <windows.h>
#include <GL/gl.h>
#include "../glext.h"
#include "../wglext.h"

class PBuffer
{
protected:
    HDC           hdc;
    HGLRC         hGlrC;
    HPBUFFERARB  hBuffer;
    int           width;
    int           height;
    int           mode;
    unsigned      texture;
    bool          shareObjects;
    HDC           saveHdc;    // save values from
                           // from makeCurrent
    HGLRC         saveHglrc;

public:
    PBuffer ( int theWidth, int theHeight, int theMode = modeAlpha |
              modeDepth | modeStencil, bool theShareObjects = true );
    ~PBuffer ();

    int getWidth () const
    {
```

```
    return width;
}

int  getHeight () const
{
    return height;
}

bool  getShareObjects () const
{
    return shareObjects;
}

int  getMode () const
{
    return mode;
}

unsigned  getTexture () const
{
    return texture;
}

void  clear ();
bool  create ();

        // set attribute (like
        // WGL_TEXTURE_CUBE_MAP*_ARB)
bool  setAttr ( int attr, int value );

bool  makeCurrent ();      // direct rendering to
                          // this pbuffer

bool  restoreCurrent ();   // direct rendering to
                          // previous target

bool  isLost () const;     // whether p-buffer is lost
                          // due to mode switch

        // bind pbuffer to previously
```

```
// bound texture
bool bindToTexture ( int buf = WGL_FRONT_LEFT_ARB );

// unbind from texture
// (release)
bool unbindFromTexture ( int buf = WGL_FRONT_LEFT_ARB );

// set specific cubemap side,
// call glFlush after side
// is done
bool setCubemapSide ( int side );

void printLastError ();

bool checkExtensions (); // check support of required
// extensions

enum Mode
{
    modeAlpha      = 1,
    modeDepth      = 2,
    modeStencil    = 4,
    modeAccum      = 8,
    modeDouble     = 16,
    modeTexture1D  = 32,
    modeTexture2D  = 64,
    modeTextureCubeMap = 128,
    modeTextureMipmap = 256
};

};

#endif
```

Конструктор этого класса принимает размеры буфера, битовую маску атрибутов и признак того, следует ли копировать объекты из текущего контекста в контекст *p*-буфера.

Маска атрибутов строится сложением (выполнением побитовой операции ИЛИ) значений, приведенных в табл. 9.2.

Таблица 9.2. Флаги для класса PBuffer

Константа	Описание
modeAlpha	Требуется альфа-канал
modeDepth	Требуется буфер глубины
modeStencil	Требуется буфер трафарета
modeAccum	Требуется буфер накопления
modeDouble	Требуется двойной буфер
modeTexture1D	Привязка к одномерной текстуре
modeTexture2D	Привязка к двухмерной текстуре
modeTextureCubeMap	Привязка к кубической карте
modeTextureMipmap	Использовать пирамидальное фильтрование

Методы `create` и `clear` предназначены для создания *p*-буфера и его уничтожения. При этом для создания *p*-буфера используются параметры, переданные в конструктор.

Метод `makeCurrent` делает данный *p*-буфер текущей целью вывода. При этом текущая на момент вызова метода `makeCurrent` цель вывода запоминается, а восстанавливается с помощью метода `restoreCurrent`.

Метод `isLost` возвращает значение `true`, если буфер был разрушен. В этом случае для его восстановления нужно вызвать сначала метод `clear`, а затем метод `create`.

Методы `bindToTexture` и `unbindFromTexture` предназначены для привязывания и освобождения данного *p*-буфера от текущей текстуры.

Метод `setCubemapSide` задает грань кубической карты, в которую осуществляется рендеринг. Он автоматически вызывает функцию `glFlush`.

Метод `printLastError` записывает в стандартный файл сообщений, об ошибках (`stderr`) расшифровку последней возникшей ошибки, что может быть полезно при отладке приложения.

Метод `checkExtensions` проверяет поддержку необходимых для работы с *p*-буфером расширений.

Работа с *p*-буфером в Linux

Для работы с *p*-буфером на платформе Linux необходима поддержка расширений `GLX_SGIX_pbuffer` и `GLX_SGIX_fbconfig`.

Как и для Windows, сначала нужно получить текущее состояние (контекст), только в этом случае он несколько сложнее и состоит из трех частей — ука-

зателя на объект `Display`, целочисленного идентификатора экрана и самого контекста OpenGL:

```
Display * display = glXGetCurrentDisplay ();
GLXContext context = glXGetCurrentContext ();
int screen = DefaultScreen (display);
```

Далее следует при помощи функции `glXChooseFBConfig` получить список форматов, соответствующих требованию:

```
GLXFBCConfig * glxChooseFBConfig (Display * display, int screen,
                                   const int * attrList,
                                   int * numConfigs);
```

Параметры: `display` и `screen` — определяют, для чего будет выбираться подходящий формат; `attrList` — указатель списка атрибутов (массива пар (имя, значение), завершенного, как и для Windows, нулевым значением (табл. 9.3)); `numConfigs` — возвращает количество найденных форматов.

Таблица 9.3. Основные атрибуты для функции `glXChooseFBConfig`

Имя	Значение
GLX_DRAWABLE_TYPE_SGIX	Определяет, куда будет производиться вывод. Используется значение <code>GLX_PBUFFER_BIT_SGIX</code>
GLX_RENDER_TYPE_SGIX	Задает тип поверхности для вывода. Используется значение <code>GLX_RGBA_BIT_SGIX</code>
GLX_DOUBLE_BUFFER	Ненулевое значение обозначает использование двойной буферизации
GLX_RED_SIZE	Минимальное количество бит для значения в красном канале
GLX_GREEN_SIZE	Минимальное количество бит для значения в зеленом канале
GLX_BLUE_SIZE	Минимальное количество бит для значения в синем канале
GLX_ALPHA_SIZE	Минимальное количество бит для значения в альфа-канале
GLX_DEPTH_SIZE	Минимальное количество бит для значения в буфере глубины
GLX_STENCIL_SIZE	Минимальное количество бит для значения в буфере трафарета
GLX_ACCUM_RED_SIZE	Минимальное количество бит для значения в красном канале буфера накопления

Таблица 9.3 (окончание)

Имя	Значение
GLX_ACCUM_GREEN_SIZE	Минимальное количество бит для значения в зеленом канале буфера накопления
GLX_ACCUM_BLUE_SIZE	Минимальное количество бит для значения в синем канале буфера накопления
GLX_ACCUM_ALPHA_SIZE	Минимальное количество бит для значения в альфа-канале буфера накопления

После получения подходящего формата фреймбуфера можно непосредственно создать сам *p*-буфер при помощи функции `glXCreateGLXPbufferSGIX`:

```
GLXPbuffer glXCreateGLXPbufferSGIX (Display * display,
                                     GLXFBConfig config,
                                     unsigned int width,
                                     unsigned int height,
                                     int * attrList);
```

Параметры: `display` — объект, для которого создается *p*-буфер; `config` — пиксельный формат *p*-буфера; `width` и `height` — желаемый размер *p*-буфера; `attrList` — позволяет передать дополнительные параметры.

В отличие от Windows, созданный таким образом *p*-буфер по умолчанию не разрушается при переключении видеорежима.

Для получения реальных размеров *p*-буфера можно использовать следующий фрагмент кода:

```
glXQueryGLXPbufferSGIX (display, hBuffer, GLX_WIDTH_SGIX,
                         (unsigned *)&width );
glXQueryGLXPbufferSGIX (display, hBuffer, GLX_HEIGHT_SGIX,
                         (unsigned *)&height );
```

К сожалению, для Linux нет аналога расширения `WGL_render_texture`, поэтому необходимо явное копирование изображения из *p*-буфера в текстуру при помощи функции `glCopyTexSubImage2D`.

Как и в случае реализации для Windows, мы "завернем" *p*-буфер в класс с аналогичным открытым (`public`) интерфейсом, что позволяет легко использовать *p*-буфер в кроссплатформенных приложениях (листинг 9.2).

Листинг 9.2. Описание класса `PBuffer` для платформы Linux

```
#ifndef __P_BUFFER__
#define __P_BUFFER__
```

```
#include "libExt.h"
#include <GL/glxext.h>
#include <stdio.h>

class PBuffer
{
protected:
    Display * display;
    GLXContext hGLContext;
    GLXPbuffer hBuffer;
    GLXDrawable hPreviousDrawable;
    GLXContext hPreviousContext;
    int width;
    int height;
    int mode;
    bool shareObjects;
    unsigned texture;

public:
    PBuffer ( int theWidth, int theHeight, int theMode = modeAlpha |
              modeDepth | modeStencil, bool theShareObjects = true );
    ~PBuffer ();

    int getWidth () const
    {
        return width;
    }

    int getHeight () const
    {
        return height;
    }

    bool getShareObjects () const
    {
        return shareObjects;
    }

    int getMode () const
```

```
{  
    return mode;  
}  
  
unsigned getTexture () const  
{  
    return texture;  
}  
  
void clear ();  
bool create ();  
  
    // set attribute (like  
    // WGL_TEXTURE_CUBE_MAP*_ARB)  
bool setAttr ( int attr, int value );  
  
bool makeCurrent ();      // direct rendering to this  
                          // pbuffer  
  
bool restoreCurrent ();   // direct rendering to  
                          // prev. target  
  
bool isLost () const;    // whether p-buffer is lost due  
                          // to mode switch  
  
    // bind pbuffer to previously  
    // bound texture  
bool bindToTexture ();  
  
    // unbind from texture  
    // (release)  
bool unbindFromTexture ();  
  
void printLastError () {}  
  
bool checkExtensions ();  
  
enum Mode  
{
```

```
modeAlpha          = 1,
modeDepth         = 2,
modeStencil       = 4,
modeAccum          = 8,
modeDouble         = 16,
// modeTexture1D      = 32,
// modeTexture2D      = 64,
// modeTextureCubeMap = 128,
// modeTextureMipmap  = 256
};

};

#endif
```

Использование расширения EXT_framebuffer_object

Хотя *p*-буфера и поддерживаются многими графическими ускорителями, применение *p*-буферов имеет ряд неудобств.

- Работа с *p*-буфером сильно зависит от конкретной платформы (нам удалось сделать реализацию, почти не зависящую от платформы, но возможности этой реализации на разных plataформах отличаются).
- Каждый *p*-буфер имеет свой контекст рендеринга. Это приводит к тому, что операции переключения (на *p*-буфер, с одного *p*-буфера на другой *p*-буфер, с *p*-буфера на главный фреймбуфер) оказываются весьма дорогостоящими в плане быстродействия.
- Каждый *p*-буфер имеет свой набор внутренних буферов (цвета, глубины, трафарета и т. п.), но возможность совместного использования одного и того же внутреннего буфера несколькими *p*-буферами отсутствует.
- Схема выбора формата пикселов достаточно сложна и сильно зависит от конкретной платформы (хотя использование класса `PBuffer` и маскирует это).

Однако несмотря на все эти недостатки до последнего времени *p*-буфер оставался практически единственным приемлемым механизмом для рендеринга в текстуру (Render-To-Texture, RTT).

Ситуация сильно изменилась с выходом расширения `EXT_framebuffer_object` [15, 18], представляющего собой очень простую и платформенно-независимую альтернативу *p*-буферам.

Расширение `EXT_framebuffer_object` вводит возможности, вообще никак не зависящие ни от конкретной платформы, ни от используемой оконной системы.

Кроме того, для использования вводимых этим расширением фреймбуферов не требуется создание дополнительных контекстов рендеринга, что делает операцию переключения между различными фреймбуферами (включая и главный, предоставленный оконной системой) гораздо более дешевой (по времени), по сравнению с *p*-буферами.

Процедура выбора формата пикселов в этом расширении гораздо проще, чем для *p*-буфера.

Еще одним плюсом является то, что отдельные логические буферы фреймбуфера (например, буфер цвета или глубины) могут совместно использоваться сразу несколькими фреймбуферами.

Данное расширение вводит новый тип объекта в OpenGL — "фреймбуфер" (framebuffer object, FBO).

Объект "фреймбуфер" состоит из набора отдельных логических буферов (цвета, глубины, трафарета) и параметров состояния (рис. 9.3).

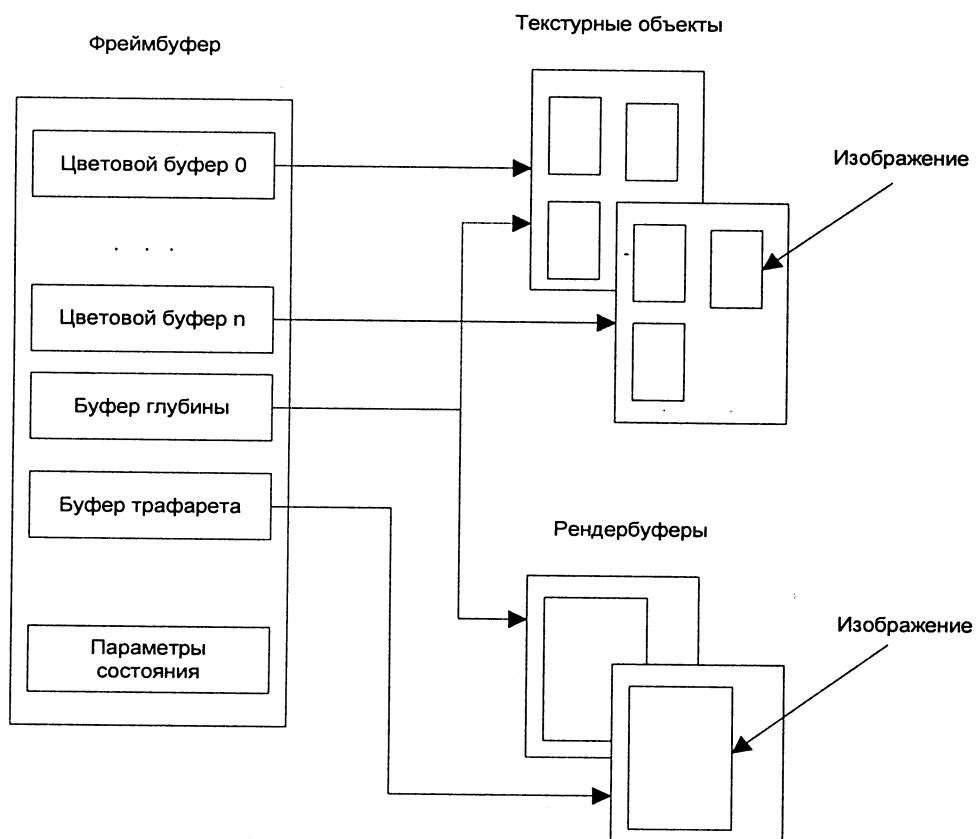


Рис. 9.3. Структура объекта "фреймбуфер"

В качестве логических буферов могут выступать как текстуры подходящих форматов, так и специальные объекты еще одного вводимого типа — "рендербуфер" (`renderbuffer object`, RBO).

Объект "рендербуфер" содержит внутри себя простое двухмерное изображение (без использования пирамидального фильтрования) и может использоваться для хранения результатов рендеринга в качестве одного из логических буферов.

Приложение может использовать много объектов-фреймбуферов (далее мы будем называть их просто фреймбуферами), но все они разделяются на начальный фреймбуфер, созданный оконной системой для приложения (он имеет идентификатор 0), и созданные самим приложением фреймбуферы (они идентифицируются беззнаковыми целыми числами, отличными от нуля).

Приложение может выбрать (`bind`) один из имеющихся фреймбуферов как текущий. Тогда соответствующие логические буферы данного фреймбуфера используются (как для чтения, так и для записи) при всех фрагментных операциях.

Каждый фреймбуфер идентифицируется беззнаковым целым числом (`GLuint`), причем нулевое значение зарезервировано за фреймбуфером, предоставленным оконной системой.

Выбор фреймбуфера как текущего осуществляется при помощи функции `glBindFramebufferEXT`:

```
void glBindFramebufferEXT (GLenum target, GLuint id);
```

В качестве параметра `target` всегда выступает константа `GL_FRAMEBUFFER_EXT`, а параметр `id` является идентификатором выбранного фреймбуфера.

Выделение блока идентификаторов фреймбуферов и их освобождение выполняются с помощью функций `glGenFramebuffersEXT` и `glDeleteFramebuffersEXT`:

```
void glGenFramebuffersEXT (GLsizei count, GLuint * ids);
void glDeleteFramebuffersEXT (GLsizei count, GLuint * ids);
```

Как и для ряда аналогичных функций, параметр `count` определяет, сколько идентификаторов следует выделить, а параметр `ids` указывает на массив, куда будут записаны выделенные идентификаторы.

Для проверки того, является ли данное беззнаковое целое число `id` идентификатором какого-либо фреймбуфера, предназначена функция `glIsFramebufferEXT`:

```
GLboolean glIsFramebufferEXT (GLuint id);
```

В качестве логических буферов для фреймбуфера могут выступать как текстуры, так и рендербуфера.

Каждый рендербуфер идентифицируется беззнаковым целым числом, отличным от нуля. Выделить и освободить блок идентификаторов рендербуферов можно при помощи функций `glGenRenderbuffersEXT` и `glDeleteRenderbuffersEXT`:

```
void glGenRenderbuffersEXT (GLsizei count, GLuint * ids);
void glDeleteRenderbuffersEXT (GLsizei count, GLuint * ids);
```

Для проверки того, является ли данное беззнаковое целое число `id` идентификатором какого-либо рендербуфера, предназначена функция `glIsRenderbufferEXT`:

```
GLboolean glIsRenderbufferEXT (GLuint id);
```

Перед использованием рендербуфера в качестве логического буфера следует задать для него размер и формат пикселов при помощи функции `glRenderbufferStorageEXT`:

```
void glRenderbufferStorageEXT ( GLenum target, GLenum internalFormat,
                               GLsizei width, GLsizei height );
```

Параметры: `target` — всегда принимает значение `GL_RENDERBUFFER_EXT`; `width` и `height` — задают размер рендербуфера в пикселях; `internalFormat` — задает внутренний формат пикселов буфера.

В качестве последнего параметра можно использовать как обычные форматы текстур (например, `GL_RGBA8` или `GL_DEPTH_COMPONENT24`), так и специальные форматы для буфера трафарета, приведенные в табл. 9.4.

Таблица 9.4. Допустимые типы внутренних форматов

Тип формата	Значение
<code>GL_STENCIL_INDEX1_EXT</code>	1-битовый буфер трафарета (1 бит на каждый пикセル)
<code>GL_STENCIL_INDEX4_EXT</code>	4-битовый буфер трафарета (4 бита на каждый пикセル)
<code>GL_STENCIL_INDEX8_EXT</code>	8-битовый буфер трафарета (8 бит на каждый пикセル)
<code>GL_STENCIL_INDEX16_EXT</code>	16-битовый буфер трафарета (16 бит на каждый пикセル)

Для подключения рендербуфера в качестве логического буфера к текущему фреймбуферу предназначена функция `glFramebufferRenderbufferEXT`:

```
void glFramebufferRenderbufferEXT (GLenum target, GLenum attachment,
                                   GLenum rbTarget,
                                   GLuint rbfId);
```

Параметр `target` принимает значение `GL_FRAMEBUFFER_EXT`, параметр `rbTarget` — значение `GL_RENDERBUFFER_EXT`. Параметр `attachment` задает логический буфер, в качестве которого следует подключить рендербуфер с идентификатором `rbfId` (может принимать значение `GL_COLOR_ATTACHMENT0_EXT`,

`GL_COLOR_ATTACHMENT1_EXT, ..., GL_DEPTH_ATTACHMENT_EXT` или
`GL_STENCIL_ATTACHMENT_EXT`).

Максимальное количество логических буферов цвета, которые можно подключить, можно узнать при помощи следующего фрагмента кода:

```
int maxColorAttachments;
glGetIntegerv ( GL_MAX_COLOR_ATTACHMENTS_EXT, &maxColorAttachments );
```

Переданное в качестве параметра `rbiId` нулевое значение отсоединяет ранее выбранный рендербуфер от текущего фреймбуфера.

Для подключения к фреймбуферу в качестве логического буфера текстуры предназначены функции `glFramebufferTexture1DEXT`, `glFramebufferTexture2DEXT` и `glFramebufferTexture3DEXT`:

```
void glFramebufferTexture1DEXT (GLenum target, GLenum attachment,
                               GLenum texTarget,
                               GLuint texId, int level);
void glFramebufferTexture2DEXT (GLenum target, GLenum attachment,
                               GLenum texTarget,
                               GLuint texId, int level);
void glFramebufferTexture3DEXT (GLenum target, GLenum attachment,
                               GLenum texTarget,
                               GLuint texId, int level, int zOffset );
```

В качестве подключаемых текстур могут выступать как одно- и двухмерные текстуры, так и отдельные стороны кубической текстурной карты и слои трехмерной текстуры.

Параметр `target` всегда принимает значение `GL_FRAMEBUFFER_EXT`, параметр `attachment` задает логический буфер, в качестве которого следует подсоединить данную текстуру, и принимает значение `GL_COLOR_ATTACHMENT0`, `GL_COLOR_ATTACHMENT1_EXT`, ..., `GL_DEPTH_ATTACHMENT_EXT` или `GL_STENCIL_ATTACHMENT_EXT`.

Параметр `texTarget`, определяющий тип используемой текстуры, может принимать значение как `GL_TEXTURE_nD`, так и `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, ..., `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

Параметр `level` задает уровень (в тирт-пирамиде) данной текстуры, который следует использовать в качестве логического буфера.

Для трехмерных текстур параметр `zOffset` определяет используемый срез трехмерной текстуры.

Примечание

Для успешного использования объекта "фреймбуфер" он должен удовлетворять определенным условиям. Например, все логические буферы должны

иметь один и тот же размер, форматы логических буферов должны соответствовать их типам подключения (использования). При подключении сразу нескольких буферов цвета необходимо, чтобы все они имели одинаковый внутренний формат.

Для проверки состояния фреймбуфер (т. е. можно ли осуществлять в него рендеринг) предназначена функция `glCheckFramebufferStatusEXT`:

```
GLenum glCheckFramebufferStatusEXT (GLenum target);
```

В качестве параметра `target` следует использовать константу `GL_FRAMEBUFFER_EXT`. Возможными возвращаемыми значениями являются:

```
GL_FRAMEBUFFER_COMPLETE_EXT,  
GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT_EXT,  
GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT_EXT,  
GL_FRAMEBUFFER_INCOMPLETE_DUPLICATE_ATTACHMENT_EXT,  
GL_FRAMEBUFFER_INCOMPLETE_DIMENSIONS_EXT,  
GL_FRAMEBUFFER_INCOMPLETE_FORMATS_EXT,  
GL_FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER_EXT,  
GL_FRAMEBUFFER_INCOMPLETE_READ_BUFFER_EXT,  
GL_FRAMEBUFFER_UNSUPPORTED_EXT и GL_FRAMEBUFFER_STATUS_ERROR_EXT.
```

В случае готовности фреймбуфера возвращается значение `GL_FRAMEBUFFER_COMPLETE_EXT`. Возвращенное значение `GL_FRAMEBUFFER_UNSUPPORTED_EXT` говорит о том, что следует попробовать другую комбинацию форматов буферов.

Также это расширение вводит функцию `glGenerateMipmapsEXT`, позволяющую построить все уровни в пирамидальном фильтровании для текстуры:

```
void glGenerateMipmapsEXT (GLenum target);
```

Параметр `target` принимает одно из значений `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP` или `GL_TEXTURE_3D`.

Удобно сразу "завернуть" фреймбуфер в класс, инкапсулировав внутри него все операции с ним. В листинге 9.3 приведен заголовочный файл такой реализации.

Листинг 9.3. Реализация фреймбуфера в виде класса

```
class FrameBuffer  
{  
    int      flags;  
    int      width;  
    int      height;  
    unsigned frameBuffer;  
    unsigned depthBuffer;
```

```
unsigned stencilBuffer;

public:
    FrameBuffer ( int theWidth, int theHeight, int theFlags = 0 );
    ~FrameBuffer () ;

    int getWidth () const
    {
        return width;
    }

    int getHeight () const
    {
        return height;
    }

    bool hasStencil () const
    {
        return stencilBuffer != 0;
    }

    bool hasDepth () const
    {
        return depthBuffer != 0;
    }

    bool isOk () const;
    bool create ();
    bool bind ();
    bool unbind ();

    bool attachColorTexture ( GLenum target, unsigned texId,
                             int no = 0 );
    bool attachDepthTexture ( GLenum target, unsigned texId );

    bool detachColorTexture ( GLenum target )
    {
        return attachColorTexture ( target, 0 );
    }
```

```

    bool detachDepthTexture ( GLenum target )
    {
        return attachDepthTexture ( target, 0 );
    }

    // create texture for attaching
    unsigned createColorTexture ( GLenum format = GL_RGB,
                                  GLenum internalFormat = GL_RGB8 ) ;

    enum
    {
        depth16      = 1,           // flags for depth and
        depth24      = 2,           // stencil buffers
        depth32      = 4,           //
        stencil1     = 16,          // 16-bit depth buffer
        stencil4     = 32,          // 24-bit depth buffer
        stencil8     = 64,          // 32-bit depth buffer
        stencil16    = 128,         // 1-bit stencil buffer
        stencil4     = 32,          // 4-bit stencil buffer
        stencil8     = 64,          // 8-bit stencil buffer
        stencil16    = 128,         // 16-bit stencil buffer
    };

    static bool isSupported () ;
    static int maxColorAttachments () ;
    static int maxSize () ;
};

```

Конструктор класса `FrameBuffer` в качестве входных параметров принимает размеры буфера в пикселях (`theWidth` и `theHeight`) и набор битовых флагов, задающих форматы пикселов для буферов глубины и трафарета (`theFlags`). При этом считается, что в качестве буфера цвета будет подключена текстура. Если в качестве буфера глубины также планируется использовать текстуру, то не следует задавать флаг для формата буфера глубины.

Обратите внимание: в конструкторе класса происходит только присваивание значений, никаких объектов OpenGL в нем не создается, для их создания служит метод `create`.

Метод `create` данного класса создает фреймбуфер и необходимые рендербуфера и возвращает значение `false` в случае ошибки. Если буфер создан успешно, то возвращается значение `true`.

Метод `bind` выбирает данный фреймбуфер как текущий.

Метод `unbind` выбирает в качестве текущего фреймбуфера фреймбуфер, предоставленный оконной системой (т. е. имеющий идентификатор 0).

Метод `attachColorTexture` подключает текстуру с заданным идентификатором в качестве цветового буфера с номером `no`. Параметр `target` принимает значение `GL_TEXTURE_2D` или значение, задающее определенную сторону кубической текстурной карты (`GL_TEXTURE_CUBE_MAP_POSITIVE_X`, ..., `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`).

Метод `attachDepthTexture` служит для подключения текстуры, заданной ее идентификатором, в качестве буфера глубины.

Методы `detachColorTexture` и `detachDepthTexture` предназначены для отсоединения текстур от соответствующих буферов.

Метод `createColorTexture` служит для создания текстуры, которая в дальнейшем может быть выбрана в качестве цветового буфера.

Метод `isOK` проверяет готовность фреймбуфера к использованию.

Метод `isSupported` проверяет, поддерживается ли расширение `EXT_framebuffer_object`.

Реализация данного класса приведена в листинге 9.4.

Листинг 9.4. Реализация класса `FrameBuffer`

```
#include "libExt.h"
#include "FrameBuffer.h"

FrameBuffer :: FrameBuffer ( int theWidth, int theHeight, int theFlags )
{
    width      = theWidth;
    height     = theHeight;
    flags      = theFlags;
    frameBuffer = 0;
    depthBuffer = 0;
    stencilBuffer = 0;
}

FrameBuffer :: ~FrameBuffer ()
{
    if ( depthBuffer != 0 )
        glDeleteRenderbuffersEXT ( 1, &depthBuffer );

    if ( stencilBuffer != 0 )
```

```
glDeleteRenderbuffersEXT ( 1, &stencilBuffer );\n\n    if ( frameBuffer != 0 )\n        glDeleteFramebuffersEXT ( 1, &frameBuffer );\n}\n\nbool FrameBuffer :: create ()\n{\n    if ( width <= 0 || height <= 0 )\n        return false;\n\n    glGenFramebuffersEXT ( 1, &frameBuffer );\n    glBindFramebufferEXT ( GL_FRAMEBUFFER_EXT, frameBuffer );\n\n    int depthFormat = 0;\n    int stencilFormat = 0;\n\n    if ( flags & depth16 )\n        depthFormat = GL_DEPTH_COMPONENT16_ARB;\n    else\n        if ( flags & depth24 )\n            depthFormat = GL_DEPTH_COMPONENT24_ARB;\n        else\n            if ( flags & depth32 )\n                depthFormat = GL_DEPTH_COMPONENT32_ARB;\n\n    if ( flags & stencil1 )\n        stencilFormat = GL_STENCIL_INDEX1_EXT;\n    else\n        if ( flags & stencil4 )\n            stencilFormat = GL_STENCIL_INDEX4_EXT;\n        else\n            if ( flags & stencil8 )\n                stencilFormat = GL_STENCIL_INDEX8_EXT;\n            else\n                if ( flags & stencil16 )\n                    stencilFormat = GL_STENCIL_INDEX16_EXT;\n\n    if ( depthFormat != 0 )
```

```
{  
    glGenRenderbuffersEXT      ( 1, &depthBuffer );  
    glBindRenderbufferEXT     ( GL_RENDERBUFFER_EXT,  
                                depthBuffer );  
    glRenderbufferStorageEXT   ( GL_RENDERBUFFER_EXT, depthFormat,  
                                width, height );  
    glFramebufferRenderbufferEXT ( GL_FRAMEBUFFER_EXT,  
                                 GL_DEPTH_ATTACHMENT_EXT,  
                                 GL_RENDERBUFFER_EXT,  
                                 depthBuffer );  
  
}  
  
if ( stencilFormat != 0 )  
{  
    glGenRenderbuffersEXT      ( 1, &stencilBuffer );  
    glBindRenderbufferEXT     ( GL_RENDERBUFFER_EXT,  
                                stencilBuffer );  
    glRenderbufferStorageEXT   ( GL_RENDERBUFFER_EXT,  
                                stencilFormat, width, height );  
    glFramebufferRenderbufferEXT ( GL_FRAMEBUFFER_EXT,  
                                 GL_STENCIL_ATTACHMENT_EXT,  
                                 GL_RENDERBUFFER_EXT,  
                                 stencilBuffer );  
  
}  
  
GLenum status = glCheckFramebufferStatusEXT ( GL_FRAMEBUFFER_EXT );  
  
glBindFramebufferEXT ( GL_FRAMEBUFFER_EXT, 0 );  
  
return status == GL_FRAMEBUFFER_COMPLETE_EXT;  
}  
  
bool FrameBuffer :: isOk () const  
{  
    unsigned currentFb;  
  
    glGetIntegerv ( GL_FRAMEBUFFER_BINDING_EXT, (int *)¤tFb );  
  
    if ( currentFb != frameBuffer )
```

```
glBindFramebufferEXT ( GL_FRAMEBUFFER_EXT, frameBuffer );  
  
bool complete = glCheckFramebufferStatusEXT ( GL_FRAMEBUFFER_EXT ) ==  
    GL_FRAMEBUFFER_COMPLETE_EXT;  
  
if ( currentFb != frameBuffer )  
    glBindFramebufferEXT ( GL_FRAMEBUFFER_EXT, currentFb );  
  
return complete;  
}  
  
bool FrameBuffer :: bind ()  
{  
    if ( frameBuffer == 0 )  
        return false;  
  
    glBindFramebufferEXT ( GL_FRAMEBUFFER_EXT, frameBuffer );  
  
    return true;  
}  
  
bool FrameBuffer :: unbind ()  
{  
    if ( frameBuffer == 0 )  
        return false;  
  
    glBindFramebufferEXT ( GL_FRAMEBUFFER_EXT, 0 );  
  
    return true;  
}  
  
bool FrameBuffer :: attachColorTexture ( GLenum target, unsigned texId,  
                                         int no )  
{  
    if ( frameBuffer == 0 )  
        return false;  
  
    if ( target != GL_TEXTURE_2D &&  
        (target < GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB ||
```

```
target > GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB) )
return false;

glFramebufferTexture2DEXT ( GL_FRAMEBUFFER_EXT,
                           GL_COLOR_ATTACHMENT0_EXT + no,
                           target, texId, 0 );

return true;
}

bool FrameBuffer :: attachDepthTexture ( GLenum target, unsigned texId )
{
    if ( frameBuffer == 0 )
        return false;

    glFramebufferTexture2DEXT ( GL_FRAMEBUFFER_EXT,
                               GL_DEPTH_ATTACHMENT_EXT,
                               target, texId, 0 );

    return true;
}

unsigned FrameBuffer :: createColorTexture ( GLenum format,
                                             GLenum internalFormat )
{
    unsigned tex;

    glGenTextures ( 1, &tex );
    glBindTexture ( GL_TEXTURE_2D, tex );
    glTexImage2D ( GL_TEXTURE_2D, 0, internalFormat, getWidth (),
                  getHeight (), 0, format, GL_UNSIGNED_BYTE, NULL );

    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_REPEAT );
    glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );

    return tex;
}
```

```
}

bool FrameBuffer :: isSupported ()
{
    return isExtensionSupported ( "EXT_framebuffer_object" );
}

int FrameBuffer :: maxColorAttachments ()
{
    int n;

    glGetIntegerv ( GL_MAX_COLOR_ATTACHMENTS_EXT, &n );

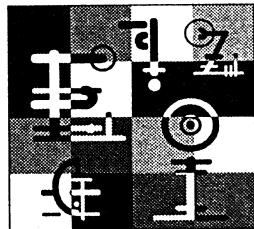
    return n;
}

int FrameBuffer :: maxSize ()
{
    int sz;

    glGetIntegerv ( GL_MAX_RENDERBUFFER_SIZE_EXT, &sz );

    return sz;
}
```

Примеры использования введенного класса `FrameBuffer` приведены в главах 15–17.



Глава 10

Понятие камеры. Работа с библиотекой libCamera

Одним из очень удобных и часто используемых понятий в визуализации трехмерных сцен является понятие камеры.

Фактически камера представляет собой систему координат (рис. 10.1), т. е. обладает своим положением (*pos*), направлением взгляда (*view*), направлениями вверх (*up*) и вправо (*right*).

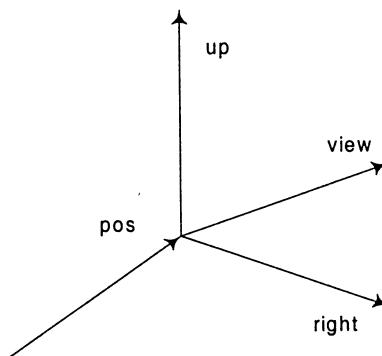


Рис. 10.1. Координатная система камеры

При этом направления *view*, *right* и *up* образуют так называемый *ортонормированный базис*, т. е. они попарно перпендикулярны и их длины равны единице.

Поскольку камера используется для рендеринга, это не просто координатная система. С ней также связана *усеченная пирамида видимости* (рис. 10.2). При этом вектор *view* задает направление этой пирамиды (*view frustum*) и ее основания, а векторы *up* и *right* — ее боковые стороны.

Пирамида видимости характеризуется двумя углами *fovX* и *fovY* (FOV-field of view), а также ближним и дальним расстояниями от сечения (*zNear* и *zFar*).

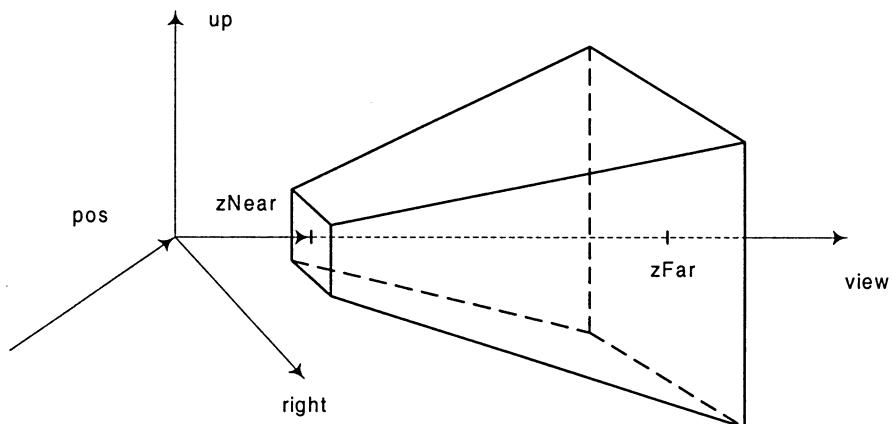


Рис. 10.2. Усеченная пирамида видимости для камеры

Таким образом, задание камеры — это фактически задание перспективного проектирования в OpenGL. Поэтому удобно добавить к свойствам камеры размеры прямоугольника (`width` и `height`), в который осуществляется рендеринг, и связать между собой значения `fovX` и `fovY` принятым в OpenGL соотношением:

$$\frac{\text{fov}X}{\text{fov}Y} = \frac{\text{width}}{\text{height}}.$$

Однако если в OpenGL обычно используются координатные системы только одной ориентации (правосторонние), то в общем случае может возникнуть необходимость использования камер с координатными системами различной ориентации.

Простейшим примером этого является построение зеркальных отражений — обычно для этого строят отраженную относительно плоскости зеркала камеру. Однако отражение меняет ориентацию координатной системы — из правосторонней она становится левосторонней и наоборот (рис. 10.3).

Поэтому удобно ввести в класс, инкапсулирующий камеру, возможность задавать и изменять ориентацию координатной системы, связанной с этой камерой.

Кроме того, задавать ориентацию камеры при помощи трех ортонормированных векторов на практике не очень удобно. Гораздо удобнее использовать в этих целях углы Эйлера (рис. 10.4) [2] или кватернионы [2, 3].

В листинге 10.1 приведено описание класса `Camera`, который мы будем использовать в дальнейшем.

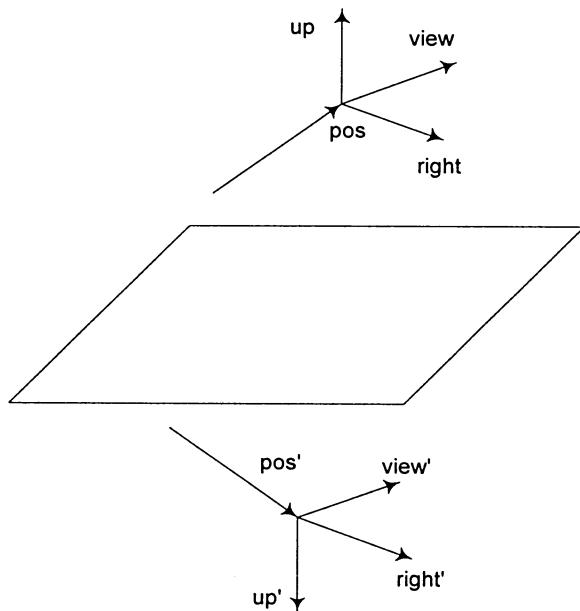


Рис. 10.3. Изменение ориентации координатной системы при отражении камеры

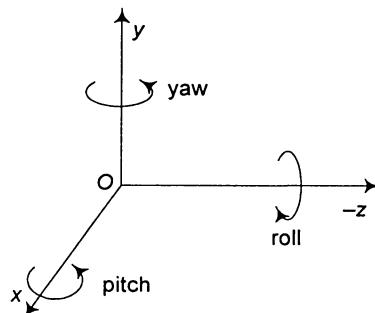


Рис. 10.4. Задание ориентации координатной системы при помощи углов Эйлера

Листинг 10.1. Описание класса Camera

```
//  
// Camera class for OpenGL  
//  
#ifndef __CAMERA__
```

```
#define __CAMERA__  
  
#include "Vector3D.h"  
#include "Matrix3D.h"  
  
class Plane;  
class Transform3D;  
class Quaternion;  
  
class Camera  
{  
    Vector3D pos;           // camera position  
    Vector3D viewDir;       // viewing direction  
                           // (normalized)  
    Vector3D upDir;         // up direction (normalized)  
    Vector3D sideDir;       // side direction (normalized)  
                           // may be left or right  
                           // depending on the handedness  
                           // of camera  
    bool rightHanded;       // whether camera is  
                           // righthanded  
    bool infinite;          // whether it uses zFar equal  
                           // infinity  
    Matrix3D transf;        // camera transform (from world  
                           // to camera space)  
    float   .fov;            // field of view angle  
    float   zNear;           // near clipping z-value  
    float   zFar;             // far clipping z-value  
    int     width;            // view width  
    int     height;           // view height  
    float   aspect;           // aspect ratio of camera  
  
public:  
    Camera ( const Vector3D& p, float yaw, float pitch, float roll,  
             float aFov = 60, float nearZ = 0.1, float farZ = 100,  
             bool rHanded = true );  
  
    Camera ( const Vector3D& p, const Quaternion& orientation,  
             float aFov = 60, float nearZ = 0.1, float farZ = 100,
```

```
        bool rHanded = true );
```

```
Camera ( const Camera& camera );
```

```
Camera ();
```

```
const Vector3D&    getPos () const
```

```
{
```

```
    return pos;
```

```
}
```

```
const Vector3D&    getViewDir () const
```

```
{
```

```
    return viewDir;
```

```
}
```

```
const Vector3D&    getSideDir () const
```

```
{
```

```
    return sideDir;
```

```
}
```

```
const Vector3D&    getUpDir () const
```

```
{
```

```
    return upDir;
```

```
}
```

```
bool    isRightHanded () const
```

```
{
```

```
    return rightHanded;
```

```
}
```

```
float   getZNear () const
```

```
{
```

```
    return zNear;
```

```
}
```

```
float   getZFar () const
```

```
{
```

```
    return zFar;
```

```
}
```

```
float    getFov () const
{
    return fov;
}

int     getWidth () const
{
    return width;
}

int     getHeight () const
{
    return height;
}

float   getAspect () const
{
    return aspect;
}

Quaternion getOrientation () const;

// map vector from world space to
// camera space
Vector3D mapFromWorld ( const Vector3D& p ) const
{
    Vector3D tmp ( p - pos );

    return Vector3D ( tmp & sideDir, tmp & upDir,
                      tmp & viewDir );
}

// map vector from camera space to
// world space
Vector3D mapToWorld ( const Vector3D& p ) const
{
    return pos + p.x * sideDir + p.y * upDir + p.z * viewDir;
}

// map vector to screen space
```

```
Vector3D mapToScreen ( const Vector3D& p ) const
{
    Vector3D scr ( transf * ( p - pos ) );

    return scr /= scr.z;      // do perspective transform
}

// move camera (absolute & relative)
void moveTo ( const Vector3D& newPos )
{
    pos = newPos;
    computeMatrix ();
}

void moveBy ( const Vector3D& delta )
{
    pos += delta;
    computeMatrix ();
}

// set orientation either via Euler angles or
// via quaternion
void setEulerAngles ( float theYaw, float thePitch,
                      float theRoll );
void setOrientation ( const Quaternion& orientation );

// set viewport parameters
void setViewSize     ( int theWidth, int theHeight,
                       float theFov );
void setFov          ( float newFovAngle );
void setRightHanded ( bool flag );

// transform camera
void mirror         ( const Plane& );
void transform       ( const Transform3D& );

void apply () const;      // sets camera transform
                         // for current context
```

```
private:  
    void computeMatrix (); // compute vectors,  
                          // transform matrix and  
                          // build viewing  
                          // frustum  
};  
  
#endif
```

Рассмотрим использование основных методов этого класса.

Конструкторы этого класса (за исключением пустого и сору-конструктора) получают на вход положение камеры в пространстве, угол обзора (fov), ближнее и дальнее расстояния отсечения, признак ориентации координатной системы и непосредственно саму ориентацию камеры в пространстве.

Ориентация камеры задается либо при помощи трех углов Эйлера (`yaw`, `pitch`, `roll`), либо при помощи кватерниона.

Методы `getPos`, `getViewDir` и `setUpDir` предназначены для доступа к текущему расположению камеры и ее направлениям вперед и вверх.

Поскольку направление бокового вектора (вправо или влево) зависит от ориентации координатной системы (право- или левосторонняя), то соответствующее направление называется *боковым* и возвращается при помощи метода `getSideDir`.

Узнать, является ли связанный с камерой система координат правосторонней или нет, можно при помощи метода `isRightHanded`.

Методы `getZNear` и `getZFar` позволяют получить ближнее и дальнее (вдоль вектора `viewDir`) расстояния отсечения.

Методы `getWidth` и `getHeight` позволяют получить размеры (в пикселях) связанной с камерой области вывода, а метод `getAspect` возвращает их отношение друг к другу (*aspect ratio*).

Текущую ориентацию камеры в виде кватерниона можно получить при помощи метода `getQuaternion`.

Методы `mapFromWorld` и `mapToWorld` служат для преобразования вектора из Мировой системы координат в систему координат камеры и из системы координат камеры в Мировую систему координат соответственно.

Метод `mapToScreen` позволяет получить экранные координаты (относительно верхнего левого угла области вывода, в пикселях), соответствующие трехмерному вектору.

Также в классе `Camera` есть ряд методов, позволяющих передвигать камеру, изменять ее ориентацию и другие параметры.

Методы `moveTo` и `moveBy` служат для изменения положения камеры (без изменения ее ориентации). Первый перемещает камеру в заданную точку в пространстве, в то время как второй просто передвигает ее в соответствии с заданным вектором.

Изменить ориентацию камеры (без изменения ее положения) можно при помощи методов `setEulerAngles` и `setOrientation`. Первый позволяет задать новую ориентацию камеры при помощи трех углов Эйлера, второй делает это с помощью кватернионов.

Метод `setRightHanded` позволяет задать ориентацию координатной системы, связанной с камерой, как правостороннюю или левостороннюю.

При помощи метода `setFov` можно изменить угол обзора камеры (угол задается в градусах).

Метод `setViewSize` предназначен для изменения размеров области, служащей для рендеринга при помощи данной камеры.

Можно подвергнуть камеру аффинным преобразованиям. Метод `mirror` служит для отражения камеры относительно заданной плоскости, а метод `transform` позволяет применить произвольное аффинное преобразование к камере.

При помощи метода `apply` камера устанавливает параметры проектирования для OpenGL.

В листинге 10.2 приведена полная реализация класса `Camera`.

Листинг 10.2. Реализация класса `Camera`

```
//  
// Camera class for OpenGL  
  
#ifndef _WIN32  
    #include <windows.h>  
#endif  
  
#include <GL/gl.h>  
#include <GL/glu.h>  
#include "Camera.h"  
#include "Quaternion.h"  
#include "Transform3D.h"  
#include "Plane.h"  
  
Camera :: Camera ( const Vector3D& p, float yaw, float pitch, float roll,
```

```

        float aFov, float nearZ, float farZ, bool rHanded )
{
    fov      = aFov;
    zNear   = nearZ;
    zFar    = farZ;
    pos     = p;
    width   = 640;           // default view size
    height  = 480;
    aspect   = (float)width / (float) height;
    rightHanded = rHanded;
    infinite = false;

    setEulerAngles ( yaw, pitch, roll );
}

Camera :: Camera ( const Vector3D& p, const Quaternion& orientation,
                   float aFov, float nearZ, float farZ, bool rHanded )
{
    fov      = aFov;
    zNear   = nearZ;
    zFar    = farZ;
    pos     = p;
    width   = 640;           // default view size
    height  = 480;
    aspect   = (float)width / (float) height;
    rightHanded = rHanded;
    infinite = false;

    setOrientation ( orientation );
}

Camera :: Camera ( const Camera& camera )
{
    zNear      = camera.zNear;
    zFar       = camera.zFar;
    fov        = camera.fov;
    pos        = camera.pos;
    viewDir   = camera.viewDir;
    upDir     = camera.upDir;
}

```

```
rightHanded = camera.rightHanded;
infinite    = camera.infinite;
width       = camera.width;
height      = camera.height;
aspect       = camera.aspect;

computeMatrix ();
}

void     Camera :: setEulerAngles ( float yaw, float pitch, float roll )
{
    viewDir = Vector3D ( 1, 0, 0 );
    sideDir = Vector3D ( 0, 1, 0 );
    upDir   = Vector3D ( 0, 0, 1 );

    Matrix3D  rot ( Matrix3D :: getRotateYawPitchRollMatrix ( yaw,
                                                               pitch, roll ) );

    viewDir = rot * viewDir;
    upDir   = rot * upDir;
    sideDir = rot * sideDir;

    computeMatrix ();
}

void     Camera :: setOrientation ( const Quaternion& orientation )
{
    viewDir = Vector3D ( 1, 0, 0 );
    sideDir = Vector3D ( 0, 1, 0 );
    upDir   = Vector3D ( 0, 0, 1 );

    Matrix3D  rot = orientation.getMatrix ();

    viewDir = rot * viewDir;
    upDir   = rot * upDir;
    sideDir = rot * sideDir;

    computeMatrix ();
}
```

```
Quaternion Camera :: getOrientation () const
{
    // build matrix from viewDir, sideDir, upDir
    Matrix3D rot ( viewDir, sideDir, upDir );

    // build quaternion from it
    return Quaternion ( rot );
}

void Camera :: setRightHanded ( bool flag )
{
    if ( flag == rightHanded )
        return;

    rightHanded = flag;

    computeMatrix ();
}

void Camera :: setFov ( float newFovAngle )
{
    fov = newFovAngle;

    computeMatrix ();
}

void Camera :: setViewSize ( int theWidth, int theHeight,
                           float theFov )
{
    width = theWidth;
    height = theHeight;
    fov = theFov;
    aspect = (float)width / (float)height;

    computeMatrix ();
}

void Camera :: computeMatrix ()
```

```
{  
    viewDir.normalize (); // normalize viewDir  
    // compute upDir perpendicular to viewDir  
    upDir -= (upDir & viewDir) * viewDir;  
  
    // compute sideDir to form orthogonal basis  
    sideDir = upDir ^ viewDir;  
  
    if ( !rightHanded ) // invert side orientation: right -> left  
        sideDir = -sideDir;  
  
    upDir .normalize (); // orthonormalize upDir and sideDir  
    sideDir.normalize ();  
  
    // now build transform matrix  
    float halfAngle = 0.5 * M_PI * fov / 180.0;  
    float fovValueY = 1 / (float)tan ( halfAngle );  
    float fovValueX = fovValueY / aspect;  
  
    transf [0][0] = (sideDir [0] * fovValueX +  
                     viewDir [0]) * 0.5 * width;  
    transf [0][1] = (sideDir [1] * fovValueX +  
                     viewDir [1]) * 0.5 * width;  
    transf [0][2] = (sideDir [2] * fovValueX +  
                     viewDir [2]) * 0.5 * width;  
    transf [1][0] = (upDir [0] * fovValueY +  
                     viewDir [0]) * 0.5 * height;  
    transf [1][1] = (upDir [1] * fovValueY +  
                     viewDir [1]) * 0.5 * height;  
    transf [1][2] = (upDir [2] * fovValueY +  
                     viewDir [2]) * 0.5 * height;  
    transf [2][0] = viewDir [0];  
    transf [2][1] = viewDir [1];  
    transf [2][2] = viewDir [2];  
}  
  
void Camera :: mirror ( const Plane& mirror )  
{  
    mirror.reflectPos ( pos );
```

```
mirror.reflectDir ( viewDir );
mirror.reflectDir ( upDir );
mirror.reflectDir ( sideDir );

rightHanded = !rightHanded;

computeMatrix ();
}

void Camera :: transform ( const Transform3D& tr )
{
    pos      = tr.transformPoint ( pos      );
    viewDir = tr.transformDir   ( viewDir );
    upDir   = tr.transformDir   ( upDir   );
    sideDir = tr.transformDir   ( sideDir );

    // now check for right/left handedness
    rightHanded = ((upDir ^ viewDir) & sideDir) > EPS;

    computeMatrix ();
}

void Camera :: apply () const
{
    float    m [16];
    // create a coordinate space transform
    // (rotation) matrix
    m [0 ] = sideDir.x;
    m [1 ] = upDir.x;
    m [2 ] = -viewDir.x;
    m [3 ] = 0;

    m [4 ] = sideDir.y;
    m [5 ] = upDir.y;
    m [6 ] = -viewDir.y;
    m [7 ] = 0;

    m [8 ] = sideDir.z;
    m [9 ] = upDir.z;
```

```
m [10] = -viewDir.z;
m [11] = 0;

m [12] = 0;
m [13] = 0;
m [14] = 0;
m [15] = 1;

// set current viewport
glViewport ( 0, 0, getWidth (), getHeight () );

glMatrixMode ( GL_PROJECTION );
glLoadIdentity ();

// calculate aspect ratio of the window
gluPerspective ( getFov (), getAspect (), getZNear (),
                  getZFar () );

glMatrixMode ( GL_MODELVIEW );
glLoadIdentity ();

glMultMatrixf ( m );
glTranslatef ( -pos.x, -pos.y, -pos.z );
}
```

Как видно из листинга, основой данного класса является метод `computeMatrix`, корректирующий основные направления и вычисляющий матрицу преобразования, переводящую векторы из Мировой системы координат в систему координат камеры.

Данный метод проводит ортонормализацию всех направлений и их коррекцию с учетом ориентации координатной системы камеры.

В листинге 10.3 приведен исходный текст простого примера, демонстрирующего использование класса `Camera`.

Листинг 10.3. Пример использования класса `Camera`

```
// 
// Simple camera class test
//
```

```
#include    "libExt.h"

#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>

#include    "libTexture.h"
#include    "TypeDefs.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "boxes.h"
#include    "Camera.h"

Vector3D   eye    ( -0.5, -0.5, 1.5 ); // camera position
unsigned    decalMap;                  // decal (diffuse) texture
unsigned    stoneMap;
unsigned    teapotMap;
float       angle     = 0;

float      yaw      = 0;
float      pitch   = 0;
float      roll    = 0;

Camera     camera ( eye, 0, 0, 0 ); // camera to be used

void init ()
{
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );
    glEnable     ( GL_DEPTH_TEST );
    glEnable     ( GL_TEXTURE_2D );
    glDepthFunc ( GL_LEQUAL );

    glHint ( GL_POLYGON_SMOOTH_HINT,           GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}

void displayBoxes ()
{
    glMatrixMode ( GL_MODELVIEW );
```

```
glPushMatrix ();

drawBox ( Vector3D ( -5, -5, 0 ), Vector3D ( 10, 10, 3 ),
          stoneMap, false );
drawBox ( Vector3D ( 3, 2, 0.5 ), Vector3D ( 1, 2, 2 ),
          decalMap );

glBindTexture ( GL_TEXTURE_2D, teapotMap );

glPushMatrix ();
glTranslatef ( 0.2, 1, 1.5 );

glRotatef ( angle * 45.3, 1, 0, 0 );
glRotatef ( angle * 57.2, 0, 1, 0 );

glutSolidTeapot ( 0.3 );

glPopMatrix ();
glPopMatrix ();
}

void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    camera.apply ();

    displayBoxes ();

    glutSwapBuffers ();
}

void reshape ( int w, int h )
{
    camera.setViewSize ( w, h, 60 );
    camera.apply      ();
}

void key ( unsigned char key, int x, int y )
```

```
{  
    if ( key == 27 || key == 'q' || key == 'Q' ) // quit requested  
        exit ( 0 );  
    else  
        if ( key == 'w' || key == 'W' )  
            camera.moveBy ( camera.getViewDir () * 0.2 );  
        else  
            if ( key == 'x' || key == 'X' )  
                camera.moveBy ( -camera.getViewDir () * 0.2 );  
            else  
                if ( key == 'a' || key == 'A' )  
                    camera.moveBy ( -camera.getSideDir () * 0.2 );  
                else  
                    if ( key == 'd' || key == 'D' )  
                        camera.moveBy ( camera.getSideDir () * 0.2 );  
  
    glutPostRedisplay ();  
}  
  
void     specialKey ( int key, int x, int y )  
{  
    if ( key == GLUT_KEY_UP )  
        yaw += M_PI / 90;  
    else  
        if ( key == GLUT_KEY_DOWN )  
            yaw -= M_PI / 90;  
    else  
        if ( key == GLUT_KEY_RIGHT )  
            roll += M_PI / 90;  
    else  
        if ( key == GLUT_KEY_LEFT )  
            roll -= M_PI / 90;  
  
    camera.setEulerAngles ( yaw, pitch, roll );  
  
    glutPostRedisplay ();  
}  
  
void    mouseFunc ( int x, int y )
```

```
{  
    static int lastX = -1;  
    static int lastY = -1;  
  
    if ( lastX == -1 )           // not initialized  
    {  
        lastX = x;  
        lastY = y;  
    }  
  
    yaw -= (y - lastY) * 0.02;  
    roll += (x - lastX) * 0.02;  
  
    lastX = x;  
    lastY = y;  
  
    camera.setEulerAngles ( yaw, pitch, roll );  
  
    glutPostRedisplay ();  
}  
  
void animate ()  
{  
    static float lastTime = 0.0;  
    float time      = 0.001f * glutGet ( GLUT_ELAPSED_TIME );  
  
    angle += 2 * (time - lastTime);  
  
    lastTime = time;  
  
    glutPostRedisplay ();  
}  
  
int main ( int argc, char * argv [] )  
{  
    // initialize glut  
    glutInit          ( &argc, argv );  
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );  
    glutInitWindowSize ( 512, 512 );
```

```

        // create window
glutCreateWindow ( "OpenGL camera test" );

        // register handlers
glutDisplayFunc      ( display      );
glutReshapeFunc      ( reshape      );
glutKeyboardFunc     ( key         );
glutSpecialFunc      ( specialKey );
glutPassiveMotionFunc ( mouseFunc );
glutIdleFunc         ( animate      );

init              ();
initExtensions ();

decalMap  = createTexture2D ( true, "..../Textures/oak.bmp" );
stoneMap   = createTexture2D ( true, "..../Textures/block.bmp" );
teapotMap = createTexture2D ( true, "..../Textures/Oxidated.jpg" );

camera.setRightHanded ( false );

glutMainLoop ();

return 0;
}

```

Еще одним простым классом, часто используемым вместе с камерой, является класс `Frustum`, содержащий информацию о плоскостях отсечения, ограничивающих усеченную пирамиду видимости.

Описание и реализация класса `Frustum` приведены в листингах 10.4 и 10.5.

Листинг 10.4. Описание класса `Frustum`

```

// 
// Frustum class for OpenGL
//

#ifndef    __FRUSTUM__
#define    __FRUSTUM__


#include  "Vector3D.h"

```

```
#include "Plane.h"

class BoundingBox;

class Frustum
{
    Plane plane [6];

public:
    Frustum ();

    void get ();           // acquire from current
                          // OpenGL context

                          // checks
    bool pointInFrustum ( const Vector3D& v ) const;
    bool boxInFrustum ( const BoundingBox& box ) const;
};

#endif
```

Листинг 10.5. Реализация класса Frustum

```
//
// Frustum class for OpenGL
//

#ifndef _WIN32
#include <windows.h>
#endif

#include <GL/gl.h>
#include "Frustum.h"
#include "Math3D.h"
#include "BoundingBox.h"

Frustum :: Frustum ()
{
```

```
    get ();
}

void Frustum :: get ()
{
    float proj [16]; // projection matrix
    float model [16]; // modelview matrix
    float clip [16]; // transform to clip space matrix

    glGetFloatv ( GL_PROJECTION_MATRIX, proj );
    glGetFloatv ( GL_MODELVIEW_MATRIX, model );

    // now multiply them to get clip
    // transform matrix

    clip[ 0] = model[ 0] * proj[ 0] + model[ 1] * proj[ 4] +
               model[ 2] * proj[ 8] + model[ 3] * proj[12];
    clip[ 1] = model[ 0] * proj[ 1] + model[ 1] * proj[ 5] +
               model[ 2] * proj[ 9] + model[ 3] * proj[13];
    clip[ 2] = model[ 0] * proj[ 2] + model[ 1] * proj[ 6] +
               model[ 2] * proj[10] + model[ 3] * proj[14];
    clip[ 3] = model[ 0] * proj[ 3] + model[ 1] * proj[ 7] +
               model[ 2] * proj[11] + model[ 3] * proj[15];
    clip[ 4] = model[ 4] * proj[ 0] + model[ 5] * proj[ 4] +
               model[ 6] * proj[ 8] + model[ 7] * proj[12];
    clip[ 5] = model[ 4] * proj[ 1] + model[ 5] * proj[ 5] +
               model[ 6] * proj[ 9] + model[ 7] * proj[13];
    clip[ 6] = model[ 4] * proj[ 2] + model[ 5] * proj[ 6] +
               model[ 6] * proj[10] + model[ 7] * proj[14];
    clip[ 7] = model[ 4] * proj[ 3] + model[ 5] * proj[ 7] +
               model[ 6] * proj[11] + model[ 7] * proj[15];
    clip[ 8] = model[ 8] * proj[ 0] + model[ 9] * proj[ 4] +
               model[10] * proj[ 8] + model[11] * proj[12];
    clip[ 9] = model[ 8] * proj[ 1] + model[ 9] * proj[ 5] +
               model[10] * proj[ 9] + model[11] * proj[13];
    clip[10] = model[ 8] * proj[ 2] + model[ 9] * proj[ 6] +
               model[10] * proj[10] + model[11] * proj[14];
    clip[11] = model[ 8] * proj[ 3] + model[ 9] * proj[ 7] +
               model[10] * proj[11] + model[11] * proj[15];
    clip[12] = model[12] * proj[ 0] + model[13] * proj[ 4] +
```

```
        model[14] * proj[ 8] + model[15] * proj[12];
clip[13] = model[12] * proj[ 1] + model[13] * proj[ 5] +
           model[14] * proj[ 9] + model[15] * proj[13];
clip[14] = model[12] * proj[ 2] + model[13] * proj[ 6] +
           model[14] * proj[10] + model[15] * proj[14];
clip[15] = model[12] * proj[ 3] + model[13] * proj[ 7] +
           model[14] * proj[11] + model[15] * proj[15];

// now extract clip planes params:
Vector3D n [6];
float    d [6];
    // right plane
n [0].x = clip [3] - clip [0];
n [0].y = clip [7] - clip [4];
n [0].z = clip [11] - clip [8];
d [0]   = clip [15] - clip [12];

    // left plane
n [1].x = clip [3] + clip [0];
n [1].y = clip [7] + clip [4];
n [1].z = clip [11] + clip [8];
d [1]   = clip [15] + clip [12];

    // top plane
n [2].x = clip [3] - clip [1];
n [2].y = clip [7] - clip [5];
n [2].z = clip [11] - clip [9];
d [2]   = clip [15] - clip [13];

    // bottom plane
n [3].x = clip [3] + clip [1];
n [3].y = clip [7] + clip [5];
n [3].z = clip [11] + clip [9];
d [3]   = clip [15] + clip [13];

    // far plane
n [4].x = clip [3] - clip [2];
n [4].y = clip [7] - clip [6];
n [4].z = clip [11] - clip [10];
```

```
d [4] = clip [15] - clip [14];

        // near plane
n [5].x = clip [3] + clip [2];
n [5].y = clip [7] + clip [6];
n [5].z = clip [11] + clip [10];
d [5] = clip [15] + clip [14];

        // normalize
for ( int i = 0; i < 6; i++ )
{
    float len = n [i].length ();

    if ( len > EPS )
    {
        n [i] /= len;
        d [i] /= len;
    }

    plane [i] = Plane ( n [i], d [i] );
}

bool Frustum :: pointInFrustum ( const Vector3D& v ) const
{
    for ( register int i = 0; i < 6; i++ )
        if ( plane [i].classify ( v ) == IN_BACK )
            return false;

    return true;
}

bool Frustum :: boxInFrustum ( const BoundingBox& box ) const
{
    for ( register int i = 0; i < 6; i++ )
        if ( box.classify ( plane [i] ) == IN_BACK )
            return false;

    return true;
}
```

Как видно из приведенной в листинге 10.5 реализации, данный класс строит произведение текущей модельной и проектирующей матриц и из этого произведения извлекает коэффициенты уравнений ограничивающих плоскостей. Поскольку произведение этих матриц переводит всю усеченную пирамиду видимости в куб $[-1, 1]^3$, ограничивающие плоскости пирамиды видимости преобразуются в плоскости, ограничивающие этот куб (т. е. $x = 1, x = -1, y = 1, y = -1, z = 1, z = -1$) (рис. 10.5).

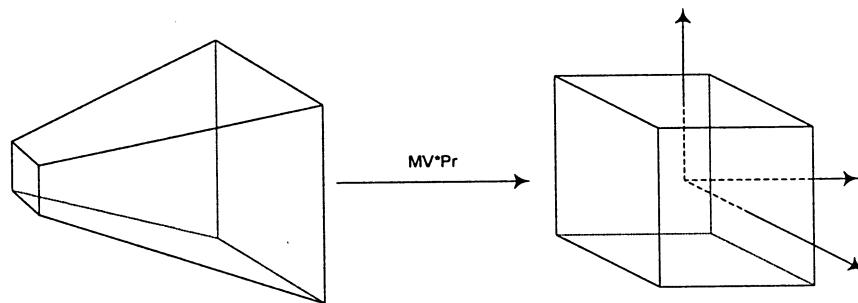
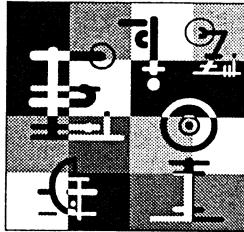


Рис. 10.5. Преобразование области видимости произведением модельной и проектирующей матриц

Наиболее важным применением класса `Frustum` является проверка на отсечение объектов — для сложных объектов и сцен в ряде случаев оказывается гораздо более выгодным сразу отбросить части, заведомо не попадающие в область видимости. Это позволяет избежать перегрузки графического процессора ненужными данными.



Глава 11

Библиотека libMesh. Работа с полигональными моделями и их загрузка из популярных форматов

На данный момент существует довольно большое количество различных форматов хранения трехмерных объектов — практически каждый моделлер (равно как и серьезная игра) вводят свои форматы (зачастую один продукт вводит сразу несколько форматов).

Для дальнейшей работы было бы удобно, с одной стороны, выбрать определенный формат для внутреннего представления трехмерных объектов, а с другой — научиться переводить модели из основных форматов в это представление.

Далее рассматривается написанная автором библиотека libMesh, которая хотя и не является идеальной, но вполне подходит для целого ряда приложений.

Рассмотрим данные, необходимые для описания трехмерного объекта, и то, как лучше их организовать для достижения достаточной гибкости и высокого быстродействия.

В общем случае можно считать, что каждый объект состоит из набора вершин (каждая из которых обладает комплектом атрибутов, таких как положение, вектор нормали и т. п.), объединенных в грани и ребра.

Удобно сразу же ограничиться случаем треугольных граней — любая выпуклая грань легко раскладывается в "веер" треугольников (рис. 11.1), а невыпуклые грани встречаются крайне редко (к тому же OpenGL работает только с выпуклыми гранями).

Одновременная поддержка информации и о гранях и о ребрах оказывается зачастую весьма желательной (например, при нахождении отбрасываемой таким объектом тени методом теневых объемов — *stencil shadow volumes* [18]).

Каждая грань модели естественным образом определяется тремя индексами в массив вершин, кроме этого бывает удобно связать с каждой гранью вектор нормали к ней.

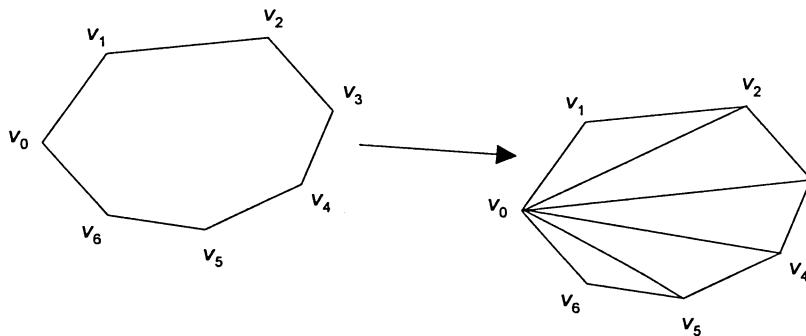


Рис. 11.1. Разложение выпуклой грани на "веер" треугольников

Тем самым, мы приходим к следующему представлению грани в нашем объекте. Описание грани:

```
struct Face // triangular face
{
    int index [3]; // indices to Vertex array
    Vector3D n; // face normal
};
```

Массив `index` состоит из трех индексов в массив вершин, а поле `n` содержит единичный вектор нормали к данной грани (как к треугольнику в пространстве) (рис. 11.2).

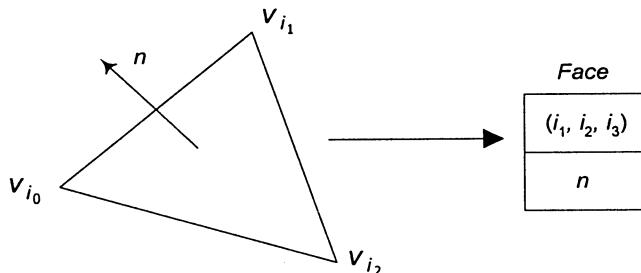


Рис. 11.2. Задание грани при помощи структуры Face

С ребром ситуация несколько сложнее — большинство ребер принадлежат сразу двум граням объекта. При этом ребро, принадлежащее двум граням, для каждой из них проходится в своем направлении (рис. 11.3).

Таким образом, с произвольным ребром связывается следующая информация:

- индексы в массив вершин для начальной и конечной вершины ребра;
- индексы двух граней, которым принадлежит данная грань;
- направление обхода для каждой из этих двух граней.

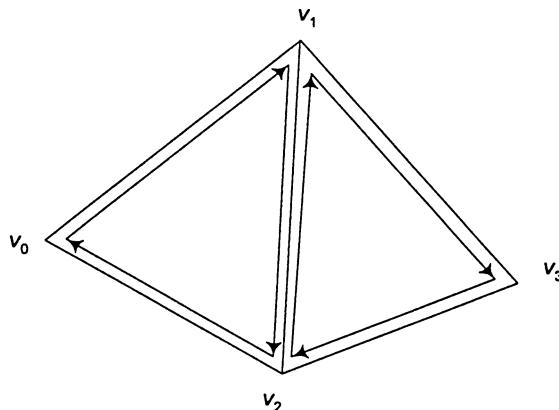


Рис. 11.3. Общее для двух граней ребро имеет противоположные направления обхода для каждой из этих граней.

Удобно объединить последние две группы — старший бит в индексе грани будет задавать направление, в котором ребро должно быть пройдено для этой грани (от конечной вершины к начальной или наоборот). А младшие 31 бит будут определять собственно сам номер грани.

Описание ребра:

```
struct Edge
{
    int      a, b;          // indices into array of Vertex
    int      f1, f2;        // indices into array of Face's
                           // hi-order bit means for this face reverse
                           // endpoints
};
```

Поля *a* и *b* задают индексы в массив вершин для начальной и конечной точек данного ребра. Поля *f1* и *f2* несут двойную информацию — младшие 31 бит содержат индексы в массив граней, а старший бит задает направление прохождения ребра для данной грани (если старший бит равен единице, то грань проходится в направлении от *b* к *a*, иначе — от *a* к *b*). Значение *-1* в поле *f2* означает, что данное ребро принадлежит только одной грани (задаваемой полем *f1*).

Теперь рассмотрим атрибуты, которые нужно задать для каждой из вершин объекта. Очевидно, что в их число входят собственно положение вершины в пространстве, базис касательного пространства для данной грани (вектора *t*, *b* и *n*, см. главу 13 и [1, 18]), текстурные координаты и цвет.

Для большинства случаев подобной информации оказывается достаточно — даже в тех случаях, когда используется сразу несколько текстур для одного

и того же набора граней, то текстурные координаты для них обычно или совпадают или же отличаются только аффинным преобразованием (поэтому можно хранить только один набор текстурных координат и для каждой текстуры хранить соответствующее преобразование текстурных координат).

Тем самым, сформулированного ранее списка атрибутов оказывается вполне достаточно и мы приходим к представлению вершины.

Описание вершины:

```
struct Vertex
{
    Vector4D pos;           // position of vertex
    Vector3D n;             // unit normal
    Vector3D t, b;          // tangent and binormal
    Vector2D tex;            // texture coordinates
    Vector4D color;          // vertex color
};
```

Поскольку, как уже отмечалось, с моделью может быть связано несколько текстур (для каждой из которых используется свой закон преобразования текстурных координат), удобно поместить в модель информацию о некотором наборе стандартных (т. е. наиболее часто используемых) текстур.

Каждая такая текстура характеризуется именем файла текстуры (чтобы потом можно было одной командой загрузить все необходимые текстуры, не влезая внутрь объекта), ее идентификатором в OpenGL и номером текстурного блока, в который она должна быть помещена.

Кроме того, было бы удобно внести туда информацию о простейшем законе преобразования текстурных координат для данной текстуры. Поскольку в большинстве случаев данное преобразование является комбинацией переноса и масштабирования (т. е. имеет вид *off* + *scale***tex*), то удобно и эти параметры внести в соответствующий класс.

В результате мы приходим к следующему классу, задающему используемую объектом текстуру (листинги 11.1 и 11.2).

Листинг 11.1. Описание класса *Map*

```
class Map           // map info
{
public:
    string mapName; // file with texture
    int unit;        // texture unit to place this
                      // texture into
```

```
unsigned    id;           // texture-id
GLenum     target;        // texture target
bool       mipmap;        // whether to use mipmapping
Vector2D   texOffs;
Vector2D   texScale;

Map   ();
~Map  ();

bool   load             ();
bool   setTextureMatrix ();
bool   bind             ();

void   bindToUnit ( int theUnit )
{
    unit = theUnit;
}

};
```

Метод `load` данного класса загружает соответствующую текстуру из файла (если его имя не пусто).

Метод `setTextureMatrix` определяет матрицу для преобразования текстурных координат (`GL_TEXTURE`).

Метод `bindToUnit` устанавливает текстурный блок, в который следует поместить данную текстуру (по умолчанию текстура помещается в нулевой текстурный блок).

Метод `bind` осуществляет установку текстуры (если она есть) в выбранный текстурный блок и настраивает матрицу преобразования текстурных координат при помощи метода `setTextureMatrix`.

Деструктор данного класса освобождает связанную с ним текстуру.

Листинг 11.2. Реализация класса Map

```
Map :: Map ()
{
    unit      = 0;                      // assign to unit 0
    id        = 0;                      // illegal id
    target    = GL_TEXTURE_2D;          // 2D texture by default
```

```
mipmap    = true;
texOffs   = Vector2D ( 0, 0 ); // empty texture transform
texScale  = Vector2D ( 1, 1 );
}

Map :: ~Map ()
{
    if ( id != 0 )
        glDeleteTextures ( 1, &id );
}

bool Map :: load ()
{
    id = 0;

    if ( mapName.empty () ) // since no texture specified it's ok
        return true;

    if ( target == GL_TEXTURE_1D )
        id = createTexture1D ( mipmap, mapName.c_str () );
    else
        if ( target == GL_TEXTURE_2D )
            id = createTexture2D ( mipmap, mapName.c_str () );
        else
            if ( target == GL_TEXTURE_3D )
                id = createTexture3D ( mipmap, mapName.c_str () );

    return id != 0;
}

bool Map :: setTextureMatrix ()
{
    if ( id == 0 || unit == -1 )
        return false;

    glMatrixMode ( GL_TEXTURE );
    glLoadIdentity ();
    glScalef      ( texScale.x, texScale.y, 1 );
    glTranslatef   ( texOffs.x, texOffs.y, 0 );
}
```

```
    return true;
}

bool Map :: bind ()
{
    if ( id == 0 || unit == -1 )
        return false;

    glActiveTextureARB ( GL_TEXTURE0_ARB + unit );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture       ( GL_TEXTURE_2D, id );

    return true;
}
```

Для удобства работы можно поместить весь набор стандартных текстур (как объектов класса `Map`) внутрь специального класса `Material`, на который возложена вся ответственность за работу (загрузку, выбор и уничтожение) соответствующих текстур. В листингах 11.3 и 11.4 приведены описание и реализация класса `Material`.

Листинг 11.3. Описание класса `Material`

Листинг 11.4. Реализация класса Material

```
void Material :: load ()      // load all maps
{
    diffuse.    load ();
    specular.   load ();
    reflection.load ();
    bump.       load ();
    tex1.       load ();
    tex2.       load ();
    tex3.       load ();
    tex4.       load ();
}

void Material :: bind ()      // bind all maps to texture units
{
    diffuse.    bind ();
    specular.   bind ();
    reflection.bind ();
    bump.       bind ();
    tex1.       bind ();
    tex2.       bind ();
    tex3.       bind ();
    tex4.       bind ();
}

// update texture file ref with correct path
void Material :: setPath ( const string& path )
{
    if ( diffuse.mapName != "" )
        diffuse.mapName = buildFileName ( path, diffuse.mapName );

    if ( specular.mapName != "" )
        specular.mapName = buildFileName ( path, specular.mapName );

    if ( reflection.mapName != "" )
        reflection.mapName = buildFileName ( path, reflection.mapName );

    if ( bump.mapName != "" )

```

```

bump.mapName = buildFileName ( path, bump.mapName );

if ( tex1.mapName != "" )
    tex1.mapName = buildFileName ( path, tex1.mapName );

if ( tex2.mapName != "" )
    tex2.mapName = buildFileName ( path, tex2.mapName );

if ( tex3.mapName != "" )
    tex3.mapName = buildFileName ( path, tex3.mapName );

if ( tex4.mapName != "" )
    tex4.mapName = buildFileName ( path, tex4.mapName );
}

```

Метод `load` данного класса осуществляет загрузку всех текстур (для которых задано имя файла) путем делегирования запроса `load` всем текстурам.

Метод `bind` данного класса осуществляет выбор всех существующих текстур в соответствующие текстурные блоки (фактически он делегирует запрос `bind` всем текстурам).

Метод `setPath` устанавливает специальный путь доступа для всех текстур материала (поскольку часто набор текстур для модели хранится в каком-либо особом месте).

Рассмотрим теперь собственно класс `Mesh`, предназначенный для инкапсуляции объекта, состоящего из набора граней, вершин, ребер и текстур (листинги 11.5 и 11.6).

Листинг 11.5. Описание класса `Mesh`

```

class Mesh
{
protected:
    string name;
    int numVertices;
    int numFaces;
    int numEdges;
    Vertex * vertices;
    Face   * faces;
    Edge   * edges;
}

```

```
unsigned vertexBuffer;           // VBO for vertices
unsigned indexBuffer;           // VBO for face indices

           // mapping from vertex data to OpenGL data
list <pair <int,int> > mapping;

           // bounding box for mesh
BoundingBox      boundingBox;

           // set of textures for mesh
Material        material;

public:
Mesh ( const char * theName, Vertex * theVertices, int nv,
       Face * theFaces, int nf, bool copy = true );
~Mesh ();

const string&  getName () const
{
    return name;
}
           // access to verices
int  getNumVertices () const
{
    return numVertices;
}

const Vertex&  getVertex ( int index ) const
{
    return vertices [index];
}

Vertex&  getVertex ( int index )
{
    return vertices [index];
}

           // access to faces
int  getNumFaces () const
```

```
{  
    return numFaces;  
}  
  
const Face& getFace ( int index ) const  
{  
    return faces [index];  
}  
  
Face& getFace ( int index )  
{  
    return faces [index];  
}  
  
        // access to edges  
intgetNumEdges () const  
{  
    return numEdges;  
}  
  
const Edge& getEdge ( int index ) const  
{  
    return edges [index];  
}  
  
Edge& getEdge ( int index )  
{  
    return edges [index];  
}  
  
const Material& getMaterial () const  
{  
    return material;  
}  
  
Material& getMaterial ()  
{  
    return material;  
}
```

```
void setMaterial ( const Material& mat )
{
    material = mat;
}

// add assignment for source data
// to outputs from in
// (tagVertex,...,tagBinormal)
// to in
// (tagVertex,tagNormal,tagColor,
// tagTex0,...,tagTex7)
bool addCoordAssignment ( int from, int to );

// create VBO's
void createBuffers ();

// render the mesh
void render ();

// compute tangents and binormals
// from texCoords
void computeTangents ();

// compute edge data from vertices
// and faces
void computeEdges ();

// compute face normals
void computeFaceNormals ();

enum // enums for what
{
    tagVertex = 0,
    tagNormal = 1,
    tagColor = 2,
    tagTexCoord = 3,
    tagTangent = 4,
    tagBinormal = 5
};

enum // enums for where
```

```

    {
        // (plus tagVertex, tagColor,
        // tagNormal)
        tagTex0 = 10,      // tex coords for unit 0
        tagTex1 = 11,      // tex coords for unit 1
        tagTex2 = 12,
        tagTex3 = 13,
        tagTex4 = 14,
        tagTex5 = 15,
        tagTex6 = 16,
        tagTex7 = 17
    };

protected:
    void addEdge ( int v1, int v2, int face );
};

```

Как видно из листинга 11.5, объект класса `Mesh` содержит внутри себя массив вершин (`vertices`), граней (`faces`) и ребер (`edges`), ограничивающее тело (`boundingBox`) и набор текстур (`material`) (рис. 11.4).

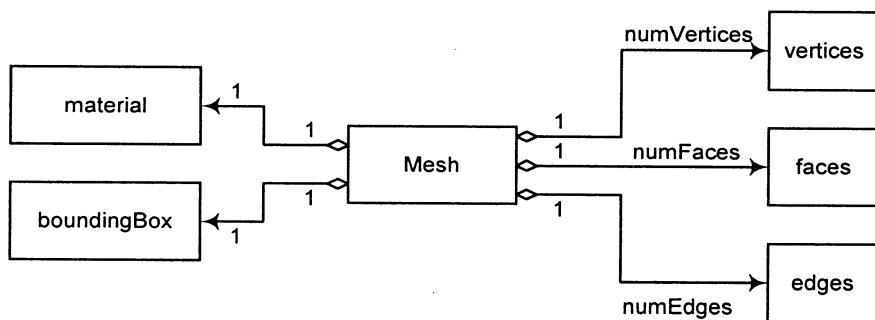


Рис. 11.4. Строение класса `Mesh`

Данный класс предназначен для хранения всех необходимых атрибутов вершин и индексов для всех граней в памяти GPU, используя для этого вершинные буфера, предоставляемые расширением `ARB_vertex_buffer_object` [1].

Для этих целей используется два таких буфера — один для хранения всех атрибутов вершин (`vertexBuffer`), другой — для хранения индексов для всех граней в массив вершин (`indexBuffer`).

Рассмотрим подробнее назначение основных методов данного класса.

Методы `getName`, `getNumVertices`, `getVertex`, `getNumFaces`, `getFace`, `getNumEdges`, `getEdge`, `getMaterial` и `setMaterial` осуществляют доступ

к основным данным объекта — имени, вершинам, граням, ребрам и материалу.

Важную роль играет метод `addCoordAssignment`, его задача — установить стандартные вершинные атрибуты OpenGL, в которые отображаются поля структуры `Vertex` (рис. 11.5).

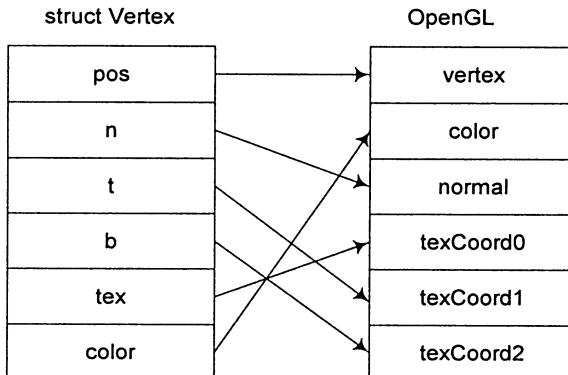


Рис. 11.5. Пример установки соответствия между полями структуры `Vertex` и вершинными атрибутами OpenGL

Первый параметр этого метода (`from`) является константой, идентифицирующей одно из полей структуры `Vertex`, и может принимать одно из значений — `tagVertex`, `tagNormal`, `tagTangent`, `tagBinormal`, `tagColor` или `tagTexCoord`.

Второй параметр (`to`) является константой, задающей один из стандартных вершинных атрибутов в OpenGL, и может принимать одно из значений — `tagVertex`, `tagNormal`, `tagColor`, `tagTex0`, `tagTex1`, ..., `tagTex7`.

Пример задания соответствия атрибутов:

```

addCoordAssignment ( tagVertex,      tagVertex );
addCoordAssignment ( tagNormal,     tagNormal );
addCoordAssignment ( tagTangent,    tagTex0   );
addCoordAssignment ( tagBinormal,   tagTex1   );
  
```

Соответствие, устанавливаемое в этом примере, показано на рис. 11.6.

Метод `computeEdges` по набору вершин и граней создает и инициализирует массив ребер (изначально он не инициализирован).

Метод `computeFaceNormals` вычисляет вектор нормали для каждой из граней объекта.

Метод `computeTangents` использует текстурные и пространственные координаты вершин для вычисления касательного вектора (`t`) и бинормали (`b`) в каждой вершине.

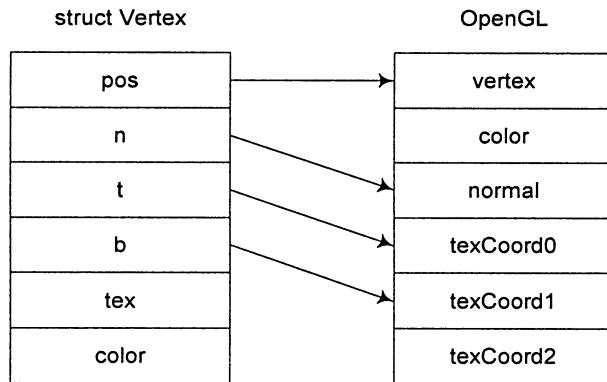


Рис. 11.6. Соответствие атрибутов, устанавливаемое в примере

После того как все требуемые атрибуты в вершинах вычислены, можно вызвать метод `createBuffers`, создающий вершинный и индексный буферы в памяти GPU и помещающий в них всю необходимую для рендеринга информацию о вершинах и гранях.

Метод `render` предназначен для рендеринга данного объекта. При этом устанавливается заданное соответствие между необходимыми атрибутами из созданного вершинного буфера (заданное при помощи метода `addCoordAssignment`), настраиваются текстуры и при помощи функции `glDrawElements` осуществляется рендеринг всех граней объекта.

Деструктор класса `Mesh` освобождает память, где хранятся используемые массивы, уничтожает созданные буферы в памяти GPU и освобождает загруженные текстуры.

Конструктор класса `Mesh` принимает в качестве входных параметров имя объекта (`theName`), указатель на массив вершин (`theVertices`) и их количество (`nv`), указатель на массив граней (`theFaces`) и их количество (`nf`), а также флаг (`copy`), который определяет, нужно ли создаваемому объекту создать свои массивы для вершин и граней (`copy = true`) или же можно воспользоваться ("присвоить") переданными (`copy = false`) (в последнем случае передающая их сторона не должна ни изменять, ни уничтожать их).

В листинге 11.6 приведена реализация класса `Mesh`.

Листинг 11.6. Реализация класса Mesh

```
#include <assert.h>
#include "libExt.h"
#include "Mesh.h"

static int vertexStride = sizeof ( Vertex );
```

```
static Vertex _v;

static int offsets [] =
{
    ((int)&_v.pos) - ((int)&_v),           // tagVertex
    ((int)&_v.n)   - ((int)&_v),           // tagNormal
    ((int)&_v.color) - ((int)&_v),          // tagColor
    ((int)&_v.tex)  - ((int)&_v),          // tagTexCoord
    ((int)&_v.t)    - ((int)&_v),          // tagTangent
    ((int)&_v.b)    - ((int)&_v)           // tagBinormal
};

void computeTangents (Vertex& v0, const Vertex& v1, const Vertex& v2);

Mesh :: Mesh ( const char * theName, Vertex * theVertices, int nv,
              Face * theFaces, int nf, bool copy )
{
    name        = theName;
    numVertices = nv;
    numFaces   = nf;
    numEdges   = 0;
    edges       = NULL;
    vertexBuffer = 0;
    indexBuffer = 0;

    if ( copy )
    {
        vertices = new Vertex [numVertices];
        faces    = new Face    [numFaces   ];

        memcpy ( vertices, theVertices, numVertices *
                  sizeof ( Vertex ) );
        memcpy ( faces,     theFaces, numFaces   *
                  sizeof ( Face   ) );
    }
    else
    {
        vertices = theVertices;
        faces    = theFaces;
    }
}
```

```
}

computeEdges ();

for ( int i = 0; i < numVertices; i++ )
    boundingBox.addVertex ( Vector3D ( vertices [i].pos.x,
                                      vertices [i].pos.y, vertices [i].pos.z ) );
}

Mesh :: ~Mesh ()
{
    delete vertices;
    delete faces;
    delete edges;

    if ( vertexBuffer != 0 )
        glDeleteBuffersARB ( 1, &vertexBuffer );

    if ( indexBuffer != 0 )
        glDeleteBuffersARB ( 1, &indexBuffer );
}

// add assignment for source data to outputs
bool Mesh :: addCoordAssignment ( int from, int to )
{
    if ( from < tagVertex || from > tagBinormal )
        return false;

    if ( to == tagVertex || to == tagNormal || to == tagColor ||
        (to >= tagTex0 && to <= tagTex7) )
    {
        mapping.push_back ( make_pair ( from, to ) );

        return true;
    }

    return false;
}
```

```
void Mesh :: createBuffers ()
{
    // create vertex data VBO
    glGenBuffersARB ( 1, &vertexBuffer );
    glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vertexBuffer );
    glBufferDataARB ( GL_ARRAY_BUFFER_ARB, numVertices *
                      sizeof ( Vertex ), vertices, GL_STREAM_DRAW_ARB );

    // create index data VBO
    int * buffer = new int [numFaces * 3];

    for ( int i = 0, j = 0; i < numFaces; i++ )
        for ( int k = 0; k < 3; k++ )
            buffer [j++] = faces [i].index [k];

    glGenBuffersARB ( 1, &indexBuffer );
    glBindBufferARB ( GL_ELEMENT_ARRAY_BUFFER_ARB, indexBuffer );
    glBufferDataARB ( GL_ELEMENT_ARRAY_BUFFER_ARB,
                      numFaces * 3 * sizeof ( int ), buffer,
                      GL_STATIC_DRAW_ARB );

    // unbind buffers
    glBindBufferARB ( GL_ARRAY_BUFFER_ARB, 0 );
    glBindBufferARB ( GL_ELEMENT_ARRAY_BUFFER_ARB, 0 );

    delete buffer;
}

// render the mesh
void Mesh :: render ()
{
    // save state
    glPushClientAttrib ( GL_CLIENT_VERTEX_ARRAY_BIT );

    // setup vertex buffer
    glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vertexBuffer );

    // go through the list of
    // mappings
```



```
}

    // setup index buffer
    glEnableClientState ( GL_INDEX_ARRAY );
    glBindBufferARB      ( GL_ELEMENT_ARRAY_BUFFER_ARB, indexBuffer);
    glIndexPointer       ( GL_UNSIGNED_INT, 0, 0 );

material.bind ();
                    // request draw
glDrawElements ( GL_TRIANGLES, 3*numFaces, GL_UNSIGNED_INT, 0 );

                    // unbind array buffer
glBindBufferARB ( GL_ARRAY_BUFFER_ARB, 0 );
glBindBufferARB ( GL_ELEMENT_ARRAY_BUFFER_ARB, 0 );

glPopClientAttrib ();

}

// compute tangents and binormals from texCoords
void Mesh :: computeTangents ()
{
    for ( int i = 0; i < numFaces; i++ )
    {
        Face& f = faces [i];

        :: computeTangents ( vertices [f.index [0]],
                            vertices [f.index [1]],
                            vertices [f.index [2]] );
    }
}

// compute edge data from vertices and faces
void Mesh :: computeEdges ()
{
    int maxEdges = 3 * numFaces;

    if ( edges != NULL )
        delete edges;

    numEdges = 0;
    edges     = new Edge [maxEdges];
```

```
for ( int i = 0; i < numFaces; i++ )
    for ( int j = 0; j < 3; j++ )
        addEdge ( faces [i].index [j],
                  faces [i].index [(j+1) % 3], i );
}

void Mesh :: addEdge ( int v1, int v2, int face )
{
    for ( int i = 0; i < numEdges; i++ )
    {
        bool b1 = ( edges [i].a == v1 ) && (edges [i].b == v2 );
        bool b2 = ( edges [i].a == v2 ) && (edges [i].b == v1 );

        if ( b1 || b2 )
        {
            edges [i].f2 = (b1 ? face : face | 0x80000000);

            return;
        }
    }

    Edge& edge = edges [numEdges];

    edge.a = v1;
    edge.b = v2;
    edge.f1 = face;
    edge.f2 = -1;

    numEdges++;
}

static inline Vector3D cross ( const Vector4D& v1, const Vector4D& v2 )
{
    return Vector3D(v1.x, v1.y, v1.z) ^ Vector3D(v2.x, v2.y, v2.z);
}

void Mesh :: computeFaceNormals ()
{
    if ( faces == NULL || vertices == NULL )
```

```
return;

for ( int i = 0 ; i < numFaces; i++ )
{
    Vector3D n = cross ( vertices [faces[i].index [1]].pos
                          - vertices [faces[i].index [0]].pos,
                          vertices [faces[i].index [2]].pos -
                          vertices [faces[i].index [0]].pos );

    faces [i].n = n.normalize ();
}

}

void computeTangents ( Vertex& v0, const Vertex& v1, const Vertex& v2 )
{
    Vector3D e0 ( v1.pos.x - v0.pos.x, v1.tex.x - v0.tex.x,
                  v1.tex.y - v0.tex.y );
    Vector3D e1 ( v2.pos.x - v0.pos.x, v2.tex.x - v0.tex.x,
                  v2.tex.y - v0.tex.y );
    Vector3D cp = e0 ^ e1;

    if ( fabs ( cp.x ) > EPS )
    {
        v0.t.x = -cp.y / cp.x;
        v0.b.x = -cp.z / cp.x;
    }
    else
    {
        v0.t.x = 0;
        v0.b.x = 0;
    }

    e0.x = v1.pos.y - v0.pos.y;
    e1.x = v2.pos.y - v0.pos.y;
    cp = e0 ^ e1;

    if ( fabs ( cp.x ) > EPS )
    {
```

```

v0.t.y = -cp.y / cp.x;
v0.b.y = -cp.z / cp.x;
}
else
{
    v0.t.y = 0;
    v0.b.y = 0;
}

e0.x = v1.pos.z - v0.pos.z;
e1.x = v2.pos.z - v0.pos.z;
cp = e0 ^ e1;

if ( fabs ( cp.x ) > EPS )
{
    v0.t.z = -cp.y / cp.x;
    v0.b.z = -cp.z / cp.x;
}
else
{
    v0.t.z = 0;
    v0.b.z = 0;
}

if ( ( ( v0.t ^ v0.b ) & v0.n ) < 0 )
    v0.t = -v0.t;
}

```

Несколько полезных утилит для создания объектов класса `Mesh` описаны в файле `MESHUTILS.H`, также входящем в состав библиотеки `libMesh` и находящемся в каталоге `CODE\LIBMESH` (листинг 11.7).

Листинг 11.7. Файл `MESHUTILS.H`

```

Mesh * makeTorus      ( float r1, float r2, int n1, int n2 );
Mesh * makeShape1     ( float r, float h, int n1, int n2, float factor );
Mesh * makeRevSurface ( const Vector2D d[], float tex[], int n, int n2 );

// rebuild vertices array

```

```
Vertex * fixFaces ( Vertex * vertices, int& numVertices, Face * faces,
                    Face * texFaces, int numFaces, Vector2D * texCoord,
                    int numTexCoords );
```

Функция `makeTorus` создает тор по заданным параметрам — радиусу тора и половине его толщины (радиусу протягиваемой для построения тора окружности) и разбиениям по обеим окружностям.

Функция `makeShape1` создает поверхность вращения, представленную на рис. 11.7.

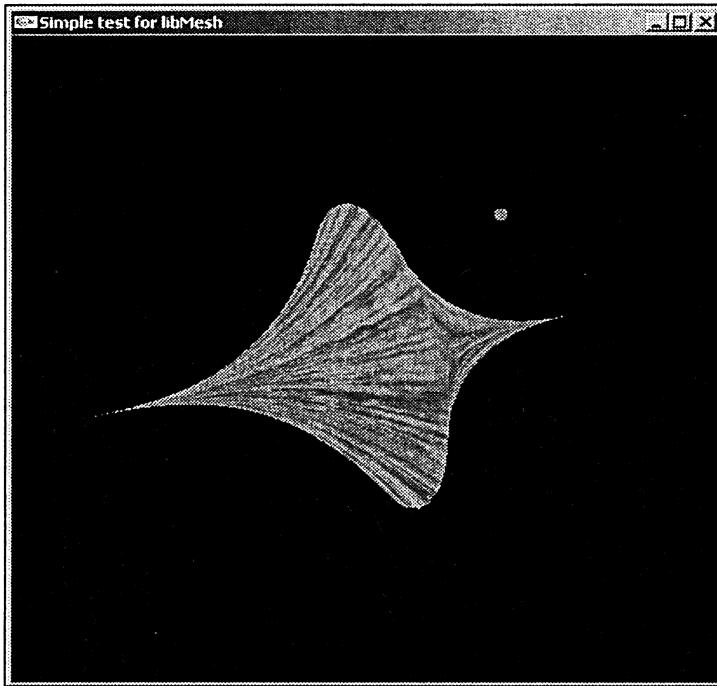


Рис. 11.7. Объект, возвращаемый функцией `makeShape1`

Функция `makeRevSurface` служит для построения поверхности вращения по заданному набору точек на плоскости и значениям одной текстурной координаты для этих точек. Параметр `n` задает размеры этих массивов, а параметр `n2` — число разбиений при вращении кривой вдоль оси `Ox`.

Функция `fixFaces` используется, когда для каждой грани есть отдельные наборы индексов (`faces` и `texFaces`) для вершин (`vertices`) и для массива текстурных координат (`texCoord`). В этом случае строится новый список вершин, в котором каждой вершине назначаются текстурные координаты. Если оказывается, что соответствующей вершине уже были назначены другие

текстурные координаты, то данная вершина дублируется (на этот раз с новыми текстурными координатами). В результате количество вершин может возрасти, но получившаяся в результате модель полностью соответствует принятым для класса `Mesh` соглашениям.

Однако многие трехмерные объекты имеют более сложную структуру, чем описанный класс `Mesh`. Довольно часто возникает ситуация, когда объект состоит из нескольких различных частей с разными текстурами или наличием определенных пространственных связей (так части тела связаны между собой). Большинство таких случаев может быть реализовано при помощи иерархической модели, диаграмма которой приведена на рис. 11.8.

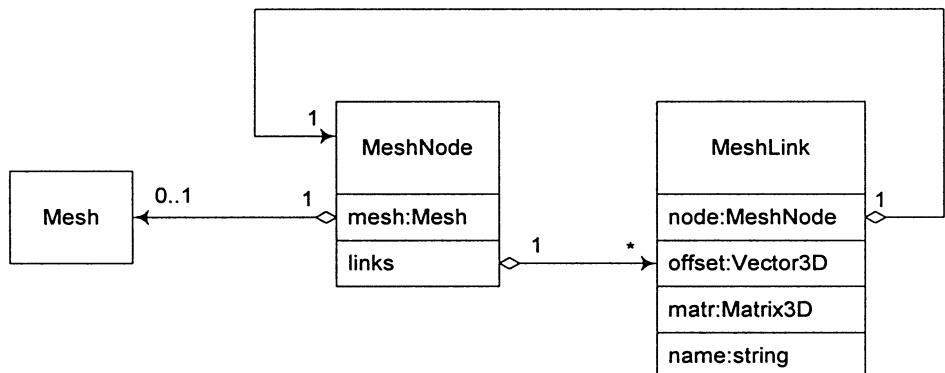


Рис. 11.8. Диаграмма для классов `Mesh`, `MeshNode` и `MeshLink`

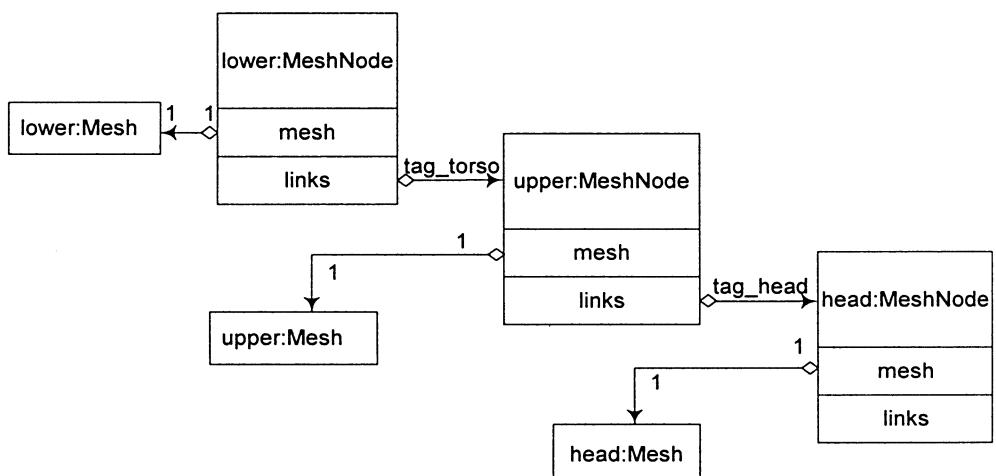


Рис. 11.9. Реализация иерархии частей в модели игрока из Quake 3 Arena при помощи классов `Mesh`, `MeshNode` и `MeshLink`

Как видно из этой диаграммы, основу составляет класс MeshNode, содержащий в себе ссылку на объект класса Mesh (или NULL), а также список ссылок на объекты MeshLink.

Каждый объект класса MeshLink обладает собственным именем, а также содержит аффинное преобразование и ссылку на объект класса MeshNode (к которому следует применить данное преобразование перед его выводом).

Так используемое в игре Quake 3 Arena представление игроков (как трех частей — head, lower и upper) легко реализуется через введенные классы, как показано на рис. 11.9.

В листингах 11.8 и 11.9 приведены описание и реализация классов MeshNode и MeshLink.

Листинг 11.8. Описание классов MeshNode и MeshLink

```
class Mesh;
class NodeVisitor;

class MeshNode
{
public:
    struct MeshLink // structure to represent a submesh
                    // with transform to it
    {
        MeshNode * node;
        Vector3D offset;      // do translation
        Matrix3D matr;
        string name;
    };
    typedef list <MeshLink *> Links;

protected:
    Mesh * mesh;
    Links links;

public:
    MeshNode ( Mesh * theMesh );
    ~MeshNode ();

                    // attach subnode
```

```

bool attach ( const char * linkName, Mesh * subMesh,
              const Vector3D& offs, const Matrix3D& m );
bool attach ( const char * linkName, MeshNode * node,
              const Vector3D& offs, const Matrix3D& m );
bool attach ( MeshLink * node );

Mesh * getMesh () const
{
    return mesh;
}

void setMesh ( Mesh * newMesh );

        // get direct subnode
MeshNode :: MeshLink * nodeWithName ( const char * name ) const;

void render ();           // render all hierarchy

        // perform hierarchy traversal
bool visit ( NodeVisitor& visitor );

        // iterating through links
Links :: const_iterator begin () const
{
    return links.begin ();
}

Links :: const_iterator end () const
{
    return links.end ();
}
};

```

Листинг 11.9. Реализация классов MeshNode и MeshLink

```

#ifndef _WIN32
#include <windows.h>
#endif

#include <GL/gl.h>

```

```
#include "MeshNode.h"
#include "Mesh.h"
#include "NodeVisitor.h"

MeshNode :: MeshNode ( Mesh * theMesh )
{
    mesh = theMesh;
}

MeshNode :: ~MeshNode ()
{
    delete mesh;

    for ( list <MeshLink *> :: iterator it = links.begin ();
          it != links.end (); ++it )
    {
        delete (*it) -> node;
        delete *it;
    }
}

void MeshNode :: setMesh ( Mesh * newMesh )
{
    delete mesh;

    mesh = newMesh;
}

bool MeshNode :: attach ( const char * linkName, Mesh * subMesh,
                        const Vector3D& offs, const Matrix3D& m )
{
    MeshLink * link = new MeshLink;

    link -> name      = linkName;
    link -> node      = new MeshNode ( subMesh );
    link -> offset    = offs;
    link -> matr      = m;

    links.push_front ( link );
}
```

```
    return true;
}

bool MeshNode :: attach ( const char * linkName, MeshNode * node,
                         const Vector3D& offs, const Matrix3D& m )
{
    MeshLink * link = new MeshLink;

    link -> name      = linkName;
    link -> node      = node;
    link -> offset    = offs;
    link -> matr      = m;

    links.push_front ( link );

    return true;
}

bool MeshNode :: attach ( MeshLink * node )
{
    links.push_front ( node );

    return true;
}

// get direct subnode
MeshNode :: MeshLink * MeshNode :: nodeWithName (
                                         const char * name ) const
{
    for ( list <MeshLink *> :: const_iterator it = links.begin ();
          it != links.end (); ++it )
        if ( (*it) -> name == name )
            return *it;

    return NULL;
}

// render all hierarchy
void MeshNode :: render ()
{
    if ( mesh != NULL )
```

```
mesh -> render ();

glMatrixMode ( GL_MODELVIEW );

for ( list <MeshLink *> :: iterator it = links.begin ();
      it != links.end (); ++it )
{
    MeshLink * link = *it;
    float     m [16];

    link -> matr.getHomMatrix ( m );

    glPushMatrix () ;

        // create appropriate transform matrix
    glTranslatef ( link -> offset.x, link -> offset.y,
                   link -> offset.z );
    glMultMatrixf ( m );

    if ( link -> node != NULL )
        link -> node -> render ();

    glMatrixMode ( GL_MODELVIEW );
    glPopMatrix ();
}

}

// perform hierarchy traversal
bool MeshNode :: visit ( NodeVisitor& visitor )
{
    if ( !visitor.visit ( this ) )
        return false;

    for ( list <MeshLink *> :: iterator it = links.begin ();
          it != links.end (); ++it )
        if ( visitor.visitLink ( *it ) )
            return false;

    return true;
}
```

Рассмотрим основные методы класса `MeshNode`.

Методы `attach` подсоединяют новый узел к текущему как дочерний (добавляют его к списку `links`) с заданными именем и преобразованием.

Методы `getMesh` и `setMesh` предназначены для доступа к *полигональному объекту* (`mesh`), связанному с данным узлом.

Для нахождения непосредственного дочернего узла по заданному имени предназначен метод `nodeWithName`.

Методы `begin` и `end` возвращают итераторы, которые можно использовать для обхода списка дочерних узлов.

Рендеринг всего составного объекта (включая вложенные объекты всех уровней) осуществляется с помощью функции `render`.

Еще одним вариантом доступа ко всей иерархии является метод `visit` (здесь используется шаблон Посетитель-Visitor).

Каталог CODE\LIBMESH на сопроводительном компакт-диске книги содержит исходный код для ряда классов, осуществляющих загрузку моделей из нескольких популярных форматов в объекты типа `meshNode`.

Одним из весьма распространенных (и достаточно простых) форматов хранения трехмерных моделей и сцен является формат ASE — текстовый формат довольно простой структуры. Хотя подобный файл может содержать практически всю информацию о сцене (включая камеры, источники света, анимацию), находящийся на компакт-диске загрузчик загружает только геометрию и ссылки на текстуры (листинг 11.10).

Листинг 11.10. Пример загрузки объекта из ASE-файла с явным назначением ему текстуры

```
MeshNode * root;

root = AseLoader ().load ( &Data ( ".../Models/fish.ase" ) );

if ( root == NULL )
{
    printf ( "Error loading ase file\n" );
    exit ( 1 );
}

Material mat;

mat.diffuse.mapName = ".../Textures/oak.bmp";

for ( MeshNode :: Links :: const_iterator it = root -> begin () ;
```

```
    it != root -> end (); ++it )  
{  
    Mesh * mesh = (*it) -> node -> getMesh ();  
  
    mesh -> addCoordAssignment ( Mesh :: tagVertex,  
                                    Mesh :: tagVertex );  
    mesh -> addCoordAssignment ( Mesh :: tagNormal,  
                                    Mesh :: tagNormal );  
    mesh -> addCoordAssignment ( Mesh :: tagTexCoord,  
                                    Mesh :: tagTex0 );  
    mesh -> computeTangents ();  
    mesh -> createBuffers ();  
    mesh -> setMaterial ( mat );  
    mesh -> getMaterial ().load ();  
}
```

Еще одним распространенным форматом является формат 3DS. Хотя данный формат и не является текстовым, устроен он крайне просто — файл представляет собой набор блоков (chunk). Каждый такой блок начинается с идентификатора его типа, длины и данных. Данные тоже могут представлять собой набор таких блоков (subchunk). Подобное строение файла позволяет легко извлекать из него всю необходимую информацию, пропуская ненужные (или даже неизвестные) части.

Пример загрузки 3DS-модели приведен в листинге 11.11.

Листинг 11.11. Пример загрузки 3DS-модели

```
MeshNode * root;  
  
root = ThreeDsLoader ().load (&Data ( "../Models/audi/audi.3ds" ));  
  
if ( root == NULL )  
{  
    printf ( "Error loading ase file\n" );  
    exit ( 1 );  
}  
  
for ( MeshNode :: Links :: const_iterator it = root -> begin ();  
      it != root -> end (); ++it )  
{
```

```

Mesh * mesh = (*it) -> node -> getMesh ();

mesh -> getMaterial      ().tex1.bindToUnit ( 0 );
mesh -> getMaterial      ().tex1.load        ();
mesh -> computeTangents  ();
mesh -> createBuffers    ();

mesh -> addCoordAssignment ( Mesh :: tagVertex,
                               Mesh :: tagVertex );
mesh -> addCoordAssignment ( Mesh :: tagNormal,
                               Mesh :: tagNormal );
mesh -> addCoordAssignment ( Mesh :: tagTexCoord,
                               Mesh :: tagTex0   );
}

}

```

Впервые появившийся в игре Quake 3 Arena формат MD3 предназначен для описания как отдельных моделей (оружия, предметов, аптечек и т. п.), так и моделей игроков. При этом каждая модель игрока строится из трех MD3-частей, задающих голову, верхнюю и нижнюю части тела, соединенные между собой.

Фрагмент кода, представленный в листинге 11.12, демонстрирует загрузку модели игрока из набора MD3-файлов.

Примечание

Соответствующий код полностью приведен в каталоге CODE\CHAPTER-11 сопроводительного компакт-диска книги.

Листинг 11.12. Пример загрузки модели игрока в MD3-формате

```

Data * headData, * lowerData, * upperData;
Data * headSkin, * lowerSkin, * upperSkin;

headData  = getFile ( " ../../models/players/laracroft/lara_head.MD3" );
lowerData = getFile ( " ../../models/players/laracroft/lara_lower.MD3" );
upperData = getFile ( " ../../models/players/laracroft/lara_upper.MD3" );
headSkin  = getFile ( " ../../models/players/laracroft/lara_head.skin" );
lowerSkin = getFile ( " ../../models/players/laracroft/lara_lower.skin" );
upperSkin = getFile ( " ../../models/players/laracroft/lara_upper.skin" );

addSearchPath ( " ../../" );

```

```
MeshNode * head = Md3Loader ().load ( headData, headSkin );
MeshNode * lower = Md3Loader ().load ( lowerData, lowerSkin );
MeshNode * upper = Md3Loader ().load ( upperData, upperSkin );

delete headData;
delete lowerData;
delete upperData;

lower -> nodeWithName ( "tag_torso" ) -> node = upper;
upper -> nodeWithName ( "tag_head" ) -> node = head;

root = lower;

if ( root == NULL )
{
    printf ( "Error loading md3 file\n" );
    exit ( 1 );
}

prepareNode ( root );
```

В листинге 11.13 приведен пример функции рекурсивной обработки узлов иерархии (назначения им необходимых атрибутов).

Листинг 11.13. Функция рекурсивного задания параметров для узлов иерархии

```
void prepareNode ( MeshNode * node )
{
    for ( MeshNode :: Links :: const_iterator it = node -> begin ();
          it != node -> end (); ++it )
    {
        MeshNode * n = (*it) -> node;

        if ( n == NULL )
            continue;

        prepareNode ( n );

        Mesh * mesh = n -> getMesh ();
    }
}
```

```
if ( mesh == NULL )
    continue;

mesh -> createBuffers ();
mesh -> getMaterial    ()->diffuse.bindToUnit ( 0 );
mesh -> getMaterial    ()->diffuse.load        ();

mesh -> addCoordAssignment ( Mesh :: tagVertex,
                               Mesh :: tagVertex );
mesh -> addCoordAssignment ( Mesh :: tagNormal,
                               Mesh :: tagNormal );
mesh -> addCoordAssignment ( Mesh :: tagTexCoord,
                               Mesh :: tagTex0   );
}

}
```

На рис. 11.10 показано изображение модели для игры Quake 3 Arena, загруженной приведенным выше способом.

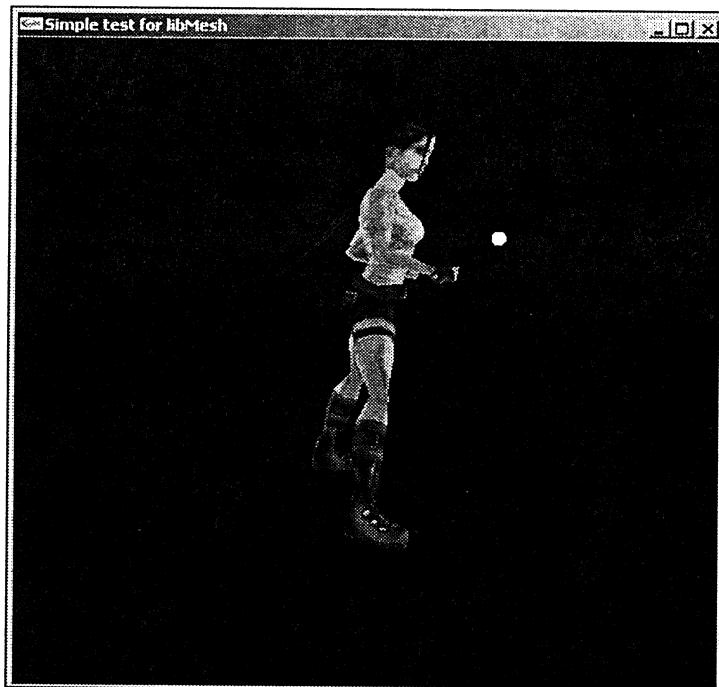


Рис. 11.10. Загруженная md3-модель

Еще одним форматом моделей из игр от idSoftware является использованный в игре Doom 3 формат MD5. На самом деле для описания моделей в игре используются два разных типа файлов — md5-mesh (определяет геометрию модели) и md5-anim (определяет анимацию модели).

В каталоге CODE\LIBMESH на сопроводительном компакт-диске вы найдете простой загрузчик для файлов типа md5-mesh. В листинге 11.14 приведен простой пример загрузки модели прямо из файлов игры. Полученное (без использования шейдеров) изображение представлено на рис. 11.11.

Листинг 11.14. Пример загрузки md5-mesh-файла

```
addZipFileSystem ( string ( DOOM_PATH ) + "pak002.pk4").c_str () );
addZipFileSystem ( string ( DOOM_PATH ) + "pak004.pk4").c_str () );

string modelName = "maggot3";

if ( argc > 1 )
    modelName = argv [1];

string fileName = string ( "models/md5/monsters/" ) +
                  modelName + "/" +
                  modelName + ".md5mesh";
Data * data = getFile
( fileName.c_str () );

root = Md5Loader ().load ( data, modelName );

if ( root == NULL )
{
    printf ( "Error loading md5 file\n" );
    exit ( 1 );
}

for ( MeshNode :: Links :: const_iterator it = root -> begin ();
      it != root -> end (); ++it )
{
    Mesh * mesh = (*it) -> node -> getMesh ();

    mesh -> createBuffers      ();
```

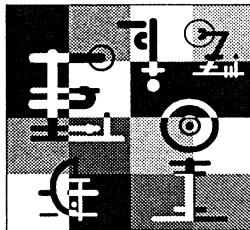
```
mesh -> getMaterial      () .diffuse.bindToUnit ( 0 );
mesh -> getMaterial      () .diffuse.load      ();

mesh -> addCoordAssignment ( Mesh :: tagVertex,
                             Mesh :: tagVertex );
mesh -> addCoordAssignment ( Mesh :: tagNormal,
                             Mesh :: tagNormal );
mesh -> addCoordAssignment ( Mesh :: tagTexCoord,
                             Mesh :: tagTex0   );
}

}
```



Рис. 11.11. Загруженная модель из Doom 3



Часть III

Шейдеры

Глава 12. Введение в GLSL. Описание синтаксиса, простые примеры

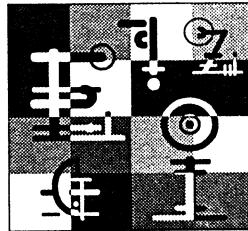
Глава 13. Простейшие вершинные и фрагментные шейдеры. Реализация основных моделей освещения

Глава 14. Практическое использование GLSL в программах на C++, необходимые расширения

Глава 15. Разработка шейдеров на GLSL в интегрированной среде RenderMonkey

Глава 16. Использование основных моделей освещения, моделирование преломления и дифракции, обработка изображений на GLSL

Глава 17. Использование шумовой функции в моделировании. Моделирование облаков, волн и материалов



Глава 12

Введение в GLSL. Описание синтаксиса, простые примеры

Язык GLSL очень близок к языкам C/C++, RenderMan Shading Language и Cg. При этом, по сравнению с C++, из языка GLSL "выброшены" элементы, затрудняющие его понимание и способствующие возникновению ошибок и неясностей.

В данный момент на языке GLSL можно писать шейдеры всего двух типов — вершинные и фрагментные (возможно, в дальнейшем появятся и новые типы шейдеров).

Основные типы данных и переменных

Как и в C++, в языке GLSL все переменные и функции перед использованием обязательно должны быть описаны. В языке отсутствуют какие-либо типы по умолчанию, сам язык является типово-безопасным (type-safe), т. е. в нем нет автоматического преобразования данных из одного типа в другой — все эти преобразования следует задавать явно в тексте самой программы.

В табл. 12.1 приведены основные типы языка GLSL, однако пользователь может вводить структуры и одномерные массивы.

Таблица 12.1. Основные типы языка GLSL

Тип	Комментарий
void	Функции, не возвращающие значение, должны быть описаны как void
bool	Булевский (логический) тип, принимающий всего два значения (<code>true</code> и <code>false</code>)
int	Целое число со знаком. Точность может зависеть от реализации, но гарантируется точность в 16 бит

Таблица 12.1 (окончание)

Тип	Комментарий
float	Вещественное скалярное значение
vec2	Двухмерный вектор вещественных (float) чисел
vec3	Трехмерный вектор вещественных чисел
vec4	Четырехмерный вектор вещественных чисел
bvec2	Двухмерный вектор логических значений
bvec3	Трехмерный вектор логических значений
bvec4	Четырехмерный вектор логических значений
ivec2	Двухмерный вектор целых чисел
ivec3	Трехмерный вектор целых чисел
ivec4	Четырехмерный вектор целых чисел
mat2	Матрица 2×2 вещественных чисел
mat3	Матрица 3×3 вещественных чисел
mat4	Матрица 4×4 вещественных чисел
sampler1D	Ссылка на одномерную текстуру
sampler2D	Ссылка на двухмерную текстуру
sampler3D	Ссылка на трехмерную текстуру
samplerCube	Ссылка на кубическую текстурную карту
sampler1DShadow	Ссылка на одномерную карту глубин
sampler2DShadow	Ссылка на двухмерную карту глубин

Примеры задания базовых типов:

```
int i, j = 42;
int k = 0xFF;
float a, b;
float c = 1.5, d = 1e-5;
vec2 v;
mat3 mv;
```

Структуры

Кроме стандартных типов пользователь может создавать свои типы — структуры. Обратите внимание: каждая структура должна иметь имя.

Пример задания структуры:

```
struct Light
{
    vec3    pos;
    float   intensity;
    vec4    color;
};

Light light1;
```

Структура должна содержать как минимум одно поле. В структурах не допускаются битовые поля; используемые типы должны быть или уже определены или определены прямо на месте. Структуры могут содержать в себе массивы заданной ненулевой длины.

Массивы

Кроме структур пользователь может вводить одномерные массивы фиксированной ненулевой длины. Допускается предварительное описание массива без задания его длины, тогда далее он должен быть описан с заданием размера.

Не допускается повторное описание массива с размером больше первоначально заданного или равным ему.

При обращении к массиву по отрицательному индексу или по индексу, выходящему за размер массива, поведение программы непредсказуемо. При использовании массива в качестве формального параметра функции задание размера обязательно.

Индексация массива начинается с нуля; доступ к компонентам массива производится при помощи оператора [].

Примеры описания массивов:

```
float      frequencies [4];
uniform vec3  directions [4];
Light      light [3];
```

Переменные

Область видимости *переменной* определяется местом ее описания. Если переменная описана вне всех функций, то она является *глобальной* и доступна отовсюду, начиная с места ее объявления.

Если переменная задана в условии оператора `while` или в операторе `for`, то областью ее видимости является следующий подоператор (тело цикла).

Если переменная определена внутри составного оператора, то область ее видимости ограничена концом соответствующего составного оператора.

Переменная, объявленная внутри функции, действует внутри данной функции.

Глобальные переменные, имеющие одинаковые имена в нескольких шейдерах, являются общими для них и должны иметь одинаковые типы.

Кроме собственно типа переменной (т. е. типа принимаемых ею значений) описание переменной может включать в себя дополнительные описатели, расположенные перед именем типа (табл. 12.2).

Таблица 12.2. Возможные типы описателей переменных

Тип	Комментарий
Не указан	Локальная переменная <code>read/write</code> или входной параметр для функции
<code>const</code>	Константа времени компиляции или параметр <code>read/only</code> функции
<code>attribute</code>	Вершинный атрибут. Обеспечивает связь между вершинным шейдером и вершинными данными в OpenGL
<code>uniform</code>	Величина не меняет своего значения вдоль всего обрабатываемого примитива. Эти переменные образуют связь между вершинным шейдером, OpenGL и приложением
<code>varying</code>	Обеспечивает связь между вершинным и фрагментным шейдером; интерполируется (с учетом перспективы) вдоль всего примитива
<code>in</code>	Входной параметр для функции
<code>out</code>	Выходной параметр функции, не инициализированный при передаче в функцию
<code>inout</code>	Параметр функции, предназначенный как для передачи данных в функцию, так и для возвращаемого значения

В описании глобальной переменной можно использовать только описатели `const`, `attribute`, `uniform` или `varying`, указав только *один* из этих описателей.

Для локальных переменных допускается только описатель `const`.

Для описания параметров функций можно использовать только описатели `const`, `in`, `out` и `inout`. Если при описании глобальной переменной не был задан ни один из этих описателей, то данная переменная не может участвовать в обмене данными с приложением, OpenGL и другими шейдерами.

Атрибуты (описатель `attribute`)

Описатель `attribute` служит для описания *атрибутов* — величин, передаваемых вершинному шейдеру из OpenGL с каждой вершиной. Переменные с этим описателем можно объявлять только в вершинном шейдере.

В вершинном шейдере атрибуты доступны только для чтения.

В качестве атрибутов могут выступать только переменные следующих типов: `float`, `vec2`, `vec3`, `vec4`, `mat2`, `mat3` и `mat4`.

Массивы и структуры нельзя описывать как атрибуты.

Примеры задания атрибутов:

```
attribute vec4 position;  
attribute vec3 normal;  
attribute vec2 texCoord;
```

Всем стандартным вершинным атрибутам OpenGL сопоставлены имена встроенных переменных (см. табл. 12.9).

Обычно место под вершинные атрибуты (за исключением матриц) выделяется как под четырехмерный вещественный вектор (`vec4`), за исключением матриц, представленных соответственно при помощи двух, трех или четырех четырехмерных векторов.

Примечание

Обычно реализации языка GLSL накладывают ограничения на количество используемых вершинных атрибутов. Далее в этой главе будет приведена программа, печатающая информацию о поддержке GLSL, включая различные ограничения, накладываемые графическим ускорителем и его драйвером.

Все атрибуты должны быть глобальными, и объявлены вне функций до их первого использования.

Если вершинный атрибут (кроме стандартных) не был проинициализирован, то ему присваивается начальное значение (0, 0, 0, 1).

***uniform*-переменные**

Описатель `uniform` используется для описания глобальных переменных, значения которых не изменяются вдоль обрабатываемого примитива. Простейшим примером таких переменных являются величины, задаваемые вне блока `glBegin / glEnd`.

Все такие переменные в шейдерах доступны только для чтения, и запись в них может производиться либо явно самим приложением через соответствующий интерфейс API, либо посредством самой библиотеки OpenGL.

Описатель `uniform` можно использовать для переменных любых типов, массивов и структур.

При совместной сборке нескольких шейдеров (например, вершинного и фрагментного) они обладают общим пространством имен `uniform`-переменных.

Реализация языка GLSL также может накладывать ограничения на количество используемых uniform-переменных.

Примеры описания uniform-переменных:

```
uniform vec4 lightPosition;
uniform vec4 lightColor;
```

varying-переменные

Эти переменные служат для связи между вершинным и фрагментным шейдерами. Вершинный шейдер, вычисляя значения для вершин, записывает их в соответствующие varying-переменные (вершинный шейдер также может читать записанные им ранее значения этих переменных).

При передаче фрагментному шейдеру эти переменные интерполируются вдоль всего примитива (с учетом перспективы), и фрагментный шейдер получает уже проинтерполированное значение для каждого фрагмента.

В фрагментном шейдере varying-переменные доступны только для чтения.

Поскольку эти переменные предназначены для связи между вершинным шейдером и фрагментным шейдером, они должны быть объявлены в каждом из этих шейдеров и их типы должны соответствовать.

Примеры описания varying-переменных:

```
varying vec3 normal;
varying vec3 t;
varying vec4 color;
```

Атрибут varying может применяться только к переменным следующих типов — float, vec2, vec3, vec4, mat2, mat3, mat4 или к массивам таких переменных. Структуры нельзя описывать как varying.

Как и uniform-переменные, varying-переменные должны быть глобальными, объявленными до первого использования.

Операторы и выражения

В табл. 12.3 приведены все операторы языка GLSL с учетом их приоритетов и порядка выполнения.

Таблица 12.3. Операторы языка GLSL

Приоритет	Класс	Операторы	Ассоциативность
1 (наи высший)	Группировка при помощи скобок	()	Не применима
2	Индексирование массивов	[]	Слева направо

Таблица 12.3 (окончание)

Приоритет	Класс	Операторы	Ассоциативность
2	Вызов функции или конструктора	()	Слева направо
2	Выбор поля структуры или компонента поля	.	Слева направо
2	Постфиксное изменение значения на единицу	++, --	Слева направо
3	Префиксное изменение значения на единицу	++, --	Справа налево
3	Унарные операторы (оператор ~ зарезервирован)	+, -, ~, !	Справа налево
4	Мультиплексивные операторы (оператор ? зарезервирован)	*, /, %	Слева направо
5	Аддитивные операторы	+, -	Слева направо
6	Побитовые операторы (зарезервированы)	<<, >>	Слева направо
7	Сравнение	<, >, <=, >=	Слева направо
8	Равенство	==, !=	Слева направо
9	Побитовое "И" (зарезервировано)	&	Слева направо
10	Побитовое "Исключающее ИЛИ" (зарезервировано)	^	Слева направо
11	Побитовое "ИЛИ" (зарезервировано)		Слева направо
12	Логическое "И"	&&	Слева направо
13	Логическое "Исключающее ИЛИ"	^^	Слева направо
14	Логическое "ИЛИ"		Слева направо
15	Выбор	?, :	Справа налево
16	Присвоение	=	Справа налево
16	Изменение (операторы %=, сдвига и битовых операций зарезервированы)	+=, -=, *=, /=, %=, <<=, >>=, &=, =, ^=	Справа налево
17	Последовательность	,	Слева направо

Конструкторы

Конструкторы предназначены для получения (приведения) значения к заданному типу.

Обращение к конструкторам имеет вид вызова функции, где в качестве имени функции выступает имя базового типа или структуры. Конструкторы можно использовать для преобразования одного скалярного значения в другое, построения векторов, матриц и структур.

Не существует некоторого заранее заданного списка допустимых конструкторов. Необходимо, чтобы обращение к конструктору было лексически-корректным; типы и количество переданных параметров должны соответствовать требуемым для построения нужного значения.

Примеры преобразования скалярных типов:

```
int    i = int ( 1.5 );      // convert float to int
int    j = int ( true );     // convert bool to int
float  f = float ( false );  // convert bool to float
float  h = float ( 3 );      // convert int to float
bool   b = bool ( 2 );       // convert int to bool
bool   c = bool ( 1.4 );     // convert float to bool
```

При преобразовании `float`-значения к типу `int` отбрасывается дробная часть. При преобразовании к типу `bool` значения 0 и 0.0 переводятся в значение `false`, а все остальные значения — в значение `true`.

При преобразовании значений типа `bool` в типы `int` и `float` значение `false` переводится в ноль (0 или 0.0), а значение `true` переводится в единицу (1 или 1.0).

Конструкторы скалярных типов могут принимать в качестве входных значений значения нескалярных типов, возвращая в этом случае первый элемент переданного векторного (или матричного) значения.

Конструкторы можно использовать для получения векторов и матриц, а также наборов скалярных значений, векторов и матриц.

Если для инициализации вектора используется скалярное значение, то оно используется для инициализации всех компонентов вектора:

```
vec3  v = vec3 ( 0.5 );      // will build (0.5, 0.5, 0.5)
```

Если единственное скалярное значение принимается в качестве параметра для конструктора матрицы, то оно используется для инициализации главной диагонали этой матрицы, все остальные элементы матрицы инициализируются нулем:

```
mat2  m = mat2 ( 1.0 );      // will build identity matrix I
```

Если заданы нескалярные параметры и/или несколько скалярных параметров, то они будут присваиваться компонентам строящегося значения в порядке слева направо. Если в последнем аргументе было передано больше компонентов, чем нужно для построения соответствующего значения, то лишние компоненты будут просто проигнорированы. Таким способом можно строить более короткие векторы из более длинных.

Примечание

Передача дополнительных аргументов после последнего необходимого является ошибкой.

Матрицы строятся по столбцам (сначала первый столбец, затем второй и т. д.). Не допускается построение матриц из матриц.

Примеры инициализации векторов и матриц:

```
vec2 v2 = vec2 ( 1.0, 2.0 ); // создаст (1.0, 2.0)
vec3 v3 = vec3 ( v2, 0.3 ); // создаст (1.0, 2.0, 0.3)
vec4 v4 = vec4 ( v2, v2 ); // создаст (1.0, 2.0, 1.0, 2.0)
vec4 v5 = vec4 ( 4.5, v3 ); // создаст (4.5, 1.0, 2.0, 0.3)
vec4 rgba = vec4 ( 0.0 ); // создаст (0.0, 0.0, 0.0, 0.0)
vec3 rgb = vec3 ( rgba ); // создаст (0.0, 0.0, 0.0)
mat3 m3 = mat3 ( 2.0 ); // создаст 2*IdentityMatrix
mat2 m2 = mat2 ( v2, v2 ); // создаст матрицу, у которой оба
                           // столбца будут равны (1.0, 2.0)
mat2 m = mat2 ( 1.0, 0.0, 0.0, 1.0 ); // создаст двухмерную
                                         // единичную матрицу
```

Для инициализации структур используется конструктор, имя которого совпадает с именем самой структуры. Параметрами являются значения тех же типов и в том же порядке, в каком они перечислены в описании структуры.

Пример описания структуры:

```
Light light2 = Light ( p, 0.4, vec4 ( 1.0, 1.0, 0.0, 1.0 ) );
```

Работа с компонентами векторов и матриц

Для обозначения компонентов векторов используются символные обозначения (например, x , y , z). Поскольку с различными применениями векторов (как координат в пространстве, цветов и текстурных координат) связаны различные имена компонентов, то в GLSL возможно использовать три набора имен компонентов, приведенных в таблице 12.4.

Таблица 12.4. Обозначения групп компонентов векторов, используемые в GLSL

Обозначение	Использование
(x, y, z, w)	Удобно для доступа к векторам, обозначающим положение или направление
(r, g, b, a)	Удобно для доступа к векторам, представляющим собой значения цвета
(s, t, p, q)	Удобно для доступа к векторам, являющимся текстурными координатами

Примечание

Третий компонент в текстурных координатах (λ) был переименован в p , чтобы не возникло путаницы с r -компонентом цвета.

Для доступа к компонентам вектора используется точка (.), при этом после точки может идти произвольный набор имен компонентов, что позволяет перемешивать компоненты и/или дублировать их. Имена компонентов, указанные после точки, должны принадлежать одной и той же группе компонентов, указанной в табл. 12.4. Таким образом запись `myVector.xgr` является недопустимой, т. к. в ней использованы имена компонентов из всех трех групп.

Использование набора имен компонентов после точки предоставляет еще один способ (кроме применения конструкторов) для построения одних векторов из других.

Кроме того, конструкции с именами компонентов после точки можно использовать в левой части оператора присваивания, обеспечивая запись только в заданные поля вектора.

Примеры *правильного* использования имен компонентов векторов:

```

vec4 v4 ( 1.0, 2.0, 3.0, 4.0 );
float f1 = v4.x;      // все величины f1, f2 и f3 получат значение e
float f2 = v4.r;      // одного и того же компонента v
float f3 = v4.s;
vec3 v3 = v4.xzy;
vec4 v5 = v4.wzyx;
vec3 v6 = v4.zxx;

v6.xz = 1.0;

```

Примеры *неправильного* использования имен компонентов векторов:

```
v6.xx = 1.0;          // два раза используется компонент x
```

```
v6.xy = vec3 ( 1.0, 1.0, 2.0 ); // несовпадение размеров  
  
vec2 v7 = v5.xgbt; // используются имена полей  
// из различных групп
```

Также для доступа к компонентам векторов можно использовать *индексацию* — вектор трактуется как массив из соответствующего числа элементов (2, 3 или 4); как и для обычных массивов, индексация начинается с нуля. Таким образом, обе записи `v[0]` и `v.x` относятся к одному и тому же (первому) элементу вектора.

Для доступа к элементам матрицы используется индексирование, при этом задание только первого компонента возвращает соответствующий столбец матрицы, т. е. матрица размера N трактуется как массив из N векторов-столбцов (состоящих каждый из N элементов). Крайнему левому столбцу соответствует индекс 0, т. е. столбцы нумеруются слева направо. При использовании двух индексов первый из них указывает столбец матрицы, а второй индекс — элемент столбца.

Примеры обращения к матрице и ее элементам:

```
mat4 m;  
m [1] = vec4 ( 0.0 ); // set second column to zeroes  
m [0][0] = 1.0; // set top left element to one  
m [2][3] = 2.0;
```

Работа со структурами

Как и для выбора компонентов вектора, для обращения к полям структуры используется точка `(.)`.

К структурам применимы только три оператора — обращение к полю `(.)`, сравнение на равно/не равно `(== и !=)` и присваивание `(=)`.

Сравнение структур трактуется как покомпонентное сравнение каждого из полей.

Основные операции над векторами и матрицами

Большинство операций над векторами и матрицами (за некоторыми исключениями) являются покомпонентными, т. е. независимо выполняются для каждой пары соответствующих компонентов (рис. 12.1 и 12.2).

Покомпонентными операциями *не* являются операции умножения матрицы на вектор (рис. 12.3), вектора на матрицу (см. рис. 12.5) и матрицы на матрицу (рис. 12.4). Эти операции предъявляют определенные требования к размерам operandов.

```
vec3 u, v;
float f;
v = u + f;
```

Эквивалентно

```
v.x = u.x + f;
v.y = u.y + f;
v.z = u.z + f;
```

Рис. 12.1. Покомпонентная операция (пример 1)

```
vec4 a, b;
vec4 c;
c = a * b;
```

Эквивалентно

```
c.x = a.x * b.x;
c.y = a.y * b.y;
c.z = a.z * b.z;
c.w = a.w * b.w;
```

Рис. 12.2. Покомпонентная операция (пример 2)

```
vec3 u, v;
mat3 m;
v = m * u;
```

Эквивалентно

```
v.x = dot ( u, m [0] );
v.y = dot ( u, m [1] );
v.z = dot ( u, m [2] );
```

Рис. 12.3. Умножение матрицы на вектор (не покомпонентная операция)

```
mat2 a, b;
mat2 c;
c = a * b;
```

Эквивалентно

```
c[0].x = a[0].x * b[0].x + a[1].x * b[0].x;
c[1].x = a[0].x * b[1].x + a[1].x * b[1].y;
c[0].y = a[0].y * b[0].x + a[1].y * b[0].y;
c[1].y = a[0].y * b[1].x + a[1].y * b[1].y;
```

Рис. 12.4. Умножение матрицы на матрицу (не покомпонентная операция)

```
vec3 u, v;
mat3 m;
v = u * m;
```

Эквивалентно

```
v.x = m[0].x * v.x + m[1].x * v.y + m[2].x * v.z;
v.y = m[0].y * v.x + m[1].y * v.y + m[2].y * v.z;
v.z = m[0].z * v.x + m[1].z * v.y + m[2].z * v.z;
```

Рис. 12.5. Умножение вектора на матрицу (не покомпонентная операция)

Для вычисления скалярного произведения векторов предназначена функция `dot` (см. табл. 12.10—12.18).

В случае умножения вектора на матрицу вектор трактуется как строка, умножаемая слева на матрицу (рис. 12.5). Сами же эти операции полностью эквивалентны операциям, вводимым в курсе линейной алгебры.

Все унарные операции выполняются только покомпонентно.

Основные операторы и конструкции

Имена переменных и функций в GLSL строятся из заглавных (больших) и строчных (маленьких) латинских букв, цифр и символа подчеркивания (_), при этом цифра не может быть первым символом в имени.

Имена с префиксом `gl_` зарезервированы и не могут быть использованы для обозначения пользовательских переменных и функций.

Комментариями в языке GLSL, как и в C++, считается текст любой длины, расположенный между символами `/*` (начало комментария) и `*/` (конец комментария), а также текст от символов `//` до конца строки.

Программы на GLSL сначала обрабатываются препроцессором, поэтому существует ряд команд для препроцессора (табл. 12.5).

Таблица 12.5. Команды для препроцессора GLSL

Команда	Значение
<code>#</code>	Строка, начинающаяся с этой команды, игнорируется
<code>#define</code>	Аналогично языку C (как с параметрами, так и без)
<code>#undef</code>	Аналогично языку C (как с параметрами, так и без)
<code>#if</code>	Аналогично языку C
<code>#ifdef</code>	Аналогично языку C
<code>#ifndef</code>	Аналогично языку C
<code>#else</code>	Аналогично языку C
<code>#elif</code>	Аналогично языку C
<code>#endif</code>	Аналогично языку C
<code>#error</code>	Помещает сообщение, идущее за этой командой и до конца строки, в лог компиляции; компиляция считается завершившейся неудачно (это способ искусственно вызвать ошибку с заданной диагностикой)
<code>#pragma</code>	Позволяет работать с особенностями отдельных компиляторов
<code>#extension</code>	Позволяет управлять различными расширениями языка
<code>#version</code>	Определяет версию языка, для которой написана данная программа
<code>#line</code>	Позволяет задать новую нумерацию для строк

Также доступен оператор `defined`, позволяющий узнать, определен ли уже данный символ.

В языке GLSL есть набор предопределенных макросов (табл. 12.6).

Таблица 12.6. Предопределенные макросы

Макрос	Значение
<code>__LINE__</code>	Содержит количество символов начала строки (\n) в текущей строке
<code>__FILE__</code>	Содержит номер текущей строки
<code>__VERSION__</code>	Содержит версию GLSL

Все макросы, имена которых содержат два стоящих рядом символа подчеркивания (`__`), зарезервированы для дальнейшего использования.

С помощью команды `#pragma` можно управлять режимом компиляции. Доступны следующие опции:

```
#pragma optimize(on)
#pragma optimize(off)
#pragma debug(on)
#pragma debug(off)
```

Язык GLSL поддерживает ряд операторов из языка C, управляющих выполнением программы:

```
if ( bool expression )
    ...
else
    ...
for ( initialization; bool expression; loop expression )
    ...
while ( bool expression )
    ...
do
    ...
while ( bool expression )
```

Также доступны команды для осуществления переходов в программе — `continue`, `break` и `discard`. Действие первых двух полностью аналогично их действию в языке C. Команда `discard`, доступная только в фрагментном шейдере, немедленно прекращает обработку текущего фрагмента (аналогично команде `KIL` для фрагментных программ из главы 2).

Примечание

Поддержка этих команд зависит от конкретного графического ускорителя. На ряде графических ускорителей поддерживаются условный оператор и оператор цикла, для которого количество итераций известно на момент компиляции (т. е. его можно преобразовать в линейную последовательность команд), но операторы цикла общего вида не поддерживаются.

Программы на языке GLSL также могут содержать функции. Обратите внимание: для каждой функции должен быть явно задан тип возвращаемого значения:

```
returnType functionName ( type0 arg0, type1 arg1, ..., typen argn )
{
    // do some computation
}
```

Также можно просто объявлять функции, задавая их реализацию позже:

```
returnType functionName ( type0 arg0, type1 arg1, ..., typen argn );
```

В листинге 12.1 приведен пример введения функции.

Листинг 12.1. Пример задания функции на GLSL

```
vec4 toonify ( in float intensity )
{
    vec4 color;

    if ( intensity > 0.98 )
        color = vec4 ( 0.8, 0.8, 0.8, 1.0 );
    else
        if ( intensity > 0.5 )
            color = vec4 ( 0.4, 0.4, 0.4, 1.0 );
        else
            if ( intensity > 0.25 )
                color = vec4 ( 0.2, 0.2, 0.2, 1.0 )
            else
                color = vec4 ( 0.1, 0.1, 0.1, 1.0 );

    return color;
}
```

Каждый шейдер должен иметь одну главную точку входа — функцию `main`, описанную следующим образом:

```
void main (void)
{
    // Здесь могут находиться операторы GLSL
}
```

Стандартные переменные, атрибуты и константы

И для вершинного и для фрагментного шейдеров определен набор стандартных переменных, атрибутов и констант, которые не требуют никаких объявлений, их тип и смысл заранее определены.

Все имена стандартных переменных начинаются с префикса `gl_`.

Специальные переменные для вершинных шейдеров

В каждом вершинном шейдере определены три стандартные переменные (табл. 12.7), представленные следующими объявлениями:

```
vec4 gl_Position;
float gl_PointSize;
vec4 gl_ClipVertex;
```

Все эти переменные являются глобальными, т. е. доступны в любом месте вершинного шейдера и не требуют объявления.

Переменная `gl_Position` доступна только в вершинном шейдере и в нее обязательно должны быть записаны однородные координаты вершины. Эта переменная доступна и для чтения.

Переменная `gl_Pointsize` также доступна только в вершинном шейдере. Шейдер может записать в нее размер точки для растеризации (в пикселях). Если запись значения в эту переменную не производилась, то ее чтение дает неопределенное значение.

Переменная `gl_ClipVertex` также доступна только в вершинном шейдере, в нее могут быть записаны координаты для использования с задаваемыми пользователем плоскостями отсечения. Если запись значения в эту переменную не производилась, то ее чтение дает неопределенное значение.

Примечание

Убедитесь, что переданные координаты и задаваемые пользователем плоскости отсечения заданы в одном и том же координатном пространстве.

Таблица 12.7. Специальные переменные для вершинного шейдера

Переменная	Тип	Запись	Чтение
gl_Position	vec4	Обязательна	Возможно, но если до этого запись не производилась, значение переменной не определено
gl_PointSize	float	Не обязательна	Возможно, но если до этого запись не производилась, значение переменной не определено
gl_ClipVertex	vec4	Не обязательна	Возможно, но если до этого запись не производилась, значение переменной не определено

Специальные переменные для фрагментных шейдеров

Для фрагментного шейдера определены специальные переменные (табл. 12.8):

```
vec4 gl_FragCoord;
bool gl_FrontFacing;
vec4 gl_FragColor;
vec4 gl_FragData [gl_MaxDrawBuffers];
float gl_FragDepth;
```

Фрагментный шейдер должен либо отбросить фрагмент (команда `discard`), либо записать параметры фрагмента в переменные `gl_FragColor`, `gl_FragData` и `gl_FragDepth`.

В переменную `gl_FragColor` фрагментный шейдер записывает итоговый цвет (в формате RGBA) для данного фрагмента.

В переменную `gl_FragDepth` шейдер может записать выходное нормированное (приведенное к отрезку [0, 1]) значение глубины фрагмента. При этом если шейдер не производит запись в эту переменную, то в качестве глубины используется значение, полученное при растеризации соответствующего примитива. Однако если в шейдере присутствует хотя бы одна команда для записи в переменную `gl_FragDepth` (для какого-то пути выполнения), то шейдер отвечает за запись в эту переменную, в противном случае выходное значение глубины окажется неопределенным.

Переменная `gl_FragData` является массивом выходных цветов для различных буферов, т. е. шейдер может осуществлять рендеринг сразу в несколько буферов цвета. При этом если шейдер производит запись в переменную `gl_FragColor`, то он не может записывать значения ни в один элемент массива `gl_FragData` и наоборот.

В случае выполнения шейдером команды `discard` значения переменных `gl_FragColor`, `gl_FragDepth` и `gl_FragData` становятся не важны.

В доступной только для чтения переменной `gl_FragCoord` содержатся координаты (x , y , z) и $1 / w$ для текущего фрагмента относительно окна. Эти значения получаются путем интерполяции при растеризации соответствующего примитива; z -компонент этой переменной содержит значение глубины, которое будет использовано, если шейдер не производит запись в `gl_FragDepth`.

Также фрагментному шейдеру доступна (только для чтения) булевская переменная `gl_FrontFacing`, принимающая значение `true`, если фрагмент принадлежит лицевому примитиву.

Таблица 12.8. Специальные переменные для фрагментного шейдера

Переменная	Тип	Запись	Чтение
<code>gl_FragCoord</code>	<code>vec4</code>	Нет	Возможно
<code>gl_FrontFacing</code>	<code>bool</code>	Нет	Возможно
<code>gl_FragColor</code>	<code>vec4</code>	Обязательна, если фрагмент не отбрасывается	Возможно, но если до этого записи не производилась, значение переменной не определено
<code>gl_FragData</code>	<code>vec4 []</code>	Обязательна, если фрагмент не отбрасывается и не производится запись в переменную <code>gl_FragColor</code>	Возможно, но если до этого записи не производилась, значение переменной не определено
<code>gl_FragDepth</code>	<code>float</code>	Не обязательна	Возможно, но если до этого записи не производилась, значение не определено

Значения цвета и глубины после записи в соответствующие выходные переменные автоматически отсекаются по отрезку $[0, 1]$.

Стандартные константы

Каждому шейдеру доступен ряд констант, определяющих ограничения, накладываемые как самим графическим ускорителем, так и драйвером для него (фактически поддержка GLSL обеспечивается именно через драйвер).

Стандартные константы в GLSL:

```
const int gl_MaxLights;           // >= 8
const int gl_MaxClipPlanes;      // >= 6
const int gl_MaxTextureUnits;    // >= 2
```

```

const int gl_MaxTextureCoords;           // >= 2
const int gl_MaxVertexAttribs;          // >= 16
const int gl_MaxVertexUniformComponents; // >= 512
const int gl_MaxVaryingFloats;          // >= 32
const int gl_MaxVertexTextureImageUnits; // >= 0
const int gl_MaxCombinedTextureImageUnits; // >= 2
const int gl_MaxTextureImageUnits;       // >= 2
const int gl_MaxFragmentUniformComponents; // >= 64
const int gl_MaxDrawBuffers;             // >= 1

```

Стандартные атрибуты для вершинного шейдера

Существует определенный набор атрибутов, всегда доступных вершинному шейдеру (табл. 12.9).

Таблица 12.9. Стандартные атрибуты для вершинного шейдера

Атрибут	Описание
vec4 gl_Color;	Цвет для вершины
vec4 gl_SecondaryColor;	Вторичный цвет для вершины
vec3 gl_Normal;	Нормаль в вершине
vec4 gl_Vertex;	Координаты вершины
vec4 gl_MultiTexCoord0;	Текстурные координаты для текстурного блока 0
vec4 gl_MultiTexCoord1;	Текстурные координаты для текстурных блоков 1–7
vec4 gl_MultiTexCoord2;	
vec4 gl_MultiTexCoord3;	
vec4 gl_MultiTexCoord4;	
vec4 gl_MultiTexCoord5;	
vec4 gl_MultiTexCoord6;	
vec4 gl_MultiTexCoord7;	
float gl_FogCoord;	Степень затуманивания вершины

Стандартные *uniform*-переменные — параметры состояния OpenGL

Также для шейдеров, написанных на языке GLSL, доступен набор *uniform*-переменных, содержащих в себе параметры состояния OpenGL.

Основные матричные uniform-переменные:

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];
uniform mat3 gl_NormalMatrix; // transpose of the inverse of the
                             // upper leftmost 3x3 of gl_ModelViewMatrix
uniform mat4 gl_ModelViewMatrixInverse;
uniform mat4 gl_ProjectionMatrixInverse;
uniform mat4 gl_ModelViewProjectionMatrixInverse;
uniform mat4 gl_TextureMatrixInverse[gl_MaxTextureCoords];
uniform mat4 gl_ModelViewMatrixTranspose;
uniform mat4 gl_ProjectionMatrixTranspose;
uniform mat4 gl_ModelViewProjectionMatrixTranspose;
uniform mat4 gl_TextureMatrixTranspose[gl_MaxTextureCoords];
uniform mat4 gl_ModelViewMatrixInverseTranspose;
uniform mat4 gl_ProjectionMatrixInverseTranspose;
uniform mat4 gl_ModelViewProjectionMatrixInverseTranspose;
uniform mat4 gl_TextureMatrixInverseTranspose[gl_MaxTextureCoords];
```

Эти переменные содержат стандартные матрицы OpenGL или их транспонированные и/или обращенные варианты.

Параметры, задающие приведение глубины к отрезку [0, 1], представлены в виде структуры `gl_DepthRangeParameters`:

```
struct gl_DepthRangeParameters
{
    float near;      // n
    float far;       // f
    float diff;      // f - n
};

uniform gl_DepthRangeParameters gl_DepthRange;
```

Информация о плоскостях отсечения представлена в виде массива `gl_ClipPlane` четырехмерных векторов, где плоскости задаются при помощи четырехмерных векторов (коэффициентов уравнения плоскости):

```
uniform vec4 gl_ClipPlane[gl_MaxClipPlanes];
```

Параметры, задающие свойства точки (расширение `ARB_point_parameters`, см. [1]), доступны в виде структуры `gl_PointParameters`:

```
struct gl_PointParameters
{
```

```
float size;
float sizeMin;
float sizeMax;
float fadeThresholdSize;
float distanceConstantAttenuation;
float distanceLinearAttenuation;
float distanceQuadraticAttenuation;
};

uniform gl_PointParameters gl_Point;
```

Также шейдеры имеют доступ к свойствам материала, представленным в виде структуры `gl_MaterialParameters`:

```
struct gl_MaterialParameters
{
    vec4 emission;
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    float shininess;
};

uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

Структура `gl_LightSourceParameters` предоставляет информацию о свойствах источников света:

```
struct gl_LightSourceParameters
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec4 position;
    vec4 halfVector;
    vec3 spotDirection;
    float spotExponent;
    float spotCutoff;
    float spotCosCutoff;
    float constantAttenuation;
    float linearAttenuation;
    float quadraticAttenuation;
};

uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
```

Кроме свойств источников света доступна информация об используемой модели освещения и вспомогательных величинах, удобных для расчета освещенности (структуры `gl_LightModelParameters`, `gl_LightModelProducts` и `gl_LightProducts`):

```
struct gl_LightModelParameters
{
    vec4 ambient;
};

uniform gl_LightModelParameters gl_LightModel;

struct gl_LightModelProducts
{
    vec4 sceneColor; // Derived. Ecm + Acm * Acs
};

uniform gl_LightModelProducts gl_FrontLightModelProduct;
uniform gl_LightModelProducts gl_BackLightModelProduct;

struct gl_LightProducts
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
};

uniform gl_LightProducts gl_FrontLightProduct [gl_MaxLights];
uniform gl_LightProducts gl_BackLightProduct [gl_MaxLights];

Еще один тип доступной информации — параметры автоматического вычисления текстурных координат и цвета для каждого текстурного блока:
uniform vec4 gl_TextureEnvColor [gl_MaxTextureImageUnits];
uniform vec4 gl_EyePlaneS      [gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneT      [gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneR      [gl_MaxTextureCoords];
uniform vec4 gl_EyePlaneQ      [gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneS   [gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneT   [gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneR   [gl_MaxTextureCoords];
uniform vec4 gl_ObjectPlaneQ   [gl_MaxTextureCoords];
```

Параметры, используемые для вычисления тумана, представлены при помощи структуры `gl_FogParameters`:

```
struct gl_FogParameters
{
    vec4 color;
    float density;
    float start;
    float end;
    float scale;      // Derived: 1.0 / (end - start)
};

uniform gl_FogParameters gl_Fog;
```

Стандартные *varying*-переменные

Помимо набора стандартных и доступных `uniform`-переменных GLSL предоставляет стандартный набор `varying`-переменных:

```
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;
varying vec4 gl_FrontSecondaryColor;
varying vec4 gl_BackSecondaryColor;
varying vec4 gl_TexCoord[]; // размер не более gl_MaxTextureCoords
varying float gl_FogFragCoord;
varying vec4 gl_Color;
varying vec4 gl_SecondaryColor;
varying vec4 gl_TexCoord[]; // размер не более gl_MaxTextureCoords
varying float gl_FogFragCoord;
```

Стандартные функции

Программам, написанным на языке GLSL (как вершинным, так и фрагментным), доступен целый ряд встроенных функций, которые можно разделить на следующие классы:

- функции, реализующие непосредственную работу с графическим ускорителем (например, доступ к текстуре);
- функции, реализующие обычные и довольно простые операции (например, `clamp`, `mix` и т. п.), которые хотя и очень просты, но очень часто используются и могут иметь аппаратную поддержку;
- функции, которые могут начиная с какого-то момента времени получить аппаратную поддержку.

Многие из них имеют аналоги в C, однако в качестве аргументов могут принимать не только скалярные значения, но и векторные.

В таблицах следующих подразделов тип *T* в определениях функций обозначает любой из типов *float*, *vec2*, *vec3* и *vec4*, а в качестве типа *M* может выступать любой из матричных типов *mat2*, *mat3* и *mat4*.

Тригонометрические функции и функции для работы с углами

Все тригонометрические функции работают с углами, выраженными в радианах.

Таблица 12.10. Тригонометрические функции и функции для работы с углами

Функция	Описание
<i>T radians</i> (<i>T deg</i>)	Перевод угла из градусов в радианы, возвращает $\frac{\pi \cdot \text{deg}}{180}$
<i>T degrees</i> (<i>T radians</i>)	Перевод угла из радиан в градусы, возвращает $\frac{180 \cdot \text{radians}}{\pi}$
<i>T sin</i> (<i>T angle</i>)	Вычисление синуса угла (заданного в радианах)
<i>T cos</i> (<i>T angle</i>)	Вычисление косинуса угла
<i>T tan</i> (<i>T angle</i>)	Вычисление тангенса угла
<i>T asin</i> (<i>T x</i>)	Вычисление арксинуса величины, результат находится в пределах $[-\pi/2, \pi/2]$; результат не определен для $ x > 1$
<i>T acos</i> (<i>T x</i>)	Вычисление арккосинуса величины, результат находится в пределах $[0, \pi]$; результат не определен для $ x > 1$
<i>T atan</i> (<i>T y, T x</i>)	Вычисление арктангенса <i>y / x</i> , знаки аргументов определяют, в каком квадранте лежит угол; результат находится в пределах $[-\pi, \pi]$
<i>T atan</i> (<i>T y_over_x</i>)	Вычисляет арктангенс величины <i>y_over_x</i> ; результат находится в пределах $[-\pi/2, \pi/2]$

Экспоненциальные функции (возвведение в степень, нахождение логарифмов)

Все эти экспоненциальные функции (табл. 12.11) выполняются покомпонентно.

Таблица 12.11. Экспоненциальные функции

Функция	Описание
<code>T pow (T x, T y)</code>	Вычисляет значение (x в степени y). Результат не определен, если $x < 0$ или $x = 0$ и $y \leq 0$
<code>T exp (T x)</code>	Вычисляет значение e^x
<code>T log (T x)</code>	Вычисляет значение $\ln x$. Результат не определен для $x \leq 0$
<code>T exp2 (T x)</code>	Вычисляет значение 2^x
<code>T log2 (T x)</code>	Вычисляет значение $\log_2 x$. Результат не определен для $x \leq 0$
<code>T sqrt (T x)</code>	Вычисляет значение \sqrt{x} . Результат не определен для $x < 0$
<code>T inversesqrt (T x)</code>	Вычисляет значение $\frac{1}{\sqrt{x}}$. Результат не определен для $x \leq 0$

Функции общего назначения

Все эти функции общего назначения (табл. 12.12) являются покомпонентными.

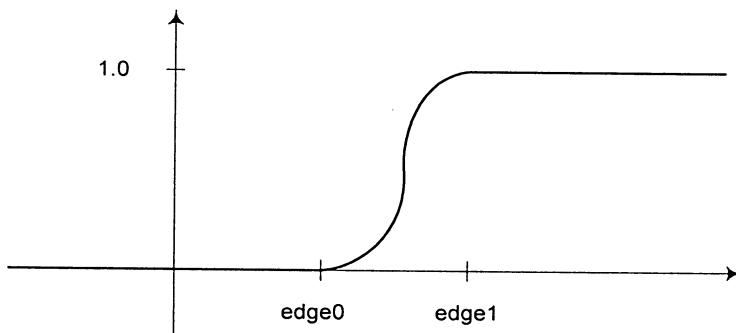
Таблица 12.12. Функции общего назначения

Функция	Описание
<code>T abs (T x)</code>	Вычисляет значение $ x $
<code>T sign (T x)</code>	Возвращает значение: -1, если $x < 0$; 0, если $x = 0$; 1, если $x > 0$
<code>T floor (T x)</code>	Возвращает наименьшее целое, не превосходящее x
<code>T ceil (T x)</code>	Возвращает наименьшее целое, большее или равное x
<code>T fract (Type x)</code>	Возвращает дробную часть x (результат равен значению $x - \text{floor}(x)$)
<code>T mod (T x, float y)</code>	Возвращает значение $x - y * \text{floor}(x/y)$

Таблица 12.12 (окончание)

Функция	Описание
T mod (T x, T y)	Возвращает $x - y * \text{floor}(x/y)$
T min (T x, T y)	Возвращает наименьшую из двух величин
T min (T x, float y)	
T max (T x, T y)	Возвращает наибольшую из двух величин
T max (T x, float y)	
T clamp (T x, T minValue, T maxValue)	Возвращает значение $\min(\max(x, minValue), maxValue)$.
T clamp (T x, float minValue, float maxValue)	Фактически первый аргумент отсекается по отрезку [minValue, maxValue]
T mix (T x, T y, Te a)	Возвращает $x*(1-a)+y*a$
T mix (T x, T y, float a)	
T step (T edge, T x)	Возвращает значение 0.0, если $x < edge$, в противном случае возвращает значение 1.0
T step (float edge, T x)	
T smoothstep (T edge0, T edge1, T x)	Возвращает значение 0.0, если $x \leq edge0$.
T smoothstep (float edge0, float edge1, T x)	Возвращает значение 1.0, если $x \geq edge1$. На интервале [edge0, edge1] используется интерполяция Эрмита.
	Функция описывается следующим фрагментом кода:
	<pre>T t = clamp ((x-edge0) / (edge1 - edge0), 0.0, 1.0); return t*t*(3-2*t);</pre>

Последняя функция `smoothstep` использует интерполяцию Эрмита для гладкого перехода от значения 0.0 к 1.0. На рис. 12.6 приведен ее график.

Рис. 12.6. График функции `smoothstep`

Геометрические функции

Все геометрические функции (табл. 12.13) работают непосредственно с векторами, т. е. не являются покомпонентными.

Таблица 12.13. Геометрические функции

Функция	Описание
float length (T x)	Возвращает длину вектора x
float distance (T p0, T p1)	Возвращает расстояние между точками p0 и p1, т. е. значение, равное length (p0 - p1)
float dot (T x, T y)	Возвращает скалярное произведение аргументов
vec3 cross (vec3 x, vec3 y)	Возвращает векторное произведение аргументов
T normalize (T x)	Возвращает единичный вектор, направленный в ту же сторону, что и вектор x
vec4 ftransform ()	Может использоваться только в вершинном шейдере. Возвращает преобразованную текущую вершину таким же образом, как и стандартный конвейер OpenGL
T faceforward (T N, T I, T Nref)	Возвращает значение N, если dot (I, Nref) < 0. Иначе возвращается значение -N
T reflect (T I, T N)	Возвращает отражение вектора I для нормали в точке N, т. е. $I - 2 * \text{dot}(I, N) * N$. Вектор N должен быть единичным
T refract (T I, T N, float eta)	Возвращает преломлённый вектор I, где N — вектор нормали, а eta — относительный коэффициент преломления. Действие функции описывает следующий фрагмент кода:
	<pre> k = 1.0 - eta*eta*(1.0 - dot(N, I) * dot(N, I)) if (k < 0.0) return T (0.0); else return eta*I-(eta*dot (N, I) + sqrt(k)) * N </pre>

Функция `ftransform` обычно применяется в том случае, когда используется рендеринг в несколько проходов, причем часть из них осуществляется через стандартный конвейер. Использование этой функции позволяет вершинному шейдеру возвращать то же самое значение для переменной `gl_Position`, что и в остальных проходах.

Матричные функции

Язык GLSL предоставляет одну стандартную матричную функцию — `matrixCompMult` (табл. 12.14).

Таблица 12.14. Матричные функции

Функция	Описание
<code>M_matrixCompMult (M x, M y)</code>	Возвращает поэлементное произведение двух матриц

Функции для сравнения векторов

Функции сравнения векторов предназначены для поэлементного сравнения векторов, поскольку стандартные операторы сравнения (`<`, `>`, `<=`, `>=`, `==`, `!=`) работают только со скалярными аргументами.

В табл. 12.15 параметр `vec` обозначает любой вектор с вещественными компонентами (`vec2`, `vec3` и `vec4`), параметр `ivec` — любой вектор с целочисленными компонентами (`ivec2`, `ivec3` и `ivec4`), параметр `bvec` — любой вектор логических значений (`bvec2`, `bvec3` или `bvec4`).

Таблица 12.15. Функции для сравнения векторов

Функция	Описание
<code>bvec lessThan (vec x, vec y)</code>	Возвращает результат покомпонентного сравнения операндов: $x < y$
<code>bvec lessThan (ivec x, ivec y)</code>	Возвращает результат покомпонентного сравнения операндов: $x < y$
<code>bvec lessThanEqual (vec x, vec y)</code>	Возвращает результат покомпонентного сравнения операндов: $x \leq y$
<code>bvec lessThanEqual (ivec x, ivec y)</code>	Возвращает результат покомпонентного сравнения операндов: $x \leq y$
<code>bvec greaterThan (vec x, vec y)</code>	Выполняет покомпонентное сравнение операндов: $x > y$
<code>bvec greaterThan (ivec x, ivec y)</code>	Выполняет покомпонентное сравнение операндов: $x > y$
<code>bvec greaterThanEqual (vec x, vec y)</code>	Выполняет покомпонентное сравнение операндов: $x \geq y$
<code>bvec greaterThanEqual (ivec x, ivec y)</code>	Выполняет покомпонентное сравнение операндов: $x \geq y$

Таблица 12.15 (окончание)

Функция	Описание
bvec equal (vec x, vec y)	Выполняет покомпонентное сравнение операндов: $x == y$
bvec equal (ivec x, ivec y)	
bvec equal (bvec x, bvec y)	
bvec notEqual (vec x, vec y)	Выполняет покомпонентное сравнение операндов: $x != y$
bvec notEqual (ivec x, ivec y)	
bvec notEqual (bvec x, bvec y)	
bool any (bvec x)	Возвращает значение <code>true</code> , если хотя бы один компонент вектора x принимает значение <code>true</code>
bool all (bvec x)	Возвращает значение <code>true</code> , если все компоненты вектора x принимают значение <code>true</code>
bvec not (bvec x)	Выполняет покомпонентное логическое отрицание вектора x

Функции для доступа к текстурам

Все функции доступа к текстурам (табл. 12.16) могут вызываться как из фрагментных шейдеров, так и из вершинных. Считается, что все параметры текстуры (размер, формат и т. п.) определены средствами OpenGL.

Через `bias` обозначен необязательный параметр, доступный только фрагментным шейдерам. Если он задан, то при вычислении уровня детализации (Level of detail, LOD) он добавляется к вычисленному значению перед обращением к текстуре. Если его нет, то обращение к текстуре производится стандартным образом.

Все функции, в именах которых есть суффикс `Lod`, можно использовать только в вершинных шейдерах, при этом параметр `lod` задает уровень детализации.

Если в имени функции присутствует суффикс `Proj`, то перед обращением значения текстурных координат делятся на значение последнего компонента.

Таблица 12.16. Функции для доступа к текстурам

Функции	Описание
<code>vec4 texture1D (sampler1D sampler, float coord [, float bias])</code>	Используют параметр <code>coord</code> для доступа к одномерной текстуре, задаваемой параметром <code>sampler</code> . В <code>Proj</code> -версии для индексации текстуры используется величина <code>coord.s/coord.t</code>
<code>vec4 texture1DProj (sampler1D sampler, vec2 coord [, float bias])</code>	
<code>vec4 texture1DProj (sampler1D sampler, vec4 coord [, float bias])</code>	
<code>vec4 texture1DLod (sampler1D sampler, float coord, float lod)</code>	
<code>vec4 texture1DProjLod (sampler1D sampler, vec2 coord, float lod)</code>	
<code>vec4 texture1DProjLod (sampler1D sampler, vec4 coord, float lod)</code>	
<code>vec4 texture2D (sampler2D sampler, vec2 coord [, float bias])</code>	Используют параметр <code>coord</code> для доступа к двухмерной текстуре, задаваемой параметром <code>sampler</code> . В <code>Proj</code> -версии для индексации текстуры используется величина <code>(coord.s/coord.p, coord.t/coord.p)</code>
<code>vec4 texture2DProj (sampler2D sampler, vec3 coord [, float bias])</code>	
<code>vec4 texture2DProj (sampler2D sampler, vec4 coord [, float bias])</code>	
<code>vec4 texture2DLod (sampler2D sampler, vec2 coord, float lod)</code>	
<code>vec4 texture2DProjLod (sampler2D sampler, vec3 coord, float lod)</code>	
<code>vec4 texture2DProjLod (sampler2D sampler, vec4 coord, float lod)</code>	
<code>vec4 texture3D (sampler3D sampler, vec3 coord [, float bias])</code>	Используют параметр <code>coord</code> для доступа к трехмерной текстуре, задаваемой параметром <code>sampler</code> . В <code>Proj</code> -версии для индексации текстуры используется величина <code>(coord.s/coord.q, coord.t/coord.q, coord.p/coord.q)</code>
<code>vec4 texture3DProj (sampler3D sampler, vec4 coord [, float bias])</code>	
<code>vec4 texture3DLod (sampler3D sampler, vec3 coord, float lod)</code>	
<code>vec4 texture3DProjLod (sampler3D sampler, vec4 coord, float lod)</code>	
<code>vec4 textureCube (samplerCube sampler, vec3 coord [, float bias])</code>	Используют параметр <code>coord</code> для доступа к кубической текстурной карте, задаваемой параметром <code>sampler</code>
<code>vec4 textureCubeLod (samplerCube sampler, vec3 coord, float lod)</code>	

Таблица 12.16 (окончание)

Функции	Описание
<code>vec4 shadow1D (sampler1DShadow sampler, vec3 coord [, float bias])</code>	
<code>vec4 shadow2D (sampler2DShadow sampler, vec3 coord [, float bias])</code>	
<code>vec4 shadow1DProj (</code> <code>sampler1Dshadow sampler, vec4 coord [, float bias])</code>	
<code>vec4 shadow2DProj (</code> <code>sampler2Dshadow sampler, vec4 coord [, float bias])</code>	Используют параметр <code>coord</code> для доступа к одномерной или двухмерной текстуре, содержащей значения глубины, задаваемой параметром <code>sampler</code> . В <code>Proj</code> -версии перед обращением к текстуре значение параметра <code>coord</code> делится на последний компонент вектора текстурных координат
<code>vec4 shadow1DLod (</code> <code>sampler1DShadow sampler, vec3 coord, float lod)</code>	
<code>vec4 shadow2DLod (</code> <code>sampler2DShadow sampler, vec3 coord, float lod)</code>	
<code>vec4 shadow1DProjLod (</code> <code>sampler1Dshadow sampler, vec4 coord, float lod)</code>	
<code>vec4 shadow2DProjLod (</code> <code>sampler2Dshadow sampler, vec4 coord, float lod)</code>	

Функции для работы с производными

Функции для работы с производными (табл. 12.17) позволяют приближенно вычислять значения производных от указанных величин. Однако реализация может давать не очень точные значения.

Справедливо приближенное равенство $F(x + h) - F(x) \approx dFdx(x) \cdot h$.

Таблица 12.17. Функции для работы с производными

Функция	Описание
<code>T dFdx (T p)</code>	Возвращает приближенное значение первой производной периметра <code>p</code> по <code>x</code>
<code>T dFdy (T p)</code>	Возвращает приближенное значение первой производной периметра <code>p</code> по <code>y</code>
<code>T fwidth (T p)</code>	Возвращает значение <code>abs (dFdx (p)) + abs (dFdy (p))</code>

Шумовые функции

Шумовые функции (табл. 12.18) доступны как вершинным, так и фрагментным шейдерам. Все эти функции обладают следующими свойствами:

- значения принимаются на отрезке $[-1, 1]$;
- средним значением является 0;
- одному и тому же значению аргумента всегда соответствует одно и то же значение функции (т. е. это не генератор случайных чисел);
- статистическая инвариантность относительно поворотов;
- статистическая инвариантность относительно переноса;
- C^1 -непрерывность.

На рис. 12.7 приведено изображение, генерируемое двухмерной шумовой функцией (при этом значения из промежутка $[-1, 1]$ переводятся в оттенки серого цвета).

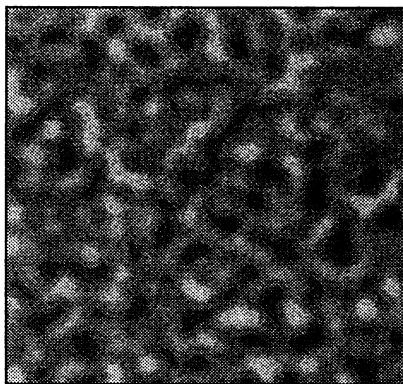


Рис. 12.7. Изображение, даваемое шумовой функцией

Таблица 12.18. Шумовые функции

Функция	Описание
<code>float noise1 (T x)</code>	Возвращает значение одномерной шумовой функции, соответствующей параметру x
<code>vec2 noise2 (T x)</code>	Возвращает значение двухмерной шумовой функции, соответствующей параметру x
<code>vec3 noise3 (T x)</code>	Возвращает значение трехмерной шумовой функции, соответствующей параметру x
<code>vec4 noise4 (T x)</code>	Возвращает значение четырехмерной шумовой функции, соответствующей параметру x

Обратите внимание: встроенная поддержка шумовых функций встречается сейчас довольно редко, поэтому в дальнейшем вместо шумовой функции будут использоваться текстуры, подобные приведенной на рис. 12.7.

Простейший пример использования вершинных и фрагментных шейдеров

В листингах 12.2 и 12.3 приведены простейшие вершинный и фрагментный шейдеры на языке GLSL.

Минимальной функциональностью, которую обязан предоставлять вершинный шейдер, является запись координат преобразованной вершины в переменную `gl_Position`.

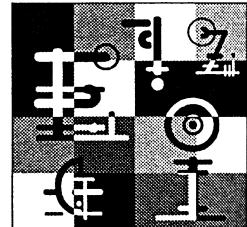
Листинг 12.2. Простейший вершинный шейдер

```
//  
// Simplest GLSL vertex shader  
//  
  
void main (void)  
{  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

Минимальной функциональностью, которой должен обладать фрагментный шейдер, является запись цвета в переменную `gl_FragColor`.

Листинг 12.3. Простейший фрагментный шейдер

```
//  
// Simplest fragment shader  
//  
  
void main (void)  
{  
    gl_FragColor = vec4 ( 0.0, 1.0, 0.0, 1.0 );  
}
```



Глава 13

Простейшие вершинные и фрагментные шейдеры. Реализация основных моделей освещения

Рассмотрим, как можно использовать язык GLSL (OpenGL Shading Language) для реализации ряда шейдеров, включая основные модели освещения.

Простейшим вершинным шейдером будет просто преобразование координат вершины при помощи произведения модельной и проектирующей матриц (листинг 13.1).

Листинг 13.1. Простейший вершинный шейдер

```
//  
// Simplest GLSL vertex shader  
  
  
void main (void)  
{  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

Простейший фрагментный шейдер оказывается еще проще — он просто присваивает всем фрагментам одно и то же значение цвета (листинг 13.2).

Листинг 13.2. Простейший фрагментный шейдер

```
//  
// Simplest fragment shader  
  
//
```

```
void main (void)
{
    gl_FragColor = vec4 ( 0.0, 1.0, 0.0, 1.0 );
}
```

Изображение объекта "чайник", построенное при помощи этих шейдеров, приведено на рис. 13.1. Соответствующий код на C++ будет рассмотрен в главе 14 (см. листинг 14.2).

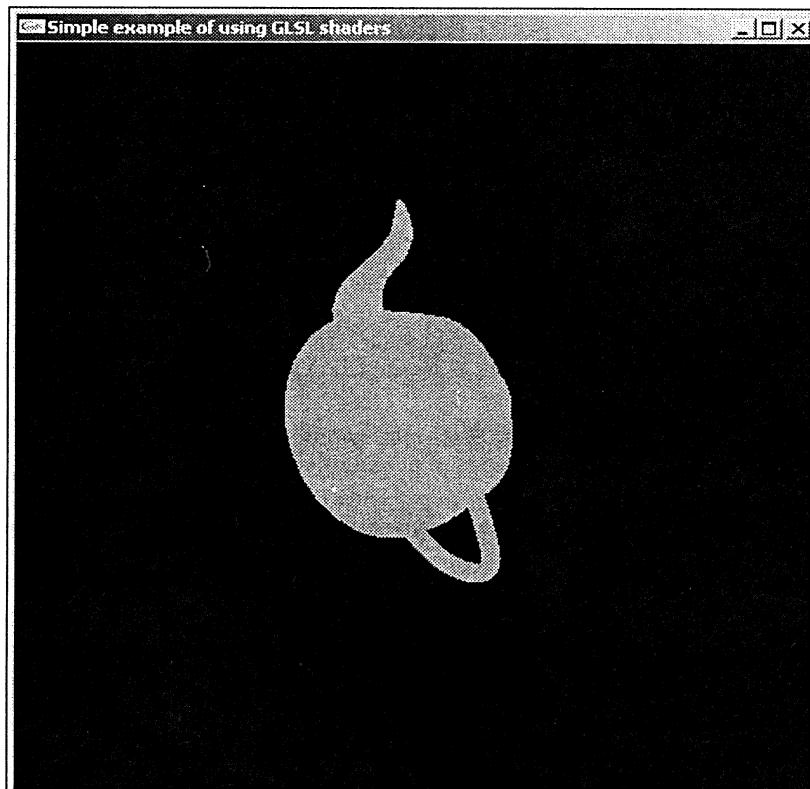


Рис. 13.1. Изображение чайника, построенное при помощи простейших шейдеров

Диффузная модель освещения

Рассмотрим теперь реализацию на GLSL простейшей диффузной модели освещения. В этой модели свет, падающий в произвольную точку P поверхности освещаемого объекта, считается равномерно рассеивающимся по всем направлениям полупространства, содержащего источник света (рис. 13.2).

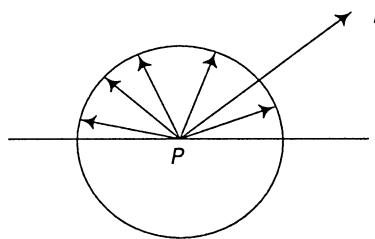


Рис. 13.2. Диффузная модель освещения

Поскольку свет рассеивается равномерно, видимая освещенность точки не зависит от положения наблюдателя, а определяется только ориентацией освещаемого участка поверхности, т. к. именно это определяет удельную плотность световой энергии, падающей в данную точку.

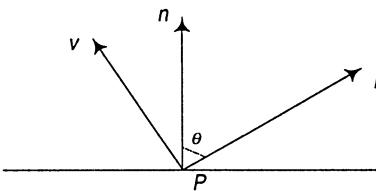


Рис. 13.3. Векторы, используемые для расчета освещенности

Обычно диффузная модель освещенности записывается в виде следующей формулы:

$$C_{out} = C \cdot (k_a + k_d \cdot \max(0, (n, l))). \quad (13.1)$$

В этой формуле параметром C обозначен цвет поверхности в точке, коэффициенты k_a и k_d задают вклад диффузного и фонового освещения (наличие последнего гарантирует, что объект всегда будет хоть как-то освещен, даже если на него не падает свет ни от одного источника света). Параметрами n и l в формуле обозначены единичные векторы нормали к поверхности и на источник света.

Если мы хотим использовать формулу (13.1) для расчета освещенности в каждом пикселе, то необходимо для каждой точки поверхности знать единичные векторы n и l . Простейший способ сделать это — найти в вершинном шейдере правильные векторы n и l для каждой вершины (вектор нормали необходимо преобразовать при помощи модельной матрицы) и передать их в фрагментный шейдер посредством `varying`-переменных.

В листингах 13.3 и 13.4 представлены вершинный и фрагментный шейдеры для диффузного освещения.

Листинг 13.3. Вершинный шейдер для диффузного освещения

```
varying    vec3 l;
varying    vec3 n;

uniform   vec4  lightPos;

void main(void)
{
    // transformed point to world space
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );

    // vector to light source
    l = normalize ( vec3 ( lightPos ) - p );
    n = normalize ( gl_NormalMatrix * gl_Normal ); // transformed n

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Через uniform-переменную `lightPos` в вершинный шейдер передаются координаты источника света. Тогда во фрагментном шейдере мы для каждого фрагмента получаем значения векторов `n` и `l`, полученных путем интерполяции значений, найденных в вершинном шейдере.

Поскольку в процессе интерполяции эти векторы могли перестать быть единичными, то перед использованием в формуле (13.1) их следует нормировать.

Листинг 13.4. Фрагментный шейдер для диффузного освещения

```
varying    vec3 l;
varying    vec3 n;

void main (void)
{
    const vec4  diffColor = vec4 ( 0.5, 0.0, 0.0, 1.0 );
    const float ka        = 0.2;
    const float kd        = 1.0;

    vec3 n2    = normalize ( n );
```

```

vec3 12    = normalize ( 1 );
vec4 diff = diffColor * (ka + kd*max ( dot ( n2, 12 ), 0.0 ) );

gl_FragColor = diff;
}

```

Бликовое освещение по Блинну

Рассмотрим расширение диффузной модели освещения (13.1) для включения в нее бликов на поверхности освещаемых объектов.

Одной из наиболее распространенных моделей для получения бликов является модель Блинна. В ней освещенность задается следующей формулой:

$$C_{out} = C \cdot (k_a + k_d \cdot \max(0, (n, l)) + I \cdot k_s \cdot \max(0, (n, h)^p)). \quad (13.2)$$

В этой формуле параметрами k_s и I обозначены вклад бликового освещения и цвет источника света.

Параметром h обозначен единичный вектор — бисектор направления на источник света l и направления на наблюдателя v (рис. 13.4).

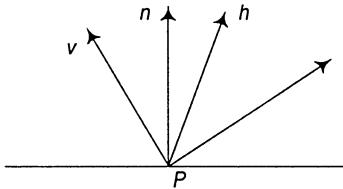


Рис. 13.4. Векторы, используемые в модели освещения Блинна

Параметр p отвечает за контрастность блика — чем он больше, тем более контрастным (и меньшего размера) получается блик.

Для нахождения вектора h можно, при условии нормированности векторов l и v , использовать формулу (13.3).

$$h = \frac{l + v}{\|l + v\|}. \quad (13.3)$$

Рассмотрим, как следует модифицировать шейдеры из листингов 13.3 и 13.4 для учета модели освещения Блинна (листинги 13.5 и 13.6).

Аналогично случаю диффузного освещения в каждой вершине вычисляются векторы n , l , v и h и посредством varying-переменных передаются в фрагментную программу.

Примечание

Перед вычислением вектора h по формуле (13.3) векторы l и v следует нормировать.

Через uniform-переменную `eyePos` в шейдер передается положение наблюдателя.

Листинг 13.5. Вершинный шейдер для модели освещения Блінна

```
//  
// Blinn vertex shader  
  
//  
  
varying vec3 l;  
varying vec3 h;  
varying vec3 n;  
varying vec3 v;  
  
uniform vec4 lightPos;  
uniform vec4 eyePos;  
  
void main(void)  
{  
    // transformed point to world space  
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );  
  
    // vector to light source  
    l = normalize ( vec3 ( lightPos ) - p );  
  
    // vector to the eye  
    v = normalize ( vec3 ( eyePos ) - p );  
    h = normalize ( l + v );  
  
    // transformed n  
    n = normalize ( gl_NormalMatrix * gl_Normal );  
  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

Задачей фрагментного шейдера, как и ранее, является нормирование всех используемых векторов (n , l , v и h) и вычисление освещенности по формуле (13.2).

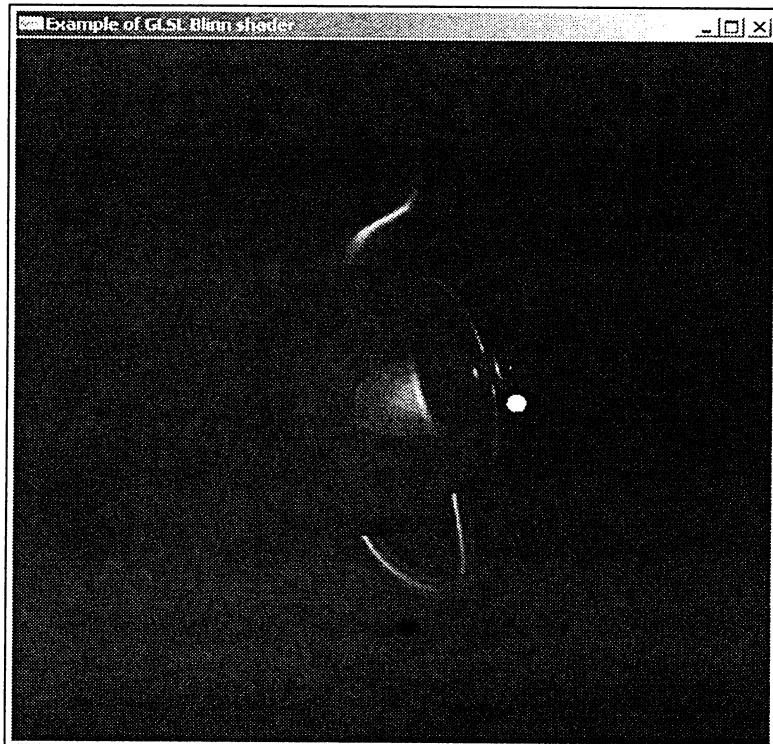


Рис. 13.5. Чайник, освещенный "по Блинну"

Листинг 13.6. Фрагментный шейдер для модели освещения Блинна

```
//  
// Blinn fragment shader  
  
//  
  
varying vec3 l;  
varying vec3 h;  
varying vec3 v;  
varying vec3 n;  
  
void main (void)  
{
```

```

const vec4 diffColor = vec4 ( 0.5, 0.0, 0.0, 1.0 );
const vec4 specColor = vec4 ( 0.7, 0.7, 0.0, 1.0 );
const float specPower = 30;

vec3 n2    = normalize ( n );
vec3 l2    = normalize ( l );
vec3 h2    = normalize ( h );
vec4 diff = diffColor * max ( dot ( n2, l2 ), 0.0 );
vec4 spec = specColor * pow ( max ( dot ( n2, h2 ), 0.0 ),
                             specPower );

gl_FragColor = diff + spec;
}

```

На рис. 13.5 приведено изображение чайника, построенное с использованием шейдеров из листингов 13.5 и 13.6.

Бликовое освещение по Фонгу

В модели Фонга бликового освещения используется другая (хотя ее формула и похожа на формулу (13.2)) модель бликов:

$$C_{out} = C \cdot (k_a + k_d \cdot \max(0, (n, l)) + I \cdot k_s \cdot \max(0, (l, r))^p). \quad (13.4)$$

В этой формуле параметр r обозначает отраженный относительно вектора n вектор на наблюдателя v (рис. 13.6).

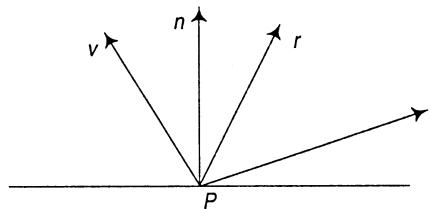


Рис. 13.6. Используемые в модели Фонга векторы

По аналогии с моделью освещения Блинна в вершинном шейдере вычисляются векторы l , v и n , которые через varying-переменные передаются фрагментному шейдеру.

В листингах 13.7 и 13.8 приведены вершинный и фрагментный шейдеры для модели освещения Фонга.

Листинг 13.7. Вершинный шейдер для модели освещения Фонга

```

//  

// Phong vertex shader  

//  
  

varying vec3 l;  

varying vec3 v;  

varying vec3 n;  
  

uniform vec4 lightPos;  

uniform vec4 eyePos;  
  

void main(void)  
{
    // transformed point to world space
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );  
  

    // vector to light source
    l = normalize ( vec3 ( lightPos ) - p );  
  

    // vector to the eye
    v = normalize ( vec3 ( eyePos ) - p );  

```

Фрагментный шейдер для модели Фонга (листинг 13.8) полностью аналогичен шейдеру из листинга 13.6 — в нем выполняется нормирование векторов, вычисление отраженного вектора с использованием встроенной функции `reflect`, после чего освещенность находится по формуле (13.4).

Листинг 13.8. Фрагментный шейдер для модели освещения Фонга

```

//  

// Phong fragment shader  

//  


```

```
varying vec3 l;
varying vec3 v;
varying vec3 n;

void main (void)
{
    const vec4 diffColor = vec4 ( 0.5, 0.0, 0.0, 1.0 );
    const vec4 specColor = vec4 ( 0.7, 0.7, 0.0, 1.0 );
    const float specPower = 30;

    vec3 n2 = normalize ( n );
    vec3 l2 = normalize ( l );
    vec3 v2 = normalize ( v );
    vec3 r = reflect ( -v2, n2 );
    vec4 diff = diffColor * max ( dot ( n2, l2 ), 0.0 );
    vec4 spec = specColor * pow ( max ( dot ( l2, r ), 0.0 ),
                                  specPower );

    gl_FragColor = diff + spec;
}
```

Можно слегка модифицировать шейдеры из листингов 13.7 и 13.8 для использования в них текстуры — тогда цвет поверхности *C* уже не будет постоянной величиной, а будет получаться обращением к текстуре (листинги 13.9 и 13.10).

Листинг 13.9. Вершинный шейдер для освещения Фонга с текстурированием

```
//  
// Phong vertex shader with texturing  
  
varying vec3 l;  
varying vec3 v;  
varying vec3 n;  
  
uniform vec4 lightPos;  
uniform vec4 eyePos;  
  
void main(void)
```

```

{
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );

    l = normalize ( vec3 ( lightPos ) - p );
    v = normalize ( vec3 ( eyePos ) - p );
    n = normalize ( gl_NormalMatrix * gl_Normal );

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord [0] = gl_MultiTexCoord0;
}

```

Листинг 13.10. Фрагментный шейдер для освещения Фонга с текстурированием

```

//  

// Phong fragment shader with texturing  

//  

varying vec3 l;  

varying vec3 v;  

varying vec3 n;  

uniform sampler2D decalMap;  

void main (void)  

{  

    const vec4 specColor = vec4 ( 0.7, 0.7, 0.0, 1.0 );  

    const float specPower = 30;  

    vec4 diffColor = texture2D ( decalMap, gl_TexCoord [0].xy );  

    vec3 n2 = normalize ( n );  

    vec3 l2 = normalize ( l );  

    vec3 v2 = normalize ( v );  

    vec3 r = reflect ( -v2, n2 );  

    vec4 diff = diffColor * max ( dot ( n2, l2 ), 0.0 );  

    vec4 spec = specColor * pow ( max ( dot ( l2, r ), 0.0 ),  

                                specPower );  

    gl_FragColor = diff + spec;  

}

```

Обратите внимание: в вершинном шейдере через переменную `g1_TexCoord[0]` передаются текстурные координаты для фрагментного шейдера. Без этого присваивания в фрагментном шейдере у всех фрагментов было бы одно и то же (нулевое) значение текстурных координат.

Для доступа к текстуре по текстурным координатам в фрагментном шейдере используется стандартная функция `texture2D`; сама текстура передается как переменная типа `uniform sampler2D`.

Использование карт нормалей

До сих пор мы считали, что вектор нормали в каждой точке получается путем интерполяции значений нормали в вершинах. В результате при рендеринге получаются изображения гладких объектов. Однако часто появляется желание добавить поверхности определенную структуру (микрорельеф), сделать ее не такой гладкой. Самый простой способ достичь этого — использовать так называемые карты нормалей [1, 18].

Под *картой нормалей* (*bump map*) подразумевается текстура, содержащая значения единичных векторов нормалей в виде цветов. Но поскольку при чтении из текстуры в шейдере получаются векторы, все компоненты которых находятся в пределах $[0, 1]$, а компоненты единичного вектора находятся в пределах $[-1, 1]$, необходим способ преобразования векторов в цвета и наоборот.

$$\begin{aligned} r &= 0,5 \cdot (x + 1), \\ g &= 0,5 \cdot (y + 1), \\ b &= 0,5 \cdot (z + 1). \end{aligned} \tag{13.5}$$

Если мы хотим по цвету из текстуры получить исходный вектор нормали, то используются формулы (13.6).

$$\begin{aligned} x &= 2 \cdot r - 1, \\ y &= 2 \cdot g - 1, \\ z &= 2 \cdot b - 1. \end{aligned} \tag{13.6}$$

Наложив такую карту на грань, мы для каждой точки грани получим некоторый единичный вектор, интерпретируемый как вектор нормали в данной точке. Однако непосредственное использование значений нормалей из карт затрудняет то, что они обычно задают нормали относительно данной грани, т. е. неискаженный вектор нормали равен $(0, 0, 1)$. Поэтому для работы с ними вводится понятие *касательного пространства* (рис. 13.7).

В качестве базиса этого пространства выступают три единичных вектора — t , b и n . Вектор t , лежащий в плоскости грани, называется *касательным*, вектор b , также лежащий в плоскости грани и перпендикулярный вектору t , называется *бинормалью*.

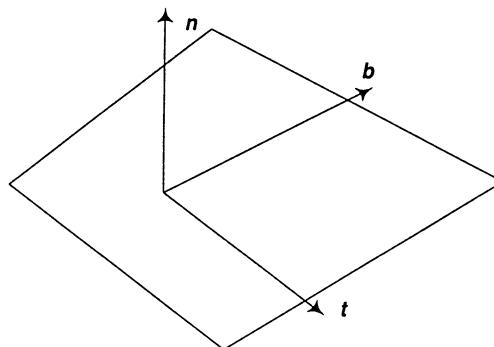


Рис. 13.7. Базис касательного пространства

Тройка векторов *t*, *b* и *n* образует ортонормированный базис пространства, поэтому процедура перевода в это пространство крайне проста.

Если нужно вектор *p* перевести в эту систему координат, для этого достаточно найти три скалярных произведения — координатами вектора *p* будут $((t, p), (b, p), (n, p))$.

Как видно, в координатах касательного пространства грани вектор нормали *n* всегда равен $(0, 0, 1)$. Таким образом, в этом пространстве можно свободно использовать карты нормалей. Только необходимо также перевести в касательное пространство все используемые векторы.

Проще всего этот перевод осуществить в вершинном шейдере. Для этого нужно для каждой вершины задавать не только положение и вектор нормали, но и касательную и бинормаль. Чаще всего их задают как текстурные координаты для текстурных блоков 1 и 2.

Тогда задачей вершинного шейдера является не только нахождение векторов *v*, *l*, *h* и *n*, но и перевод их в систему координат касательного пространства. Для этого векторы *n*, *t* и *b* необходимо сначала умножить на `gl_NormalMatrix` для перевода в мировое пространство, после чего использовать эти векторы для перевода векторов, необходимых для вычисления освещенности.

Примечание

Поскольку векторы *n*, *t* и *b* образуют ортонормированный базис, перевод в касательное пространство не изменяет ни длин векторов, ни углов между ними.

В листингах 13.11 и 13.12 приведены вершинный и фрагментный шейдеры, реализующие освещение по модели Блинна с использованием карты нормалей.

Листинг 13.11. Вершинный шейдер для освещения по модели Блинна с использованием карты нормалей

```
//  
// Simple bump vertex shader  
  
varying vec3 lt;  
varying vec3 ht;  
  
uniform vec4 lightPos;  
uniform vec4 eyePos;  
  
void main(void)  
{  
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );  
    vec3 l = normalize ( vec3 ( lightPos ) - p );  
    vec3 v = normalize ( vec3 ( eyePos ) - p );  
    vec3 h = normalize ( l + v );  
    vec3 n = gl_NormalMatrix * gl_Normal;  
  
    vec3 t = gl_NormalMatrix * gl_MultiTexCoord1.xyz;  
    vec3 b = gl_NormalMatrix * gl_MultiTexCoord2.xyz;  
  
    // now remap l, and h into tangent space  
    lt = vec3 ( dot ( l, t ), dot ( l, b ), dot ( l, n ) );  
    ht = vec3 ( dot ( h, t ), dot ( h, b ), dot ( h, n ) );  
  
    gl_Position      = gl_ModelViewProjectionMatrix * gl_Vertex;  
    gl_TexCoord [0] = gl_MultiTexCoord0;  
}
```

Листинг 13.12. Фрагментный шейдер для освещения по модели Блинна с использованием карты нормалей

```
//  
// Simple bump fragment shader  
  
varying vec3 lt;  
varying vec3 ht;  
  
uniform sampler2D bumpMap;
```

```
uniform sampler2D diffuseMap;

void main (void)
{
    const vec4 specColor = vec4 ( 1 );

    // get normal perturbation
    vec3 n      = texture2D ( bumpMap, gl_TexCoord [0] ).rgb;
    vec3 nt     = normalize ( 2.0*n - 1.0 );
    vec3 l2     = normalize ( lt );
    vec3 h2     = normalize ( ht );
    float diff = max ( dot ( nt, l2 ), 0.0 ) + 0.2;
    float spec = pow ( max ( dot ( nt, h2 ), 0.0 ), 30 );

    gl_FragColor = diff * texture2D ( diffuseMap, gl_TexCoord [0] ) +
                    spec * specColor;
}
```

На рис. 13.8 приводится изображение тора, полученное при помощи этих шейдеров.

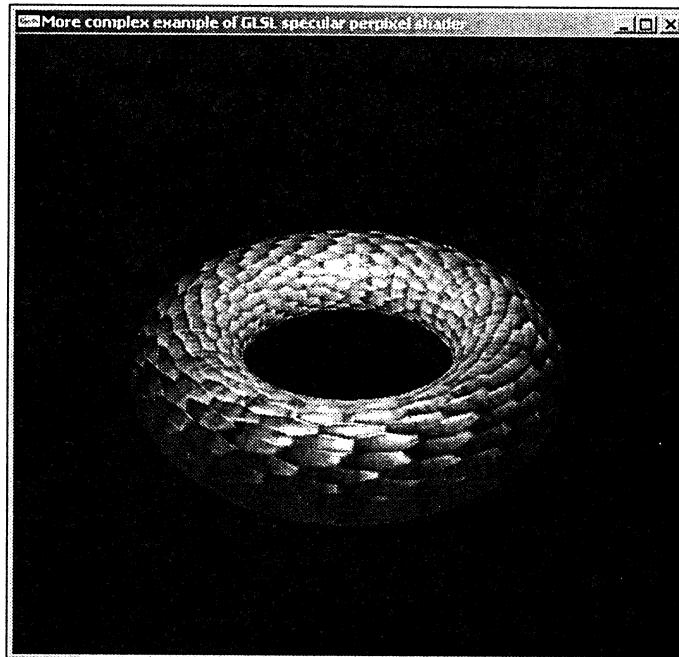


Рис. 13.8. Тор с наложенной картой нормалей

На рис. 13.9 приведена использованная карта нормалей.

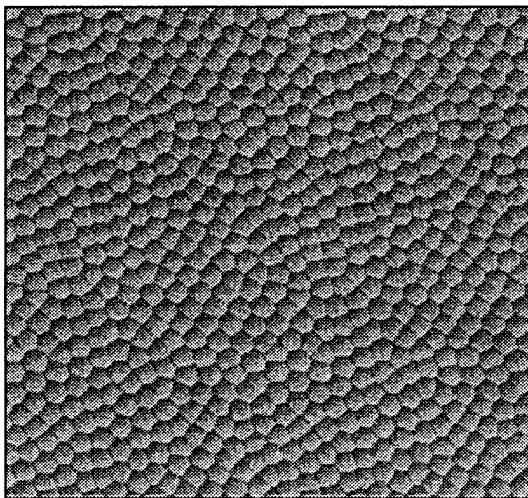


Рис. 13.9. Карта нормалей для рис. 13.8

Анизотропные модели освещения

Все рассмотренные ранее модели освещения обладают следующим свойством — освещенность поверхности определяется только взаимным расположением векторов l , v и n . При этом ориентация поверхности (при условии сохранения взаимного расположения этих векторов) никакой роли не играет. Поверхности, обладающие таким свойством, называются *изотропными*.

Однако на практике встречаются и поверхности, не обладающие этим свойством (например, поверхность компакт-диска или полированного металла) — они называются *анизотропными*.

Для того чтобы можно было учитывать ориентацию поверхности, на ней обычно вводится *поле касательных векторов*, т. е. каждой точке P сопоставляется единичный вектор t , перпендикулярный вектору нормали n (рис. 13.10).

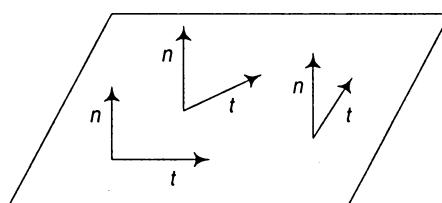


Рис. 13.10. Поле касательных векторов

Поле касательных векторов обычно вводится при помощи текстуры, называемой *карты касательных* (*tangent map*). Примеры карт касательных вы можете найти в каталоге *Textures\TangentMaps* сопроводительного компакт-диска книги.

Существует несколько моделей освещения для анизотропных поверхностей, далее мы рассмотрим две из них.

Один из простейших подходов заключается в трактовании поверхности как состоящей из множества бесконечно тонких нитей (волокон) [1, 18]. Тогда касательный вектор — это касательная к такой нити, а бинормаль — нормаль к ней (лежащая в плоскости).

Рассмотрим, каким образом можно адаптировать модель освещения Блинна для расчета освещенности таких волосков.

Пусть мы хотим найти освещенность бесконечно тонкой нити (рис. 13.11).

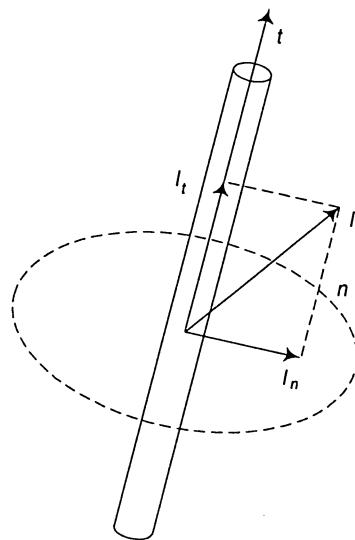


Рис. 13.11. Освещение бесконечно тонкой нити

Для нее невозможно однозначно определить вектор нормали n — любой единичный вектор, перпендикулярный вектору t , может выступать в качестве вектора нормали.

Стандартный подход заключается в том, что отдельно для каждого из компонентов освещения (диффузного и бликового) выбирается тот вектор нормали, для которого достигается максимальное значение соответствующего компонента.

Рассмотрим это на примере нахождения диффузного компонента. Заметим, что единичный вектор на источник света l можно разложить на две части — па-

раллельную вектору t и перпендикулярную ему. Тем самым вектор l можно представить в следующем виде:

$$l = (t, l)l + l^\perp. \quad (13.7)$$

В формуле (13.7) параметром l^\perp обозначен компонент вектора l , перпендикулярный вектору t .

Несложно убедиться, что максимум (n, l) будет достигаться на векторе $n = l^\perp / \|l^\perp\|$. Выразив l^\perp из (13.7) и возведя эту величину скалярно в квадрат, легко найти и длину этого компонента и сам искомый максимум (13.8).

$$\max(l, n) = \sqrt{1 - (t, l)^2}. \quad (13.8)$$

По аналогии с (13.8) легко найти и бликовый компонент освещенности для модели Блинна:

$$\max(h, n) = \sqrt{1 - (t, h)^2}. \quad (13.9)$$

Комбинируя формулы (13.2), (13.8) и (13.9), получаем следующее выражения для освещенности анизотропной поверхности:

$$C_{out} = C \cdot (k_a + k_d \cdot \max(0, (t, l))^{0.5} + I \cdot k_s \cdot \max(0, (t, h))^{0.5p}). \quad (13.10)$$

Несмотря на то что формулу (13.10) легко непосредственно реализовать при помощи фрагментного шейдера, можно произвести небольшую оптимизацию.

Легко заметить, что диффузная освещенность зависит от $(t, l)^2$, а бликовая — от $(t, h)^2$. Эту зависимость можно занести в двухмерную текстуру — в качестве координаты s выступает $(t, l)^2$, а в качестве координаты t — $(t, h)^2$. Диффузный компонент, соответствующий данной паре квадратов скалярных произведений, можно хранить в красном компоненте текстуры, а бликовый — в зеленом.

На сопроводительном компакт-диске книги в каталоге Code\scripts находится скрипт на языке Python, предназначенный для построения подобной текстуры, которой мы в дальнейшем будем свободно пользоваться.

Применяя этот подход, получаем вершинный и фрагментный шейдеры для анизотропной модели освещения, приведенные в листингах 13.13 и 13.14.

Листинг 13.13. Вершинный шейдер для рендеринга анизотропной поверхности

```
//  
// Simple anisotropic lighting vertex shader  
// Using lookup table for diffuse and specular coefficients  
//
```

```

varying vec3 lt;
varying vec3 ht;

uniform vec4 lightPos;
uniform vec4 eyePos;

void main(void)
{
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );
    vec3 l = normalize ( vec3 ( lightPos ) - p );
    vec3 v = normalize ( vec3 ( eyePos ) - p );
    vec3 h = normalize ( l + v );
    vec3 n = gl_NormalMatrix * gl_Normal;
    vec3 t = gl_NormalMatrix * gl_MultiTexCoord1.xyz;
    vec3 b = gl_NormalMatrix * gl_MultiTexCoord2.xyz;

    now remap l, and h into tangent space
    lt = vec3 ( dot ( l, t ), dot ( l, b ), dot ( l, n ) );
    ht = vec3 ( dot ( h, t ), dot ( h, b ), dot ( h, n ) );

    gl_Position      = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord [0] = gl_MultiTexCoord0;
}

```

Листинг 13.14. Фрагментный шейдер для рендеринга анизотропной поверхности

```

// 
// Simple anisotropic lighting fragment shader
// Using lookup table for diffuse and specular coefficients
//

varying vec3 lt;
varying vec3 ht;

uniform sampler2D tangentMap;
uniform sampler2D decalMap;
uniform sampler2D anisoTable;

void main (void)

```

```

{
    const vec4 specColor = vec4 ( 0, 0, 1, 0 );

    vec3 tang = normalize ( 2.0 * texture2D ( tangentMap,
                                              gl_TexCoord [0].xyz - 1.0 );
    float dot1 = dot ( normalize ( lt ), tang );
    float dot2 = dot ( normalize ( ht ), tang );
    vec2 arg = vec2 ( dot1, dot2 );
    vec2 ds = texture2D ( anisoTable, arg*arg ).rg;
    vec4 color = texture2D ( decalMap, gl_TexCoord [0].xy );

    gl_FragColor = color * ds.x + specColor * ds.y;
    gl_FragColor.a = 1;
}

```

Кроме модели (13.8) есть и другие. Одну из них — модель Уорда (Ward) мы рассмотрим далее.

В упрощенной версии этой модели освещение задается следующей формулой:

$$C_{out} = C \cdot k_a + I \cdot k_s \cdot e^{-\left(\frac{k(h,t)}{k(h,n)}\right)^2}. \quad (13.11)$$

В этой формуле параметр k определяет контрастность бликов (аналогично параметру p в моделях Блинна и Фонга).

В листингах 13.15 и 13.16 приведены вершинный и фрагментный шейдеры для модели Уорда.

Листинг 13.15. Вершинный шейдер для модели Уорда

```

// Simple anisotropic Ward lighting vertex shader
//
varying vec3 lt;
varying vec3 vt;
varying vec3 ht;

uniform vec4 lightPos;
uniform vec4 eyePos;

void main(void)

```

```

{
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );
    vec3 l = normalize ( vec3 ( lightPos ) - p );
    vec3 v = normalize ( vec3 ( eyePos ) - p );
    vec3 h = normalize ( l + v );
    vec3 n = gl_NormalMatrix * gl_Normal;
    vec3 t = gl_NormalMatrix * gl_MultiTexCoord1.xyz;
    vec3 b = gl_NormalMatrix * gl_MultiTexCoord2.xyz;

        // now remap v, l, and h into tangent space
    vt = vec3 ( dot ( v, t ), dot ( v, b ), dot ( v, n ) );
    lt = vec3 ( dot ( l, t ), dot ( l, b ), dot ( l, n ) );
    ht = vec3 ( dot ( h, t ), dot ( h, b ), dot ( h, n ) );

    gl_Position      = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord [0] = gl_MultiTexCoord0;
}

```

Листинг 13.16. Фрагментный шейдер для модели Уорда

```

//
// Simple anisotropic Ward lighting fragment shader
//
varying vec3 vt;
varying vec3 lt;
varying vec3 ht;

uniform sampler2D tangentMap;
uniform sampler2D decalMap;

void main (void)
{
    const vec4 specColor = vec4 ( 0, 0, 1, 0 );
    const vec3 n = vec3 ( 0, 0, 1 );
    const float roughness = 5;

    vec4 color = texture2D ( decalMap, gl_TexCoord [0].xy );
    vec3 tang = normalize ( 2.0 * texture2D ( tangentMap,
                                              gl_TexCoord [0] ).xyz - vec3 ( 1.0 ) );

```

```

float dot1 = dot ( ht, tang ) * roughness;
float dot2 = dot ( ht, n );
float p = dot1 / dot2;

gl_FragColor.rgb = color.rgb + specColor.rgb * exp ( -p*p );
gl_FragColor.a   = 1;
}

```

На рис. 13.12 приведено изображение тора, полученное с использованием модели освещения Уорда.

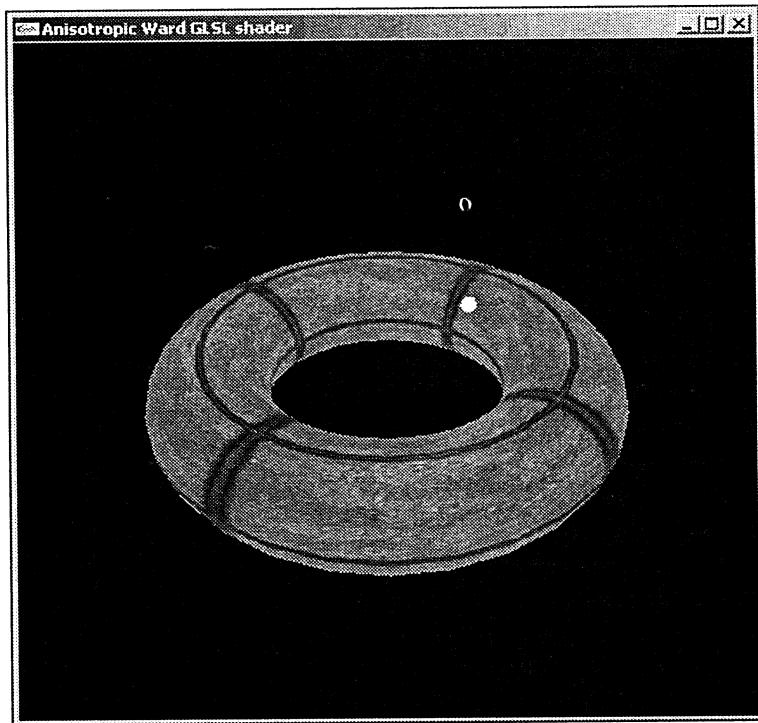


Рис. 13.12. Тор, освещенный по модели Уорда

Подсветка края

Можно слегка модифицировать модель освещения Блинна для выделения контурных линий (края) освещаемого объекта (rim lighting) (рис. 13.13).

Для подсветки края к уравнению освещенности (13.2) добавляется член вида:

$$C_{\text{rim}} = C_r \cdot (1 - \max(0, (n, v)))^p. \quad (13.12)$$

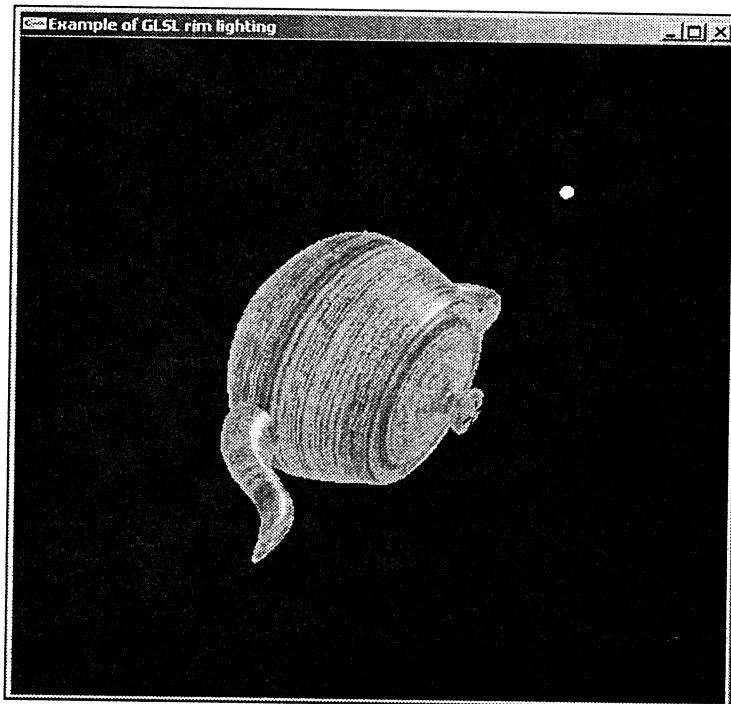


Рис. 13.13. Объект с подсветкой края

Параметром C_r обозначается цвет, которым осуществляется подсветка, параметром p — ее контрастность.

Легко заметить, что своего максимума это выражение достигает, когда векторы n и v являются перпендикулярными, т. е. действительно на контурных линиях.

Можно слегка изменить формулу (13.12), введя в нее параметр ($bias$), позволяющий регулировать размер подсвечиваемой области:

$$C_{\text{rim}} = C_r \cdot (1 + bias - \max(0, (n, v)))^p. \quad (13.15)$$

Для того чтобы добавить к модели Блинна подсветку краев, нам даже не понадобится изменять вершинный шейдер — все изменения происходят лишь в фрагментном шейдере, приводимом в листинге 13.17.

Листинг 13.17. Фрагментный шейдер для рендеринга с подсветкой края

```
//  
// Rim lighting fragment shader  
//  
varying vec3 l;
```

```
varying vec3 h;
varying vec3 v;
varying vec3 n;

uniform sampler2D decalMap;

void main (void)
{
    const vec4 diffColor = vec4 ( 0.5, 0.0, 0.0, 1.0 );
    const vec4 specColor = vec4 ( 0.7, 0.7, 0.0, 1.0 );
    const float rimPower = 10;
    const float specPower = 30;
    const float bias = 0.2;

    vec3 n2 = normalize ( n );
    vec3 l2 = normalize ( l );
    vec3 h2 = normalize ( h );
    vec3 v2 = normalize ( v );
    vec4 diff = diffColor * max ( dot ( n2, l2 ), 0.0 );
    vec4 spec = specColor * pow ( max ( dot ( n2, h2 ), 0.0 ),
                                  specPower );
    float rim = pow ( 1.0 + bias - max ( dot ( n2, v2 ), 0.0 ),
                      rimPower );

    gl_FragColor = texture2D ( decalMap, gl_TexCoord [0].xy ) +
                   rim * vec4 ( 0, 0, 1, 1 ) + spec * specColor;
}
```

Модель освещения Гуч

Одной из очень простых и в то же время дающих красивые визуальные результаты моделей освещения является *модель Гуч* (Amy Gooch).

Эта модель использует косинус угла между векторами n и l для перехода от одного цвета (обычно более теплого) к другому (обычно более холодному). При этом когда источник освещения оказывается за объектом (т. е. $(n, l) < 0$), возникает ощущение, что свет проходит сквозь объект и за счет этого приобретает другой цвет.

В листингах 13.18 и 13.19 приведены вершинный и фрагментный шейдеры для модели освещения Гуч.

Листинг 13.18. Вершинный шейдер для модели Гуч

```

//  

// Vertex shader for Gooch shading  

// Based on 3Dlabs code by Randi Rost  

//  

uniform vec4 lightPos;  

uniform vec4 eyePos;  

varying float NdotL;  

varying vec3 reflectVec;  

varying vec3 viewVec;  

void main(void)  

{  

    vec3 p          = vec3      ( gl_ModelViewMatrix * gl_Vertex );  

    vec3 norm       = normalize ( gl_NormalMatrix * gl_Normal );  

    vec3 lightVec  = normalize ( lightPos.xyz - p.xyz );  

    reflectVec     = normalize ( reflect ( -lightVec, norm ) );  

    viewVec        = normalize ( eyePos.xyz - p.xyz );  

    NdotL          = ( dot ( lightVec, norm ) + 1.0 ) * 0.5;  

    gl_Position    = ftransform();  

}

```

Листинг 13.19. Фрагментный шейдер для модели Гуч

```

//  

// Fragment shader for Gooch shading  

// Based on 3Dlabs code by Randi Rost  

//  

varying float NdotL;  

varying vec3 reflectVec;  

varying vec3 viewVec;  

void main (void)  

{

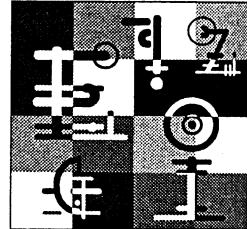
```

```
const vec3 surfaceColor = vec3 ( 0.75, 0.75, 0.75 );
const vec3 warmColor     = vec3 ( 0.6, 0.6, 0.0 );
const vec3 coolColor     = vec3 ( 0.0, 0.0, 0.6 );
const float diffuseWarm  = 0.45;
const float diffuseCool   = 0.45;

vec3 kCool      = min (coolColor + diffuseCool * surfaceColor, 1.0);
vec3 kWarm      = min (warmColor + diffuseWarm * surfaceColor, 1.0);
vec3 kFinal     = mix (kCool, kWarm, NdotL);

vec3 r      = normalize ( reflectVec );
vec3 v      = normalize ( viewVec );
float spec = pow ( max ( dot ( r, v ), 0.0 ), 32 );

gl_FragColor = vec4 ( min ( kFinal + spec, 1.0 ), 1.0 );
}
```



Глава 14

Практическое использование GLSL в программах на C++, необходимые расширения

Эта глава посвящена использованию GLSL в OpenGL и необходимым для этого расширениям, а также инкапсуляции программ на GLSL в класс C++.

Необходимые расширения

Для работы с GLSL требуется поддержка расширений `GL_ARB_shader_objects`, `GL_ARB_shading_language_100`, `GL_ARB_vertex_shader` и `GL_ARB_fragment_shader`.

В следующих разделах даны краткие сведения об этих расширениях, необходимые для работы с GLSL.

Расширение `GL_ARB_shading_language_100`

Данное расширение просто сообщает о поддержке языка GLSL версии 1.00. Более точно узнать номер поддерживаемой версии GLSL можно при помощи следующего фрагмента кода:

```
const char * slVer = (const char *) glGetString(  
                           GL_SHADING_LANGUAGE_VERSION_ARB );  
  
if ( glGetError() != GL_NO_ERROR )  
    printf ( "Shading language supported: 1.051\n" );  
else  
    printf ( "Shading language supported: %s\n", slVer );
```

Возвращаемая строка имеет вид "`major.minor[.release][vendor info]`". Если при попытке получить номер версии происходит ошибка, то считаем, что поддерживается первая версия, имеющая номер 1.051 (по номеру версии описания).

Расширение GL_ARB_shader_objects

Данное расширение вводит необходимый API для поддержки шейдеров. При этом разделяется понятие шейдера (вершинного или фрагментного) и программы, включающей в себя эти шейдеры.

Непосредственную поддержку вершинных и фрагментных шейдеров вводят расширения `GL_ARB_vertex_shader` и `GL_ARB_fragment_shader`, рассматриваемые далее.

Как сама программа, так и составляющие ее шейдеры представляются соответствующими объектами внутри OpenGL (`program object` и `shader object`). Однако в отличие от рассмотренных ранее вершинных и фрагментных программ, для идентификации этих объектов вводится новый тип — `GLhandleARB` (в OpenGL 2.0 используется тип `GLuint`). Этот тип идентифицирует внутренний объект OpenGL; соответствующие объекты могут содержать в себе какие-то данные, их можно создавать, изменять, уничтожать. Нулевому значению не может соответствовать никакой объект, поэтому если возвращенный идентификатор равен нулю, то это признак ошибки создания соответствующего объекта.

Некоторые из этих объектов могут выступать как контейнеры, т. е. содержать в себе другие объекты. Типичным примером такого объекта-контейнера служит объект "программа".

Для создания и уничтожения соответствующих объектов предназначены функции `glCreateProgramObjectARB`, `glCreateProgramObjectARB`, `glCreateShaderObjectARB` и `glDeleteObjectARB`:

```
GLhandleARB glCreateProgramObjectARB ();
GLhandleARB glCreateShaderObjectARB (GLenum shaderType);
void glDeleteObjectARB (GLhandleARB object);
```

Функция `glCreateProgramObjectARB` создает объект "программа" и возвращает его идентификатор.

Функция `glCreateShaderObjectARB` служит для создания объектов-шейдеров. Параметр `shaderType` задает требуемый тип шейдера и может пока принимать одно из двух значений — `GL_VERTEX_SHADER_ARB` или `GL_FRAGMENT_SHADER_ARB`.

Функция `glDeleteObjectARB` предназначена для удаления объекта `object` (как объекта-программы, так и объекта-шейдера). При этом соответствующий объект либо немедленно удаляется, либо помечается для удаления, если непосредственно в данный момент он содержится в другом объекте.

Следующие две функции служат для помещения одного объекта в объект-контейнер и "отсоединения" объекта от контейнера, в который он помещен:

```
void glAttachObjectARB (GLhandleARB containerObj, GLhandleARB obj);
```

```
void glDetachObjectARB (GLhandleARB containerObj,
                       GLhandleARB attachedObj);
```

Функция `glAttachObjectARB` помещает (присоединяет — attaches) объект `obj` в объект-контейнер `containerObj`. В случае, если `containerObj` не является объектом-контейнером или подсоединение невозможно, возникает ошибка. Также ошибкой является попытка повторного помещения объекта в контейнер (т. е. когда он уже присутствует в этом контейнере).

Функция `glDetachObjectARB` удаляет объект `attachedObj` из контейнера `containerObj`. Если убираемый объект не содержится больше ни в одном контейнере, то он помечается для удаления.

Для загрузки исходного кода шейдера в соответствующий объект предназначена функция `glShaderSourceARB`:

```
void glShaderSourceARB (GLhandleARB shaderObj, GLsizei count,
                       const GLcharARB ** strings,
                       const GLint * lengths);
```

Исходный код для шейдера, задаваемого параметром `shaderObj`, передается как массив строк. Параметр `count` передает количество исходных строк, параметр `strings` является указателем на массив указателей на строки. Параметр `lengths` является указателем на массив, задающий для каждой строки ее длину. В случае, когда `lengths = NULL`, полагается, что каждая строка завершена нулевым байтом '\0'. Вполне допустимым является передача всего текста в виде одной большой строки (что мы и будем делать).

После того как исходный код для шейдера загружен, его необходимо (в отличие от ранее рассмотренных вершинных и фрагментных программ) транслировать. Для этого предназначена функция `glCompileShaderARB`:

```
void glCompileShaderARB (GLhandleARB shaderObj);
```

Вызов данной функции приводит к трансляции ранее загруженного исходного текста шейдера на языке GLSL.

Узнать результат трансляции программы можно при помощи следующего фрагмента кода:

```
GLint compileStatus;
glGetObjectParameterivARB (shader, GL_OBJECT_COMPILE_STATUS_ARB,
                           &compileStatus);
```

В случае, если программа оттранслирована успешно, значение переменной `compileStatus` будет равно `GL_TRUE`.

После того как все шейдеры объекта-программы были успешно оттранслированы, необходим еще один шаг, после которого программа будет готова

к использованию, — компиляция (связывание шейдеров и окружения между собой, линковка). Для этого предназначена функция `glLinkProgramARB`:

```
void glLinkProgramARB (GLhandleARB programObj);
```

Здесь параметр `programObj` идентифицирует объект-программу. Для проверки результатов компиляции можно использовать следующий фрагмент кода:

```
GLint linked;  
glGetObjectParameterivARB (program, GL_OBJECT_LINK_STATUS_ARB,  
                           &linked);
```

Если после выполнения этого кода значение переменной `linked` равно `GL_FALSE`, значит, компиляция завершилась неудачно.

В случае успешной компиляции вызов функции `glUseProgramObjectARB` делает заданную программу активной:

```
void glUseProgramObjectARB (GLhandleARB programObj);
```

Если значение параметра `programObj` равно нулю, то этот запрос отключает шейдеры и приводит к использованию стандартного конвейера рендеринга OpenGL.

Даже когда программа является текущей, по-прежнему можно загружать и транслировать исходные тексты шейдеров, присоединять и отсоединять отдельные шейдеры. Любые такие изменения вступают в силу только после того, как программа будет снова собрана (откомпилирована).

Может оказаться, что несмотря на успешную линковку программа не может быть выполнена. Проверить корректность программы можно при помощи следующего фрагмента кода:

```
GLint validated;  
glValidateProgramARB (programObj);  
glGetObjectParameterivARB (program, GL_OBJECT_VALIDATE_STATUS_ARB,  
                           &validated);
```

Значение переменной `validated`, равное `GL_TRUE`, говорит о том, что программа может быть успешно выполнена.

Получить идентификатор активной в данный момент программы можно при помощи функции `glGetHandleARB`:

```
GLhandleARB glGetHandleARB (GLenum pname);
```

В качестве значения параметра `pname` должна выступать константа `GL_PROGRAM_OBJECT_ARB`.

С каждым из объектов-шейдеров и объектов-программ связан свой внутренний лог (*log*) — текстовый буфер, куда заносится информация о компиляции и линковке. Для получения этого лога необходимо сначала узнать его

размер, выделить буфер необходимого объема, после чего скопировать содержимое лога в этот буфер:

```

int          logLength      = 0;
int          charsWritten   = 0;
GLcharARB * infoLog;

glGetObjectParameterivARB ( object, GL_OBJECT_INFO_LOG_LENGTH_ARB,
                           &logLength );

if ( glGetError () != GL_NO_ERROR )
    return;

if ( logLength < 1 )
    return;

infoLog = (GLcharARB*) malloc ( logLength );

if ( infoLog == NULL )
    return;
glGetInfoLogARB ( object, logLength, &charsWritten, infoLog );

```

В общем случае функция `glGetInfoLogARB` описывается так:

```
void glGetInfoLogARB (GLhandleARB obj, GLsizei maxLength,
                      GLsizei * length, GLcharARB * infoLog);
```

Параметр `maxLength` задает максимальное количество байт, которое может быть записано в буфер, адрес которого содержится в параметре `infoLog`. Параметр `length` содержит адрес переменной, в которую будет записано количество скопированных в буфер байт.

Также данное расширение предоставляет ряд функций, служащих для работы с `uniform`-переменными.

Несмотря на то что внутри шейдеров все `uniform`-переменные имеют имена и работа с ними осуществляется по имени, функции для работы с `uniform`-переменными вместо имени переменной используют ее целочисленный индекс.

Для получения индекса (`location`) по имени (`name`) переменной для данной программы предназначена функция `glGetUniformLocationARB`:

```
int glGetUniformLocationARB (GLhandleARB programObj,
                           const GLcharARB * name);
```

Данная функция возвращает индекс соответствующей `uniform`-переменной или значение `-1` в случае ошибки.

Также эту функцию можно использовать для получения индексов отдельных элементов массивов (добавляя к имени массива индекс в квадратных скобках, например, `myArray[1]`) и для получения индексов к отдельным полям структуры (добавляя к имени структуры точку и имя поля, например: `light1.intensity`).

После получения индекса (`location`) следующие функции позволяют присваивать значения отдельным *uniform*-переменным заданного типа:

```
void glUniform{1234}fARB (int location, T value);  
void glUniform{1234}iARB (int location, T value);
```

При помощи следующих функций можно присваивать значения массивам вещественных и целых чисел (параметр `count` задает количество значений, которые следует записать, а параметр `value` задает адрес, начиная с которого идут значения для записи):

```
void glUniform{1234}fvARB (int location, GLsizei count, T value);  
void glUniform{1234}ivARB (int location, GLsizei count, T value);
```

Для записи значений отдельных *uniform*-матриц служит следующий класс функций:

```
void glUniformMatrix{234}fvARB (int location, GLsizei count,  
                                GLboolean transpose, T value);
```

Значение параметра `count` определяет количество матриц, которые необходимо записать (1 — запись одной матрицы, >1 — запись массива матриц), параметр `transpose` позволяет транспонировать матрицу (матрицы) перед записью. Последний параметр является указателем на блок памяти, содержащий необходимые значения.

Следующие две функции служат для получения значений вещественных и целочисленных *uniform*-переменных. Количество реально записанных значений определяется типом соответствующей переменной:

```
void glGetUniformfvARB (GLhandleARB programObj, int location,  
                        float * params);  
void glGetUniformivARB (GLhandleARB programObj, int location,  
                        int * params);
```

Для получения информации об активных *uniform*-переменных предназначена функция `glGetActiveUniformARB`:

```
void glGetActiveUniformARB (GLhandleARB programObj, GLuint index,  
                           GLsizei maxLength, GLsizei * length,  
                           GLint * size, GLenum * type,  
                           GLcharARB * name);
```

Параметр `index` задает номер `uniform`-переменной, о которой запрашивается информация (0 соответствует первой переменной); в буфер, задаваемый параметром `name`, будет записано имя переменной; количество записанных в этот буфер символов будет записано по адресу, задаваемому переменной `length`. Имя будет записано в виде строки, завершенной символом '\0'. Параметр `maxLength` задает максимальное количество символов, которые можно записать в буфер `name`. В переменную, адрес которой содержится в параметре `type`, будет записан тип соответствующей переменной. Возможными значениями для типа являются `GL_FLOAT`, `GL_FLOAT_VEC2_ARB`, `GL_FLOAT_VEC3_ARB`, `GL_FLOAT_VEC4_ARB`, `INT`, `GL_INT_VEC2_ARB`, `GL_INT_VEC3_ARB`, `GL_INT_VEC4_ARB`, `GL_BOOL_ARB`, `GL_BOOL_VEC2_ARB`, `GL_BOOL_VEC3_ARB`, `GL_BOOL_VEC4_ARB`, `GL_FLOAT_MAT2_ARB`, `GL_FLOAT_MAT3_ARB`, `GL_FLOAT_MAT4_ARB`, `GL_SAMPLER_1D_ARB`, `GL_SAMPLER_2D_ARB`, `GL_SAMPLER_3D_ARB`, `GL_SAMPLER_CUBE_ARB`, `GL_SAMPLER_1D_SHADOW_ARB`, `GL_SAMPLER_2D_SHADOW_ARB`, `GL_SAMPLER_2D_RECT_ARB` и `GL_SAMPLER_2D_RECT_SHADOW_ARB`. В переменную, адрес которой содержится в параметре `size`, записывается количество элементов соответствующего типа.

Для `uniform`-переменных типа `sampler` записываемым значением служит номер текстурного блока, в котором выбрана данная текстура (это значение считается целочисленным).

Расширение GL_ARB_vertex_shader

Расширение `GL_ARB_vertex_shader` вводит целый ряд функций и констант для поддержки вершинных шейдеров.

Вот их полный список:

```
void glVertexAttrib1fARB (GLuint index, float v0);
void glVertexAttrib1sARB (GLuint index, short v0);
void glVertexAttrib1dARB (GLuint index, double v0);
void glVertexAttrib2fARB (GLuint index, float v0, float v1);
void glVertexAttrib2sARB (GLuint index, short v0, short v1);
void glVertexAttrib2dARB (GLuint index, double v0,double v1);
void glVertexAttrib3fARB (GLuint index, float v0, float v1, float v2);
void glVertexAttrib3sARB (GLuint index, short v0, short v1, short v2);
void glVertexAttrib3dARB (GLuint index, double v0, double v1,
                        double v2);
void glVertexAttrib4fARB (GLuint index, float v0, float v1, float v2,
                        float v3);
void glVertexAttrib4sARB (GLuint index, short v0, short v1, short v2,
                        short v3);
```

```
void glVertexAttrib4dARB (GLuint index, double v0, double v1, double v2,
                           double v3);
void glVertexAttrib4NubARB (GLuint index, GLubyte x, GLubyte y,
                            GLubyte z, GLubyte w);
void glVertexAttrib1fvARB (GLuint index, const float * v);
void glVertexAttrib1svARB (GLuint index, const short * v);
void glVertexAttrib1dvARB (GLuint index, const double * v);
void glVertexAttrib2fvARB (GLuint index, const float * v);
void glVertexAttrib2svARB (GLuint index, const short * v);
void glVertexAttrib2dvARB (GLuint index, const double * v);
void glVertexAttrib3fvARB (GLuint index, const float * v);
void glVertexAttrib3svARB (GLuint index, const short * v);
void glVertexAttrib3dvARB (GLuint index, const double * v);
void glVertexAttrib4fvARB (GLuint index, const float * v);
void glVertexAttrib4svARB (GLuint index, const short * v);
void glVertexAttrib4dvARB (GLuint index, const double * v);
void glVertexAttrib4ivARB (GLuint index, const int * v);
void glVertexAttrib4bvARB (GLuint index, const byte * v);
void glVertexAttrib4ubvARB (GLuint index, const GLubyte * v);
void glVertexAttrib4usvARB (GLuint index, const GLushort * v);
void glVertexAttrib4uivARB (GLuint index, const GLuint * v);
void glVertexAttrib4NbvARB (GLuint index, const byte * v);
void glVertexAttrib4NsuvARB (GLuint index, const short * v);
void glVertexAttrib4NivARB (GLuint index, const int * v);
void glVertexAttrib4NubvARB (GLuint index, const GLubyte * v);
void glVertexAttrib4NusvARB (GLuint index, const GLushort * v);
void glVertexAttrib4NuivARB (GLuint index, const GLuint * v);
void glVertexAttribPointerARB (GLuint index, int size, GLenum type,
                               GLboolean normalized, GLsizei stride,
                               const void * pointer);
void glEnableVertexAttribArrayARB (GLuint index);
void glDisableVertexAttribArrayARB (GLuint index);
void glBindAttribLocationARB (GLhandleARB programObj, GLuint index,
                             const GLcharARB * name);
void glGetActiveAttribARB (GLhandleARB programObj, GLuint index,
                           GLsizei maxLength, GLsizei * length,
                           int * size, GLenum * type,
                           GLcharARB * name);
int glGetAttribLocationARB (handleARB programObj,
```

```

        const GLcharARB * name);
void glGetVertexAttribArraydvARB (GLuint index, GLenum pname,
                                double * params);
void glGetVertexAttribArrayfvARB (GLuint index, GLenum pname,
                                float * params);
void glGetVertexAttribArrayivARB (GLuint index, GLenum pname,
                                 int * params);
void glGetVertexAttribArrayPointervARB (GLuint index, GLenum pname,
                                         void ** pointer);

```

Функции `glVertexAttrib` осуществляют запись значения вершинного атрибута по его индексу. Для получения индекса атрибута по его имени служит функция `glGetAttribLocationARB`.

Чтение значений атрибутов осуществляется функциями `glGetVertexAttrib`.

Для получения информации об атрибуте по его индексу предназначена функция `glGetActiveAttribARB`:

```

void glGetActiveAttribARB (GLhandleARB programObj, GLuint index,
                           GLsizei maxLength, GLsizei * length,
                           int * size, GLenum * type,
                           GLcharARB * name );

```

Параметр `name` задает буфер, куда будет записано имя соответствующего атрибута (не более `maxLength` символов), количество записанных в буфер байт будет занесено в переменную по адресу `length`. Тип переменной будет записан в переменную, адрес которой содержится в параметре `type`. Возможными значениями типа являются `FLOAT`, `FLOAT_VEC2_ARB`, `GL_FLOAT_VEC3_ARB`, `G_FLOAT_VEC4_ARB`, `GL_FLOAT_MAT2_ARB`, `GL_FLOAT_MAT3_ARB` и `GL_FLOAT_MAT4_ARB`.

Существует также возможность явно задать значение индекса для атрибута. Для этого предназначена функция `glBindAttribLocationARB`. Для стандартных атрибутов (т. е. начинающиеся с префикса `gl_`) явное задание индекса не допускается.

Явное задание индекса вступает в силу только после того, как программа будет скомпилирована заново.

Расширение GL_ARB_fragment_shader

Расширение `GL_ARB_fragment_shader` добавляет необходимую поддержку для написания фрагментных шейдеров. Оно вводит всего несколько констант и не вводит никаких дополнительных функций.

Получение информации о поддержке GLSL

Приводимая в листинге 14.1 программа проверяет, поддерживается ли GLSL, и в случае его поддержки выводит на экран информацию об ограничениях для шейдеров.

Листинг 14.1. Программа, проверяющая поддержку GLSL

```
//  
// Sample to check for GLSL program support in OpenGL card and driver  
  
#include    "libExt.h"  
#include    <glut.h>  
#include    <stdio.h>  
#include    <stdlib.h>  
  
  
void init ()  
{  
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );  
    glEnable      ( GL_DEPTH_TEST );  
}  
  
void display ()  
{  
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );  
  
    glutSwapBuffers ();  
}  
  
void reshape ( int w, int h )  
{  
    glViewport      ( 0, 0, (GLsizei)w, (GLsizei)h );  
    glMatrixMode   ( GL_PROJECTION );  
    glLoadIdentity ();  
    glMatrixMode   ( GL_MODELVIEW );  
    glLoadIdentity ();  
}
```

```
void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )           // quit requested
        exit ( 0 );
}

int main ( int argc, char * argv [] )
{
    // initialize glut
    glutInit          ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize ( 400, 400 );

    // create window
    glutCreateWindow ( "OpenGL GLSL info" );

    // register handlers
    glutDisplayFunc   ( display );
    glutReshapeFunc   ( reshape );
    glutKeyboardFunc  ( key );

    init              ();
    initExtensions ();

    const char * vendor     = (const char *)glGetString ( GL_VENDOR      );
    const char * renderer   = (const char *)glGetString ( GL_RENDERER   );
    const char * version    = (const char *)glGetString ( GL_VERSION    );
    const char * extension = (const char *)glGetString ( GL_EXTENSIONS );

    printf ( "Vendor: %s\nRenderer: %s\nVersion: %s\n", vendor,
             renderer, version );

    if ( !isExtensionSupported ( "GL_ARB_shading_language_100" ) )
    {
        printf("GL_ARB_shading_language_100 extension NOT supported.\n");

        return 1;
    }
}
```

```
printf ( "GL_ARB_shading_language_100 extension is supported !\n" );

const char * slVer = (const char *) glGetString (
    GL_SHADING_LANGUAGE_VERSION_ARB );

if ( glGetError() != GL_NO_ERROR )
    printf ( "Shading language supported: 1.051\n" );
else
    printf ( "Shading language supported: %s\n", slVer );

int maxVertexAttribs;
int maxVertexTextureUnits;
int maxFragmentTextureUnits;
int maxCombinedTextureUnits;
int maxVertexUniformComponents;
int maxVaryingFloats;
int maxFragmentUniformComponents;
int maxTextureCoords;

glGetIntegerv ( GL_MAX_VERTEX_UNIFORM_COMPONENTS_ARB,
                &maxVertexUniformComponents );
glGetIntegerv ( GL_MAX_VERTEX_ATTRIBS_ARB,
                &maxVertexAttribs );
glGetIntegerv ( GL_MAX_TEXTURE_IMAGE_UNITS_ARB,
                &maxFragmentTextureUnits );
glGetIntegerv ( GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS_ARB,
                &maxVertexTextureUnits );
glGetIntegerv ( GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS_ARB,
                &maxCombinedTextureUnits );
glGetIntegerv ( GL_MAX_VARYING_FLOATS_ARB,
                &maxVaryingFloats );
glGetIntegerv ( GL_MAX_FRAGMENT_UNIFORM_COMPONENTS_ARB,
                &maxFragmentUniformComponents );
glGetIntegerv ( GL_MAX_TEXTURE_COORDS_ARB,
                &maxTextureCoords );

printf ( "\nGLSL program limits:\n" );
printf ( "\tmax vertex attributes           : %d\n",
        maxVertexAttribs );
```

```

printf ( "\tmax vertex texture units      : %d\n",
        maxVertexTextureUnits );
printf ( "\tmax fragment texture units     : %d\n",
        maxFragmentTextureUnits );
printf ( "\tmax combined texture units    : %d\n",
        maxCombinedTextureUnits );
printf ( "\tmax vertex uniform components : %d\n",
        maxVertexUniformComponents );
printf ( "\tmax varying floats           : %d\n",
        maxVaryingFloats );
printf ( "\tmax fragment uniform components : %d\n",
        maxFragmentUniformComponents );
printf ( "\tmax texture coords           : %d\n",
        maxTextureCoords );

return 0;
}

```

Данная программа определяет поддержку всех расширений, необходимых для работы с GLSL, выдает номер поддерживаемой версии GLSL и ряд ограничений на шейдеры.

Значение `maxVertexAttribs` определяет максимальное возможное количество атрибутов вершины.

Значение `maxVertexTextureUnits` определяет максимальное возможное количество текстурных блоков, доступных вершинному шейдеру.

Значение `maxFragmentTextureUnits` определяет максимальное возможное количество текстурных блоков, доступных фрагментному шейдеру.

Значение `maxCombinedTextureUnits` определяет максимальное возможное количество текстурных блоков, доступных и вершинному и фрагментному шейдерам.

Значение `maxVertexUniformComponents` определяет максимальное количество памяти (количество вещественных чисел), доступное вершинному шейдеру для создания `uniform`-переменных.

Значение `maxFragmentUniformComponents` определяет максимальное количество памяти (количество вещественных чисел), доступное фрагментному шейдеру для создания `uniform`-переменных.

Значение `maxVaryingFloats` определяет максимальный объем памяти (количество вещественных чисел), доступный для создания `varying`-переменных.

Значение `maxTextureCoords` определяет максимальное возможное количество текстурных координат, которые можно связать с вершиной.

Простейшая программа, использующая GLSL

Опираясь на приведенные описания и фрагменты кода, можно написать простейшую программу на C++, использующую GLSL-шейдеры. В листинге 14.2 приведен исходный код такой программы, причем использующей в качестве шейдеров простейшие шейдеры из листингов 13.1 и 13.2.

Листинг 14.2. Программа на C++, использующая шейдеры на GLSL

```
//  
// Sample showing how to use GLSL programs  
//  
  
#include    "libExt.h"  
  
#include    <glut.h>  
#include    <stdio.h>  
#include    <stdlib.h>  
  
#include    "libTexture.h"  
#include    "Vector3D.h"  
#include    "Vector2D.h"  
#include    "Data.h"  
  
Vector3D   eye    ( 7, 5, 7 );           // camera position  
Vector3D   light   ( 5, 0, 4 );          // light position  
float      angle = 0;  
Vector3D   rot    ( 0, 0, 0 );  
int        mouseOldX = 0;  
int        mouseOldY = 0;  
  
GLhandleARB program         = 0;           // program handles  
GLhandleARB vertexShader   = 0;  
GLhandleARB fragmentShader = 0;  
  
bool      checkOpenGLError ()  
{  
    GLenum glErr;
```

```
bool      retCode = true;

for ( ; ; )
{
    GLenum  glErr = glGetError ();

    if ( glErr == GL_NO_ERROR )
        return retCode;

    printf ( "glError: %s\n", gluErrorString ( glErr ) );

    retCode = false;
    glErr   = glGetError ();
}

return retCode;
}

//  

// Print out the information log for a shader object or a program object
//  

void printInfoLog ( GLhandleARB object )
{
    int      logLength     = 0;
    int      charsWritten  = 0;
    GLcharARB * infoLog;

    checkOpenGLError();           // Check for OpenGL errors

    glGetObjectParameterivARB ( object, GL_OBJECT_INFO_LOG_LENGTH_ARB,
                                &logLength );

    if ( !checkOpenGLError() )           // check for OpenGL errors
        exit ( 1 );

    if ( logLength > 0 )
    {
```

```
infoLog = (GLcharARB*) malloc ( logLength );

if ( infoLog == NULL )
{
    printf("ERROR: Could not allocate InfoLog buffer\n");

    exit ( 1 );
}

glGetInfoLogARB ( object, logLength, &charsWritten, infoLog );

printf ( "InfoLog:\n%s\n\n", infoLog );
free ( infoLog );
}

if ( !checkOpenGLError () )           // check for OpenGL errors
exit ( 1 );
}

bool loadShader ( GLhandleARB shader, const char * fileName )
{
    printf ( "Loading %s\n", fileName );

    Data data ( fileName );

    if ( !data.isOk () || data.isEmpty () )
        exit ( 1 );

    const char * body = (const char *) data.getPtr ( 0 );
    GLint compileStatus;

    glShaderSourceARB ( shader, 1, &body, NULL );

                    // compile the particle vertex
                    // shader, and print out
    glCompileShaderARB ( shader );

    if ( !checkOpenGLError() )           // check for OpenGL errors
```

```
    return false;

    glGetObjectParameterivARB ( shader, GL_OBJECT_COMPILE_STATUS_ARB,
                                &compileStatus);

    printInfoLog ( shader );

    return compileStatus != 0;
}

void init ()
{
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );
    glEnable      ( GL_DEPTH_TEST );
    glDepthFunc   ( GL_LEQUAL );

    glHint ( GL_POLYGON_SMOOTH_HINT, GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}

void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ( );

    glRotatef     ( rot.x, 1, 0, 0 );
    glRotatef     ( rot.y, 0, 1, 0 );
    glRotatef     ( rot.z, 0, 0, 1 );

    glutSolidTeapot ( 2 );

    glPopMatrix ();

    glutSwapBuffers ();
}

void reshape ( int w, int h )
```

```
{  
    glViewport      ( 0, 0, (GLsizei)w, (GLsizei)h );  
    glMatrixMode   ( GL_PROJECTION );  
    glLoadIdentity ();  
    gluPerspective ( 60.0, (GLfloat)w/(GLfloat)h, 1.0, 60.0 );  
    glMatrixMode   ( GL_MODELVIEW );  
    glLoadIdentity ();  
    gluLookAt      ( eye.x, eye.y, eye.z, // eye  
                    0, 0, 0,           // center  
                    0.0, 0.0, 1.0 ); // up  
}  
  
void motion ( int x, int y )  
{  
    rot.y -= ((mouseOldY - y) * 180.0f) / 200.0f;  
    rot.z -= ((mouseOldX - x) * 180.0f) / 200.0f;  
    rot.x = 0;  
  
    if ( rot.z > 360 )  
        rot.z -= 360;  
  
    if ( rot.z < -360 )  
        rot.z += 360;  
  
    if ( rot.y > 360 )  
        rot.y -= 360;  
  
    if ( rot.y < -360 )  
        rot.y += 360;  
  
    mouseOldX = x;  
    mouseOldY = y;  
  
    glutPostRedisplay ();  
}  
  
void mouse ( int button, int state, int x, int y )  
{  
    if ( state == GLUT_DOWN )
```

```
{  
    mouseOldX = x;  
    mouseOldY = y;  
}  
}  
  
void key ( unsigned char key, int x, int y )  
{  
    if ( key == 27 || key == 'q' || key == 'Q' )      // quit requested  
        exit ( 0 );  
}  
  
void     animate ()  
{  
    angle  = 0.004f * glutGet ( GLUT_ELAPSED_TIME );  
  
    glutPostRedisplay ();  
}  
  
int main ( int argc, char * argv [] )  
{  
    // initialize glut  
    glutInit          ( &argc, argv );  
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );  
    glutInitWindowSize ( 500, 500 );  
  
    // create window  
    glutCreateWindow ("Simple example of using GLSL shaders");  
  
    // register handlers  
    glutDisplayFunc   ( display );  
    glutReshapeFunc   ( reshape );  
    glutKeyboardFunc  ( key     );  
    glutMouseFunc     ( mouse   );  
    glutMotionFunc    ( motion  );  
    glutIdleFunc      ( animate );  
  
    init              ();
```

```
initExtensions ();
printfInfo      ();

if ( !isExtensionSupported ( "GL_ARB_shading_language_100" ) )
{
    printf ( "GL_ARB_shading_language_100 NOT supported.\n" );

    return 1;
}

if ( !isExtensionSupported ( "GL_ARB_shader_objects" ) )
{
    printf ( "GL_ARB_shader_objects NOT supported" );

    return 2;
}

GLint      linked;

                // create a vertex shader object
                // and a fragment shader object
vertexShader  = glCreateShaderObjectARB ( GL_VERTEX_SHADER_ARB );
fragmentShader = glCreateShaderObjectARB ( GL_FRAGMENT_SHADER_ARB );

                // load source code strings into
                // the shaders
if ( !loadShader ( vertexShader, "simplest.vsh" ) )
    exit ( 1 );

if ( !loadShader ( fragmentShader, "simplest.fsh" ) )
    exit ( 1 );

                // create a program object and attach
                // the two compiled shaders
program = glCreateProgramObjectARB ();

glAttachObjectARB ( program, vertexShader );
glAttachObjectARB ( program, fragmentShader );
```

```

        // link the program object and print
        // out the info log
    glLinkProgramARB ( program );

    if ( !checkOpenGLError() ) // check for OpenGL errors
        exit ( 1 );

    glGetObjectParameterivARB ( program, GL_OBJECT_LINK_STATUS_ARB,
                                &linked );

    printInfoLog ( program );

    if ( !linked )
        return 0;

        // install program object as part of
        // current state
    glUseProgramObjectARB ( program );
    glutMainLoop          ();

    return 0;
}

```

Эта программа загружает, компилирует и линкует простейшие шейдеры из листингов 13.1 и 13.2 и использует эти шейдеры для рендеринга "Чайника" (рис. 14.1).

Структура программы полностью соответствует рассмотренным расширениям — сначала проверяется поддержка всех необходимых расширений, после чего создаются два шейдера — вершинный и фрагментный.

```

// create a vertex shader object and a
// fragment shader object
vertexShader    = glCreateShaderObjectARB ( GL_VERTEX_SHADER_ARB );
fragmentShader = glCreateShaderObjectARB ( GL_FRAGMENT_SHADER_ARB );

```

После этого в каждый из них загружается исходный текст и компилируется.

```

// load source code strings into shaders
if (!loadShader (vertexShader, "simplest.vsh"))
    exit (1);

if (!loadShader (fragmentShader, "simplest.fsh"))
    exit (1);

```

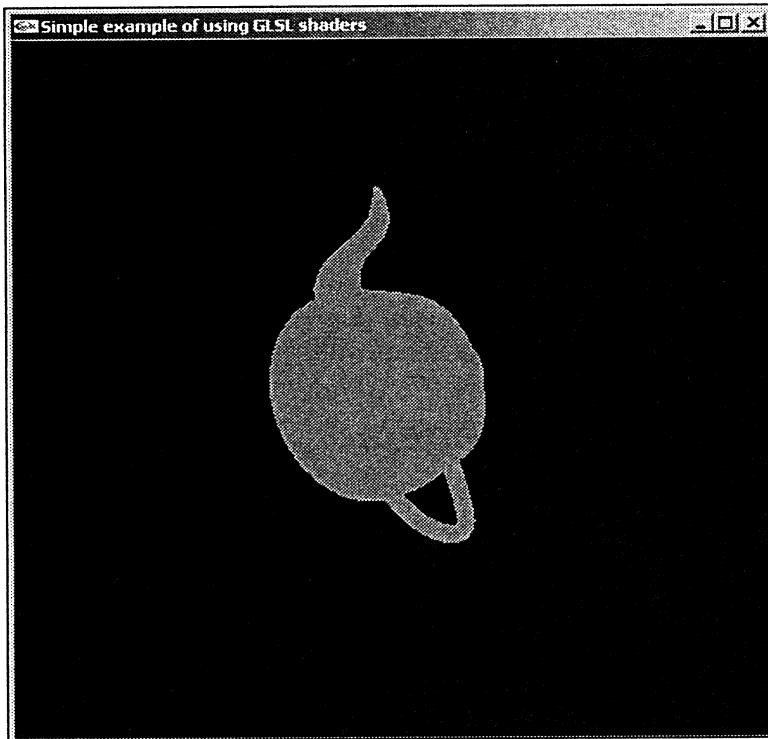


Рис. 14.1. Изображение, построенное программой из листинга 14.2

В случае успеха создается программный объект, к нему подсоединяются оба шейдера и проводится линковка программы.

```
// create a program object and attach the
// two compiled shaders
program = glCreateProgramObjectARB ();
glAttachObjectARB (program, vertexShader);
glAttachObjectARB (program, fragmentShader);

// link the program object and print out
// the info log
glLinkProgramARB (program);

// check for OpenGL errors
if (!checkOpenGLError())
    exit (1);
```

```
glGetObjectParameterivARB (program, GL_OBJECT_LINK_STATUS_ARB,
                           &linked);

printInfoLog (program);

if (!linked)
    return 0;
```

Это довольно простые и стандартные шаги, которые необходимо выполнять для каждой программы. Поэтому было бы очень удобно "завернуть" все эти шаги в класс, скрыв от пользователя различные технические детали.

"Заворачиваем" шейдеры на GLSL в класс

В листинге 14.3 описан класс `GslProgram`, инкапсулирующий работу с GLSL-программами.

Листинг 14.3. Описание класса `GslProgram`

```
//  
// Class to encapsulate GLSL program and shader objects and  
// working with them  
  
#ifndef __GLSL_PROGRAM__  
#define __GLSL_PROGRAM__  
  
#include "libExt.h"  
  
#include <string>  
  
using namespace std;  
  
class Vector2D;  
class Vector3D;  
class Vector4D;  
class Matrix3D;  
class Matrix4x4;  
class Data;
```

```
class GlslProgram
{
protected:
    GLhandleARB program;           // program object handle
    GLhandleARB vertexShader;
    GLhandleARB fragmentShader;
    bool          ok;             // whether program is loaded
                                // and ready to be used
    string        glError;
    string        log;

public:
    GlslProgram ();
    ~GlslProgram ();

                // load shaders
    bool      loadShaders ( const char * vertexFileName,
                           const char * fragmentFileName );
    bool      loadShaders ( Data * vertexShaderData,
                           Data * fragmentShaderData );

                // remove all shaders and free
                // all objects
    void      clear ();

string   getLog      () const      // get current log
{
    return log;
}

bool    isOk () const      // whether shader is ok
{
    return ok;
}

string   getGlError () const
{
    return glError;
```

```
}

void bind ();
void unbind ();

                                // uniform variables
                                // handling methods
bool    setUniformVector  ( const char * name,
                           const Vector4D& value );
bool    setUniformVector  ( int loc,
                           const Vector4D& value );
bool    setUniformVector  ( const char * name,
                           const Vector3D& value );
bool    setUniformVector  ( int loc,
                           const Vector3D& value );
bool    setUniformVector  ( const char * name,
                           const Vector2D& value );
bool    setUniformVector  ( const char * name,
                           const Vector2D& value );
bool    setUniformFloat   ( const char * name,
                           float value );
bool    setUniformFloat   ( int loc,
                           float value );
bool    setUniformMatrix  ( const char * name,
                           const Matrix4x4& value );
bool    setUniformMatrix  ( const char * name,
                           const Matrix3D& value );
bool    setUniformMatrix  ( const char * name,
                           float value [16] );
bool    setUniformInt     ( const char * name,
                           int value );
bool    setUniformInt     ( int loc,
                           int value );
Vector4D getUniformVector ( const char * name );
Vector4D getUniformVector ( int loc );
int     locForUniformName( const char * name );

                                // attribute variables handling
                                // methods
```

```
bool      setAttribute    ( const char * name,
                           const Vector4D& value );
bool      setAttribute    ( int index,
                           const Vector4D& value );
Vector4D  getAttribute   ( const char * name );
Vector4D  getAttribute   ( int index );
int       indexForAttrName ( const char * name );
bool      bindAttributeTo ( int no, const char * name );

bool      setTexture ( const char * name, int texUnit );
bool      setTexture ( int loc, int texUnit );

                                         // check whether there is a
                                         // support for GLSL

static bool      isSupported ();
static string    version        ();
                                         // some limitations on program

static int       maxVertexUniformComponents () ;
static int       maxVertexAttribs      () ;
static int       maxFragmentTextureUnits () ;
static int       maxVertexTextureUnits () ;
static int       maxCombinedTextureUnits () ;
static int       maxVaryingFloats     () ;
static int       maxFragmentUniformComponents () ;
static int       maxTextureCoords    () ;

protected:
    bool      loadShader    ( GLhandleARB shader, Data * data );
    bool      checkGlError  ();
    void      loadLog       ( GLhandleARB object );
};

#endif
```

Как видно из этого листинга, класс `GslProgram` содержит ряд простых и удобных функций для поддержки шейдеров. В первую очередь к ним относятся функции `isSupported` и `version`, позволяющие узнать, поддерживаются ли GLSL данным графическим процессором и драйвером, а также версию GLSL в случае наличия поддержки.

Функции `loadShaders` загружают шейдеры с их одновременной компиляцией и линковкой. Все возникающие при этом ошибки накапливаются во внутренней переменной `log`.

Также данный класс предоставляет ряд методов для доступа к uniform-переменным и вершинным атрибутам. Доступ может осуществляться как по индексу, так и по имени.

Методы `setTexture` служат для задания номера текстурного блока, в котором выбрана текстура, соответствующая имени `sampler`-переменной.

В листинге 14.4 приведена реализация данного класса.

Листинг 14.4. Реализация класса `GlslProgram`

```
//  
// Class to encapsulate GLSL program and shader objects  
  
//  
#include    "libExt.h"  
#include    "libTexture.h"  
#include    "Vector3D.h"  
#include    "Vector2D.h"  
#include    "Vector4D.h"  
#include    "Matrix3D.h"  
#include    "Matrix4x4.h"  
#include    "Data.h"  
#include    "GlslProgram.h"  
  
GlslProgram :: GlslProgram ()  
{  
    program      = 0;  
    vertexShader = 0;  
    fragmentShader = 0;  
    ok           = false;  
}  
  
GlslProgram :: ~GlslProgram ()  
{  
    clear ();  
}  
  
bool     GlslProgram :: loadShaders ( const char * vertexFileName,
```

```
const char * fragmentFileName )  
{  
ok = false;  
  
Data * vertexData = getFile ( vertexFileName );  
  
if ( vertexData == NULL )  
{  
    log += "Cannot open \\";  
    log += vertexFileName;  
    log += "\\";  
}  
else  
if ( !vertexData -> isOk () || vertexData -> isEmpty () )  
{  
    log += "Error loading vertex shader";  
  
    delete vertexData;  
  
    vertexData = NULL;  
}  
  
Data * fragmentData = getFile ( fragmentFileName );  
  
if ( fragmentData == NULL )  
{  
    log += "Cannot open \\";  
    log += fragmentFileName;  
    log += "\\";  
}  
else  
if ( !fragmentData -> isOk () || fragmentData -> isEmpty () )  
{  
    log += "Error loading fragment shader";  
  
    delete fragmentData;  
  
    fragmentData = NULL;  
}
```

```
bool result = loadShaders ( vertexData, fragmentData );

delete vertexData;
delete fragmentData;

return result;
}

bool GlslProgram :: loadShaders ( Data * vertexShaderData,
                                  Data * fragmentShaderData )
{
    ok = false;

        // check whether we should create
        // program object

    if ( program == 0 )
        program = glCreateProgramObjectARB ();

        // check for errors

    if ( !checkGlError () )
        return false;

        // create a vertex shader object and a
        // fragment shader object

    if ( vertexShaderData != NULL )
    {
        vertexShader = glCreateShaderObjectARB( GL_VERTEX_SHADER_ARB );
        log += "Loading vertex shader\n";

        if ( !loadShader ( vertexShader, vertexShaderData ) )
            return false;

        // attach shaders to program object
        glAttachObjectARB ( program, vertexShader );
    }

    if ( fragmentShaderData != NULL )
    {
        fragmentShader = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
        log += "Loading fragment shader\n";
    }
}
```

```
if ( !loadShader ( fragmentShader, fragmentShaderData ) )
    return false;
        . . .
        // attach shaders to program object
glAttachObjectARB ( program, fragmentShader );
}

GLint linked;

log += "Linking programs\n";

        // link the program object and print
        // out the info log
glLinkProgramARB ( program );

if ( !checkGLError () ) // check for errors
    return false;

glGetObjectParameterivARB ( program, GL_OBJECT_LINK_STATUS_ARB,
                            &linked );

loadLog ( program );

if ( !linked )
    return false;

return ok = true;
}

void GlslProgram :: bind ()
{
    glUseProgramObjectARB ( program );
}

void GlslProgram :: unbind ()
{
    glUseProgramObjectARB ( 0 );
}

bool GlslProgram :: loadShader ( GLhandleARB shader, Data * data )
```



```
glDeleteObjectARB ( vertexShader );
glDeleteObjectARB ( fragmentShader );

program      = 0;
vertexShader = 0;
fragmentShader = 0;
ok           = false;

}

void    GlslProgram :: loadLog ( GLhandleARB object )
{
    int      logLength      = 0;
    int      charsWritten   = 0;
    GLcharARB buffer [2048];
    GLcharARB * infoLog;

    glGetObjectParameterivARB ( object, GL_OBJECT_INFO_LOG_LENGTH_ARB,
                                &logLength );

    if ( !checkGlError() )          // check for OpenGL errors
        return;

    if ( logLength < 1 )
        return;
    // try to avoid allocating buffer
    if ( logLength > sizeof ( buffer ) )
    {
        infoLog = (GLcharARB*) malloc ( logLength );
        if ( infoLog == NULL )
        {
            log = "ERROR: Could not allocate log buffer";

            return;
        }
    }
    else
        infoLog = buffer;
```

```
glGetInfoLogARB ( object, logLength, &charsWritten, infoLog );  
  
log += infoLog;  
  
if ( infoLog != buffer )  
    free ( infoLog );  
}  
  
bool    GlslProgram :: setUniformVector ( const char * name,  
                                         const Vector4D& value )  
{  
    int loc = glGetUniformLocationARB ( program, name );  
  
    if ( loc < 0 )  
        return false;  
  
    glUniform4fvARB ( loc, 1, value );  
  
    return true;  
}  
  
bool    GlslProgram :: setUniformVector ( int loc,  
                                         const Vector4D& value )  
{  
    glUniform4fvARB ( loc, 1, value );  
  
    return true;  
}  
  
bool    GlslProgram :: setUniformVector ( const char * name,  
                                         const Vector3D& value )  
{  
    int loc = glGetUniformLocationARB ( program, name );  
  
    if ( loc < 0 )  
        return false;  
  
    glUniform3fvARB ( loc, 1, value );  
  
    return true;
```

```
}

bool GlslProgram :: setUniformVector ( int loc,
                                      const Vector3D& value )
{
    glUniform3fvARB ( loc, 1, value );

    return true;
}

bool GlslProgram :: setUniformVector ( const char * name,
                                      const Vector2D& value )
{
    int loc = glGetUniformLocationARB ( program, name );

    if ( loc < 0 )
        return false;

    glUniform2fvARB ( loc, 1, value );

    return true;
}

bool GlslProgram :: setUniformVector ( int loc, const Vector2D& value )
{
    glUniform2fvARB ( loc, 1, value );

    return true;
}

bool GlslProgram :: setUniformFloat ( const char * name, float value )
{
    int loc = glGetUniformLocationARB ( program, name );

    if ( loc < 0 )
        return false;

    glUniform1fARB ( loc, value );

    return true;
```

```
}

bool GlslProgram :: setUniformFloat ( int loc, float value )
{
    glUniform1fARB ( loc, value );

    return true;
}

bool      GlslProgram :: setUniformInt ( const char * name, int value )
{
    int loc = glGetUniformLocationARB ( program, name );

    if ( loc < 0 )
        return false;

    glUniform1iARB ( loc, value );

    return true;
}

bool      GlslProgram :: setUniformInt ( int loc, int value )
{
    glUniform1iARB ( loc, value );

    return true;
}

bool      GlslProgram :: setUniformMatrix ( const char * name,
                                             const Matrix4x4& value )
{
    int loc = glGetUniformLocationARB ( program, name );

    if ( loc < 0 )
        return false;

    glUniformMatrix4fvARB ( loc, 1, GL_FALSE, value [0] );

    return true;
```

```
}  
  
bool GlslProgram :: setUniformMatrix ( const char * name,  
                                      const Matrix3D& value )  
{  
    int loc = glGetUniformLocationARB ( program, name );  
  
    if ( loc < 0 )  
        return false;  
  
    glUniformMatrix3fvARB ( loc, 1, GL_FALSE, value [0] );  
  
    return true;  
}  
  
bool GlslProgram :: setUniformMatrix ( const char * name,  
                                      float value [16] )  
{  
    int loc = glGetUniformLocationARB ( program, name );  
  
    if ( loc < 0 )  
        return false;  
  
    glUniformMatrix4fvARB ( loc, 1, GL_FALSE, value );  
  
    return true;  
}  
  
int GlslProgram :: locForUniformName ( const char * name )  
{  
    return glGetUniformLocationARB ( program, name );  
}  
  
Vector4D GlslProgram :: getUniformVector ( const char * name )  
{  
    float values [4];  
  
    int loc = glGetUniformLocationARB ( program, name );  
  
    if ( loc < 0 )
```



```
int index = glGetAttribLocationARB ( program, name );

if ( index < 0 )
    return false;

glVertexAttrib4fvARB ( index, value );

return true;
}

bool GlslProgram :: setAttribute ( int index, const Vector4D& value )
{
    glVertexAttrib4fvARB ( index, value );

    return true;
}

int GlslProgram :: indexForAttrName ( const char * name )
{
    return glGetAttribLocationARB ( program, name );
}

Vector4D GlslProgram :: getAttribute ( const char * name )
{
    int index = glGetAttribLocationARB ( program, name );

    if ( index < 0 )
        return Vector4D ( 0, 0, 0, 0 );

    float buf [4];

    glGetVertexAttribfvARB ( index, GL_CURRENT_VERTEX_ATTRIB_ARB, buf );

    return Vector4D ( buf [0], buf [1], buf [2], buf [3] );
}

Vector4D GlslProgram :: getAttribute ( int index )
{
    float buf [4];
```

```
glGetVertexAttribfvARB ( index, GL_CURRENT_VERTEX_ATTRIB_ARB, buf );  
  
    return Vector4D ( buf [0], buf [1], buf [2], buf [3] );  
}  
  
bool GlslProgram :: isSupported ()  
{  
    return isExtensionSupported ( "GL_ARB_shading_language_100" ) &&  
           isExtensionSupported ( "GL_ARB_shader_objects" ) &&  
           isExtensionSupported ( "GL_ARB_vertex_shader" ) &&  
           isExtensionSupported ( "GL_ARB_fragment_shader" );  
  
}  
  
string GlslProgram :: version ()  
{  
    const char * slVer = (const char *) glGetString ( GL_SHADING_LANGUAGE_VERSION_ARB );  
  
    if ( glGetError() != GL_NO_ERROR )  
        return "1.051";  
  
    return string ( slVer );  
}  
  
int GlslProgram :: maxVertexUniformComponents ()  
{  
    int maxVertexUniformComponents;  
  
    glGetIntegerv ( GL_MAX_VERTEX_UNIFORM_COMPONENTS_ARB,  
                    &maxVertexUniformComponents );  
  
    return maxVertexUniformComponents;  
}  
  
int GlslProgram :: maxVertexAttribs ()  
{  
    int maxVertexAttribs;  
  
    glGetIntegerv ( GL_MAX_VERTEX_ATTRIBS_ARB, &maxVertexAttribs );
```

```
    return maxVertexAttribs;
}

int     GlslProgram :: maxFragmentTextureUnits ()
{
    int maxFragmentTextureUnits;

    glGetIntegerv ( GL_MAX_TEXTURE_IMAGE_UNITS_ARB,
                    &maxFragmentTextureUnits );

    return maxFragmentTextureUnits;
}

int     GlslProgram :: maxVertexTextureUnits ()
{
    int maxVertexTextureUnits;

    glGetIntegerv ( GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS_ARB,
                    &maxVertexTextureUnits );

    return maxVertexTextureUnits;
}

int     GlslProgram :: maxCombinedTextureUnits ()
{
    int maxCombinedTextureUnits;

    glGetIntegerv ( GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS_ARB,
                    &maxCombinedTextureUnits );

    return maxCombinedTextureUnits;
}

int     GlslProgram :: maxVaryingFloats ()
{
    int maxVaryingFloats;
```

```
glGetIntegerv ( GL_MAX_VARYING_FLOATS_ARB, &maxVaryingFloats );  
  
    return maxVaryingFloats;  
}  
  
int     GlslProgram :: maxFragmentUniformComponents ()  
{  
    int maxFragmentUniformComponents;  
  
    glGetIntegerv ( GL_MAX_FRAGMENT_UNIFORM_COMPONENTS_ARB,  
                    &maxFragmentUniformComponents );  
  
    return maxFragmentUniformComponents;  
}  
  
int     GlslProgram :: maxTextureCoords ()  
{  
    int maxTextureCoords;  
  
    glGetIntegerv ( GL_MAX_TEXTURE_COORDS_ARB, &maxTextureCoords );  
  
    return maxTextureCoords;  
}
```

В листинге 14.5 приводится вариант программы из листинга 14.2, переписанный с использованием класса `GlslProgram`.

Листинг 14.5. Простейший пример использования класса `GlslProgram`

```
//  
// Sample showing how to use GLSL programs  
//  
  
#include    "libExt.h"  
#include    <glut.h>  
#include    <stdio.h>  
#include    <stdlib.h>  
  
#include    "libTexture.h"
```

```
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Data.h"
#include    "GlslProgram.h"

Vector3D   eye   ( 7, 5, 7 );           // camera position
Vector3D   light ( 5, 0, 4 );           // light position
float      angle = 0;
Vector3D   rot   ( 0, 0, 0 );
int        mouseOldX = 0;
int        mouseOldY = 0;

GlslProgram program;

void init ()
{
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );
    glEnable     ( GL_DEPTH_TEST );
    glDepthFunc  ( GL_LEQUAL );

    glHint ( GL_POLYGON_SMOOTH_HINT,           GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}

void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix () ;

    glRotatef   ( rot.x, 1, 0, 0 );
    glRotatef   ( rot.y, 0, 1, 0 );
    glRotatef   ( rot.z, 0, 0, 1 );

    glutSolidTeapot ( 2 );

    glPopMatrix ();

    glutSwapBuffers ();
}
```

```
}

void reshape ( int w, int h )
{
    glViewport      ( 0, 0, (GLsizei)w, (GLsizei)h );
    glMatrixMode   ( GL_PROJECTION );
    glLoadIdentity ();
    gluPerspective ( 60.0, (GLfloat)w/(GLfloat)h, 1.0, 60.0 );
    glMatrixMode   ( GL_MODELVIEW );
    glLoadIdentity ();
    gluLookAt       ( eye.x, eye.y, eye.z,      // eye
                      0, 0, 0,                  // center
                      0.0, 0.0, 1.0 );         // up
}

void motion ( int x, int y )
{
    rot.y -= ((mouseOldY - y) * 180.0f) / 200.0f;
    rot.z -= ((mouseOldX - x) * 180.0f) / 200.0f;
    rot.x = 0;

    if ( rot.z > 360 )
        rot.z -= 360;

    if ( rot.z < -360 )
        rot.z += 360;

    if ( rot.y > 360 )
        rot.y -= 360;

    if ( rot.y < -360 )
        rot.y += 360;

    mouseOldX = x;
    mouseOldY = y;

    glutPostRedisplay ();
}

void mouse ( int button, int state, int x, int y )
```

```
{  
    if ( state == GLUT_DOWN )  
    {  
        mouseOldX = x;  
        mouseOldY = y;  
    }  
}  
  
void key ( unsigned char key, int x, int y )  
{  
    if ( key == 27 || key == 'q' || key == 'Q' )      // quit requested  
        exit ( 0 );  
}  
  
void     animate ()  
{  
    angle = 0.004f * glutGet ( GLUT_ELAPSED_TIME );  
  
    glutPostRedisplay ();  
}  
  
int main ( int argc, char * argv [] )  
{  
    // initialize glut  
    glutInit          ( &argc, argv );  
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );  
    glutInitWindowSize ( 500, 500 );  
  
    // create window  
    glutCreateWindow ("Simple example of using GLSL shaders");  
  
    // register handlers  
    glutDisplayFunc   ( display );  
    glutReshapeFunc   ( reshape );  
    glutKeyboardFunc  ( key );  
    glutMouseFunc     ( mouse );  
    glutMotionFunc    ( motion );  
    glutIdleFunc      ( animate );  
  
    init             ();
```

```
initExtensions ();
printfInfo      ();

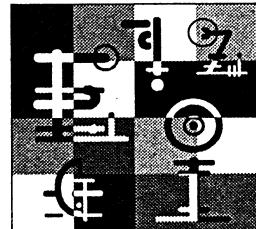
assertExtensionsSupported ( "GL_ARB_shading_language_100 " \
                           "GL_ARB_shader_objects " );

if ( !program.loadShaders ( "simplest.vsh", "simplest.fsh" ) )
{
    printf ( "Error loading shaders:\n%s\n",
             program.getLog ().c_str () );
    exit ( 1 );
}

// install program object as part of
// current state
program.bind ();
glutMainLoop ();

return 0;
}
```

Глава 15



Разработка шейдеров на GLSL в интегрированной среде RenderMonkey

Несмотря на простоту самого языка GLSL, задача написания шейдеров на нем может оказаться довольно сложной и трудоемкой. И одна из основных причин заключается в том, что часто для получения нужного эффекта очень сложно подобрать параметры и формулы, ввести новые параметры и т. п. Если все это делать вручную (подправить код основной программы на C++, откомпилировать его, подправить шейдеры, запустить получившуюся программу, посмотреть на результаты и т. д., рис. 15.1), то процесс получается действительно очень длительным и однообразным.

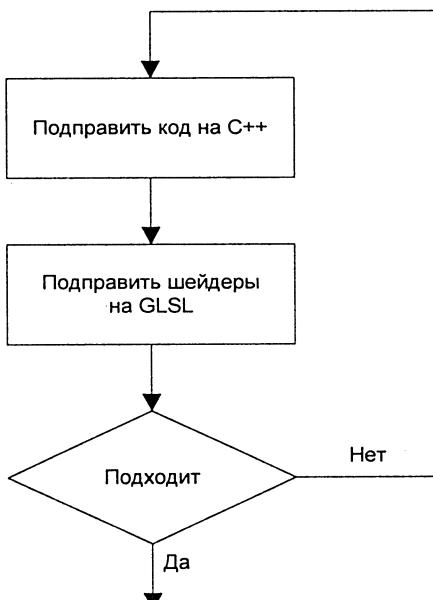


Рис. 15.1. Традиционный подход к написанию и отладке шейдеров

Именно здесь неоценимую помощь оказывают интегрированные среды (IDE) для разработки шейдеров. Одну такую среду — пакет RenderMonkey 1.5 компании ATI — мы и рассмотрим в этой главе. Данная среда позволяет разрабатывать и отлаживать шейдеры как для OpenGL (с помощью GLSL), так и для Direct3D (с помощью HLSL). Мы будем, однако, рассматривать исключительно работу с OpenGL и GLSL.

Примечание

Подробную информацию по установке и работе с RenderMonkey можно найти в [14].

Запустите RenderMonkey. На экране появится окно, похожее на приведенное на рис. 15.2.

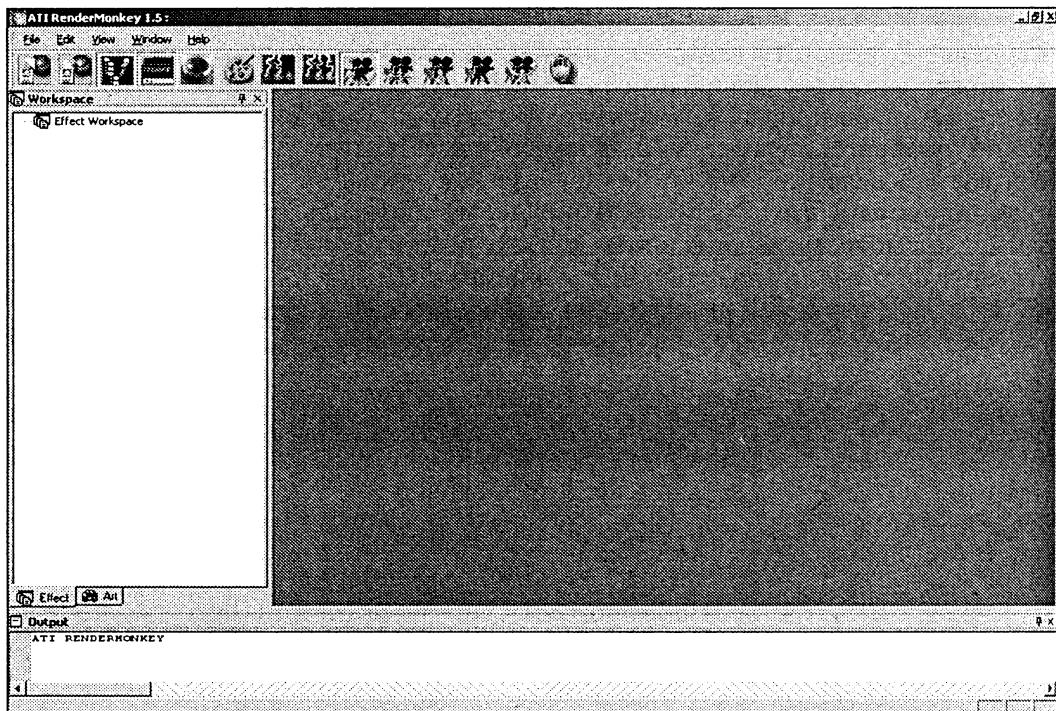


Рис. 15.2. Рабочее окно RenderMonkey

В верхней части окна расположены главное меню и панель инструментов (рис. 15.3), в левой части — окно проекта (**Workspace**), в нижней части — окно вывода (**Output**).

Окно вывода Output служит для вывода результатов компиляции, ошибок и предупреждений. Вы можете легко изменить его размер, захватив мышью его верхний край.



Рис. 15.3. Панель инструментов

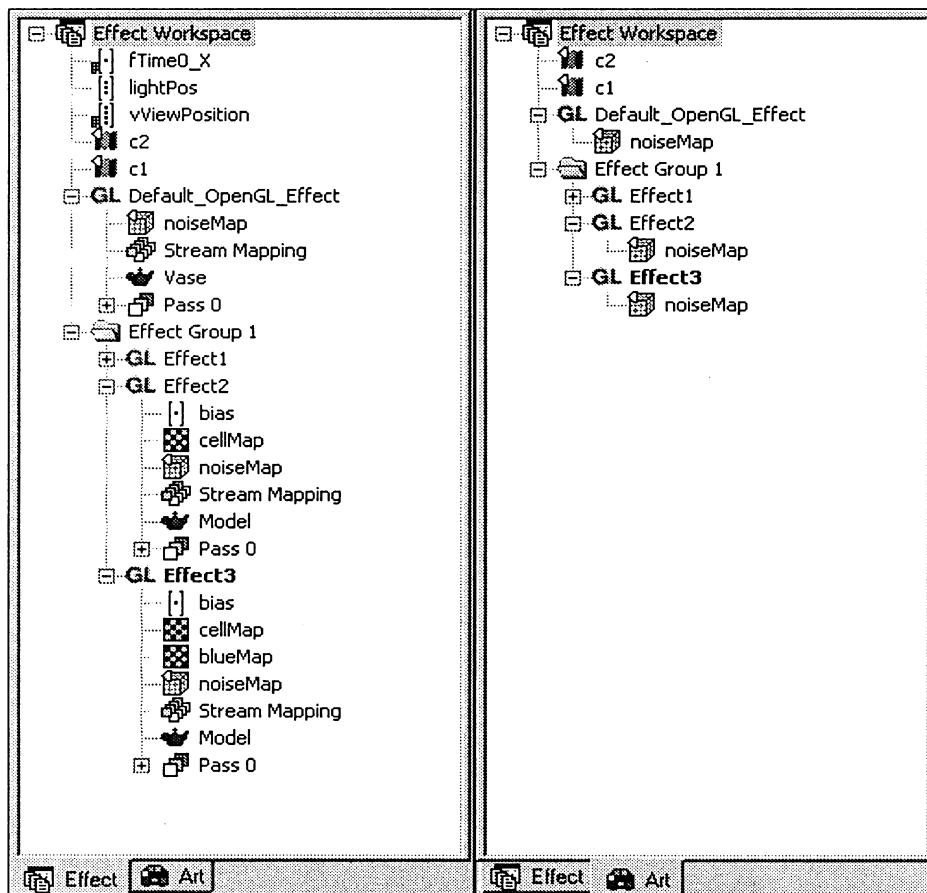


Рис. 15.4. Окно проекта RenderMonkey

Окно проекта Workspace содержит две вкладки — **Effect** (рис. 15.4).

Первая вкладка предназначена для программистов-разработчиков шейдеров и больше ориентирована на работу с кодом шейдеров. Вторая вкладка больше ориентирована на художников — в ней можно работать с текстурами и подбирать отдельные параметры (цвета, константы).

Такое разделение проекта на две части позволяет художнику и программисту работать с одним и тем же проектом без риска для одного испортить какие-то данные другого.

Среда организована настолько удобно, что главным меню пользоваться практически не приходится — основная работа легко выполняется с помощью панели инструментов и контекстных (всплывающих, рорир) меню.

Кнопки панели инструментов:

- Open Workspace** — открыть существующий проект;
- Save Workspace** — сохранить текущий проект (если он еще ни разу не был сохранен, то откроется диалоговое окно **Save As**);
- Workspace Window** — показать/скрыть окно проекта;
- Output Window** — показать/скрыть окно вывода;
- Preview window** — показать/скрыть окно рендеринга (обычно оно появляется при выполнении шейдера и в нем видно построенное с использованием шейдеров изображение объектов);
- Artist Editor** — открыть окно редактора для художника;
- Compile Active Effect** — откомпилировать все шейдеры текущего эффекта;
- Compile All Shaders in the Workspace** — откомпилировать все шейдеры в данном проекте;
- группа из пяти кнопок-переключателей (в каждый момент времени активной является только одна из них, за исключением кнопки **Camera Home**):
 - **Rotate Camera** — в этом режиме, удерживая нажатой левую кнопку мыши, можно поворачивать объект (точнее камеру) в окне рендеринга; при помощи колеса мыши можно приближать/удалять объект (камеру);
 - **Move Camera** — в этом режиме, удерживая нажатой левую кнопку мыши, можно передвигать объект в стороны в окне рендеринга; при помощи колеса мыши можно приближать/удалять объект (камеру);
 - **Zoom Camera** — в этом режиме, удерживая нажатой левую кнопку мыши, можно приближать/удалять камеру;
 - **Camera Home** — нажатие этой кнопки восстанавливает исходные параметры камеры (она не задает никакого режима и не остается нажатой);
 - **Overload Camera Mode** — в этом режиме, удерживая нажатой левую кнопку мыши, можно вращать объект; если при этом удерживать нажатой клавишу **<Ctrl>**, то движение мыши будет передвигать объект; при помощи колеса мыши можно приближать/удалять объект (камеру).
- Mouse Input Mode** — в этом режиме действия мыши вообще не затрагивают положение/ориентацию объекта, но вся информация о мыши поступает в специальные внутренние переменные (доступные для шейдеров).

Примечание

Если подвести курсор мыши к кнопке панели инструментов и задержать его на несколько секунд, то появится всплывающая подсказка.

Настройка RenderMonkey

Поскольку в среде RenderMonkey можно работать как с OpenGL, так и с Direct3D, необходимо настроить ряд параметров, которые отличаются в этих библиотеках.

Для этого при помощи команды **Edit \ Preferences** главного меню откройте окно настроек (рис. 15.5).

На вкладке **General** выберите правостороннюю ориентацию системы координат (**RHS**) в раскрывающемся списке **Default Model Orientation**.

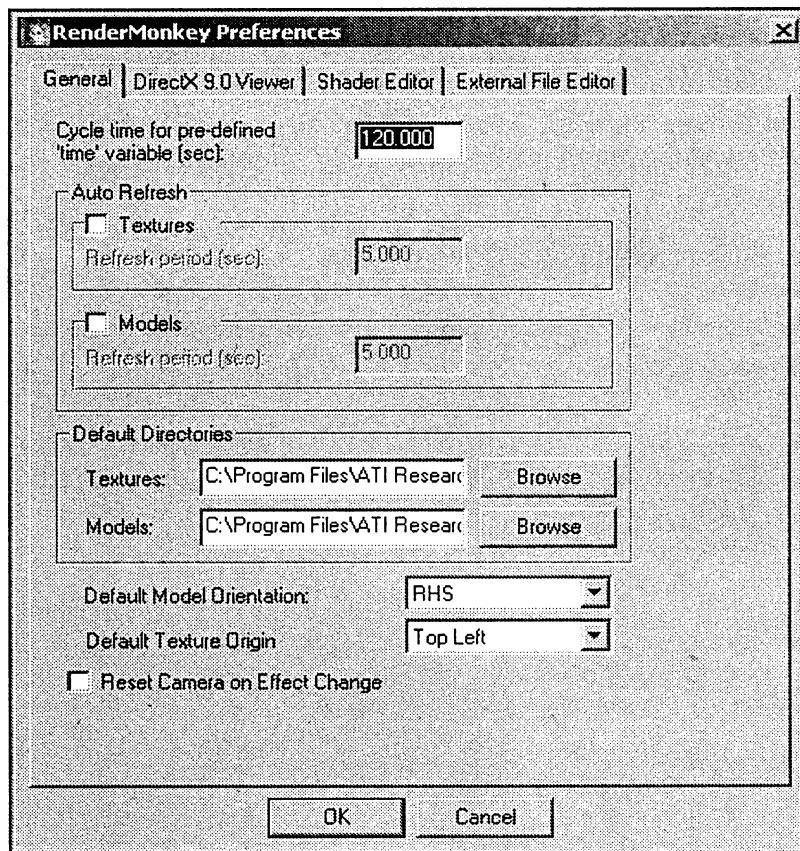


Рис. 15.5. Окно настроек RenderMonkey

В раскрывающемся списке **Default Texture Origin** можно выбрать точку, которая считается началом текстуры (т. е. соответствует текстурным координатам $(0, 0)$).

В группе **Default Directories** можно указать стандартные пути для текстур (раскрывающийся список **Textures**) и моделей (раскрывающийся список **Models**) — каталоги, на которых открываются диалоги для выбора текстур и моделей. Выбрать каталог можно с помощью кнопки **Browse**.

Вкладка **Shader Editor** позволяет настроить параметры для редактора кода шейдеров.

Оставшиеся две вкладки содержат параметры редактора текстов шейдеров. Мы не рассматриваем их подробно, т. к. стандартные установки хорошо работают и изменять их не требуется.

Создание нового проекта

Чтобы создать новый проект, в меню **File** выберите команду **New** (или нажмите комбинацию клавиш **<Ctrl>+<N>**). При этом текущий проект (если он был) сохраняется и вы увидите пустое окно проекта.

Выберите в окне **Workspace** вкладку **Effect**.

Весь проект в окне проекта RenderMonkey представлен в виде дерева, состоящего из отдельных узлов. Каждому узлу соответствует эффект, модель, текстура, переменная и т. д.

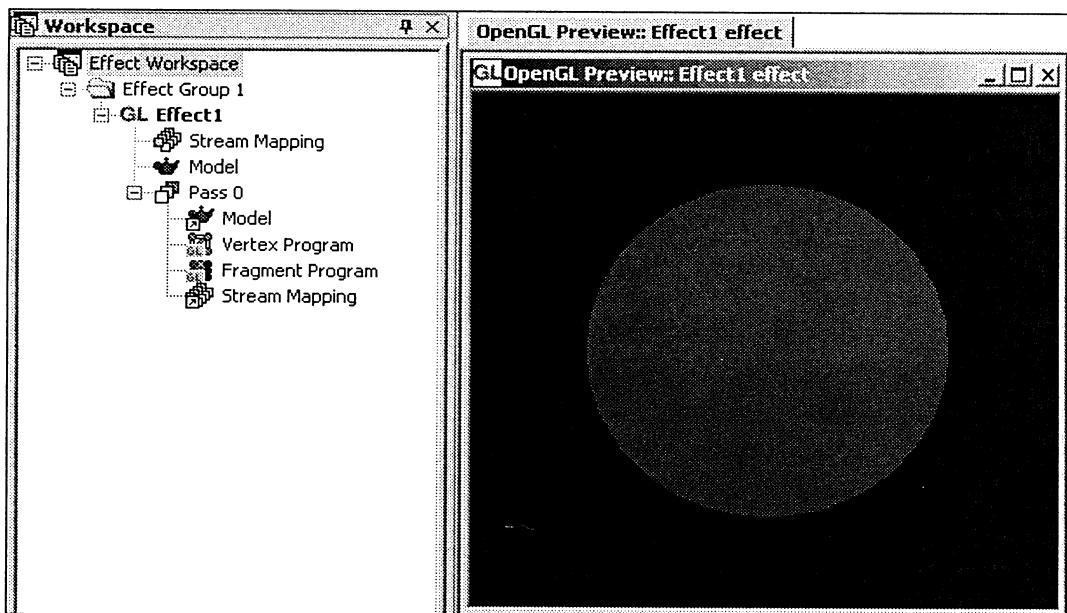


Рис. 15.6. Стандартный проект

RenderMonkey группирует шейдеры в **эффекты** (effect). Каждый эффект может состоять из нескольких проходов (pass), внутри каждого прохода есть свой вершинный и фрагментный шейдер.

Для начала работы щелчком правой кнопки мыши на корне дерева (**Effect Workspace**) вызовите всплывающее меню и в нем выберите команду **Add Effect Group \ Effect Group w / OpenGL Effect**.

После этого появится окно просмотра результатов рендеринга **Prewiew** (обычно в нем отображается синий шар) и в дереве проекта появятся узлы. Последовательно раскройте узлы **Effect 1** и **Pass 0**.

Для работы с произвольным узлом используется всплывающее меню (открывается щелчком правой кнопки мыши на узле) или окно редактирования (открывается двойным щелчком левой кнопки мыши на соответствующем узле).

Двойной щелчок на узле **Model** открывает диалоговое окно выбора модели. Выберите в нем файл **ElephantBody.3ds**.

Следующим шагом будет выбор атрибутов для каждой вершины модели. Для этого двойным щелчком на узле **Stream Mapping** откройте одноименное окно (рис. 15.7).

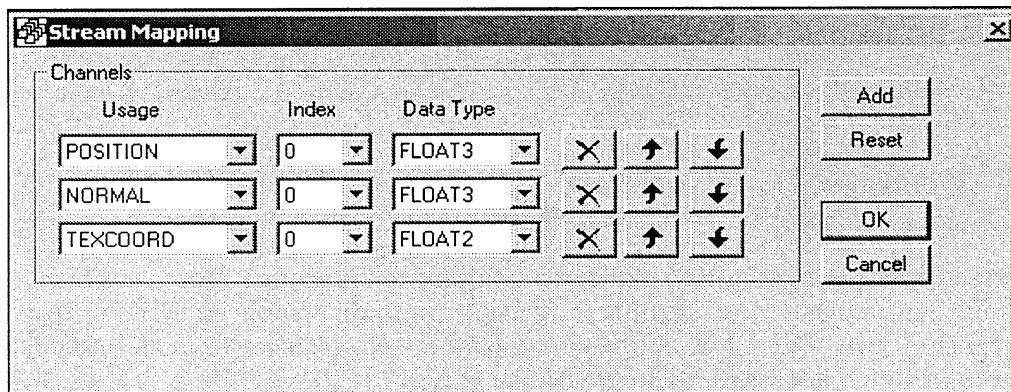


Рис. 15.7. Окно выбора вершинных атрибутов

В группе **Channels** находится список всех атрибутов, связанных с каждой вершиной. Каждый атрибут обладает следующими характеристиками — назначением (**Usage**), индексом (**Index**) и типом используемых данных (**Data Type**).

Назначение атрибута характеризует тип передаваемых в нем данных. Наиболее распространенными являются координаты (**POSITION**), цвет (**COLOR**), нормаль (**NORMAL**), текстурные координаты (**TEXCOORD**) (для задания нескольких текстурных координат для одной вершины используется параметр **Index**), касательный вектор (**TANGENT**) и бинормаль (**BINORMAL**).

Тип данных (**FLOAT1**, **FLOAT2**, **FLOAT3**, **FLOAT4**, **UBYTE4** и т. п.) определяет структуру данных, используемую для передачи данного атрибута.

Справа от поля типа расположены три кнопки, предназначенные соответственно для уничтожения атрибута, для его перемещения в списке на одну позицию вверх и для его перемещения в списке на одну позицию вниз.

Расположенная в верхнем правом углу окна кнопка **Add** позволяет добавлять к списку новые атрибуты.

Добавьте атрибуты, установите их свойства в соответствии с рис. 15.8 и закройте окно щелчком на кнопке **OK**.

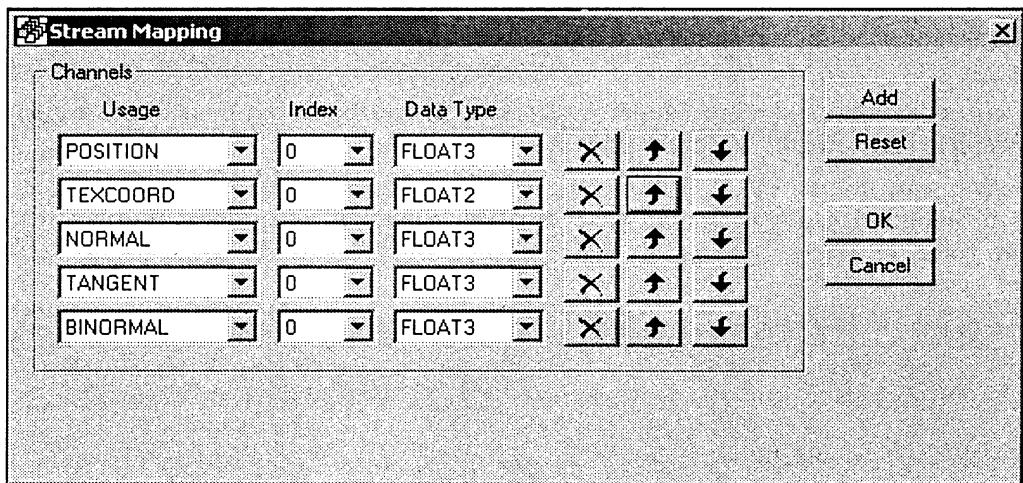


Рис. 15.8. Используемый набор вершинных атрибутов

Для связывания созданного набора вершинных атрибутов (**Stream Mapping**) с конкретным проходом щелкните правой кнопкой мыши на значке **Stream Mapping** в узле **Pass 0** и в появившемся меню последовательно выберите значения **Reference Node** и **Stream Mapping** (рис. 15.9).

Примечание

В левом нижнем углу значка, выбранного нами в проходе **Pass 0** набора вершинных атрибутов, имеется маленькая стрелка (такая же стрелка есть на значке **Model** в узле **Pass 0**). Подобная стрелка на любом значке обозначает, что данный узел является ссылкой (reference) на другой узел проекта.

Для редактирования шейдеров вызовите редактор шейдеров двойным щелчком на узле любого из шейдеров в проходе **Pass 0**.

Появится окно **Shader Editor** (рис. 15.10).

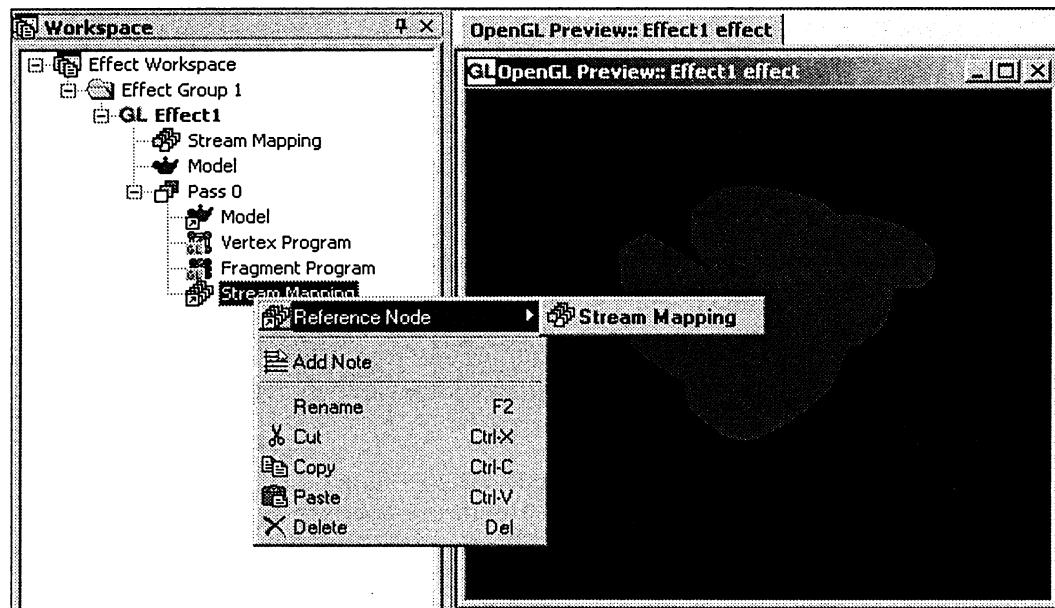


Рис. 15.9. Выбор атрибутов для конкретного прохода

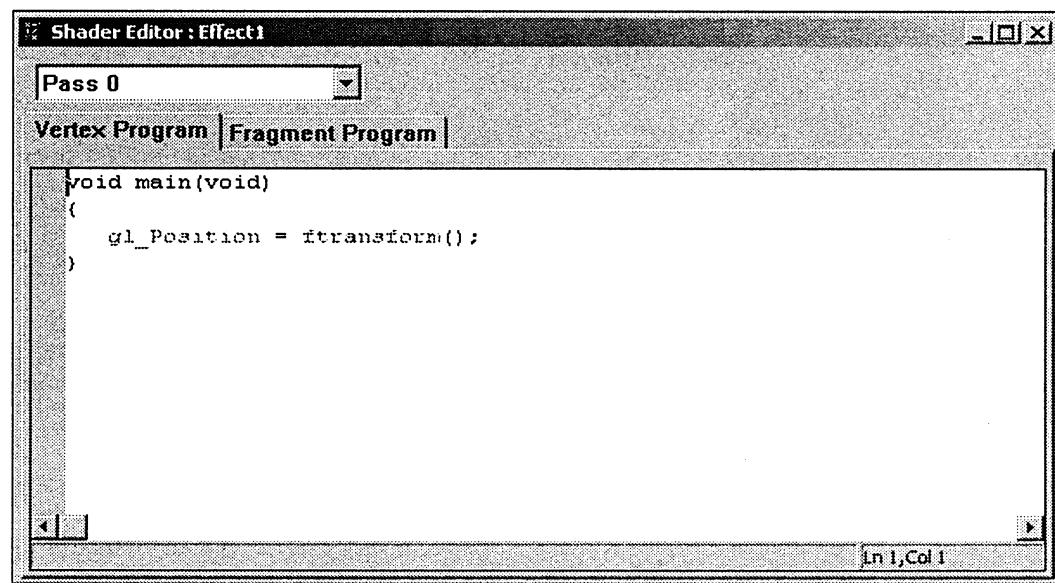


Рис. 15.10. Окно редактора шейдеров

В комбинированном списке, расположеннном в верхнем левом углу этого окна, можно выбрать проход, шейдеры которого отображаются в окне редактирования.

Вкладки **Vertex Program** и **Fragment Program** предоставляют доступ к текстам вершинного и фрагментного шейдеров.

В нижнем правом углу окна отображается текущее положение курсора в редактируемом тексте шейдера.

Для доступа из вершинной программы к касательной и бинормали добавьте к тексту вершинного шейдера следующие две строки:

```
attribute vec3 rm_Binormal;
attribute vec3 rm_Tangent;
```

Также добавьте в тело вершинного шейдера следующую строку:

```
gl_TexCoord [0] = gl_MultiTexCoord0;
```

Теперь добавим к текущему эффекту текстуру — щелчком правой кнопки на узле **Effect 1** откройте его контекстное меню и выберите пункт **Add Texture \ Add 3D Texture \ C:/..../Examples/Media/Textures/NoiseVolume.dds** (рис. 15.11).

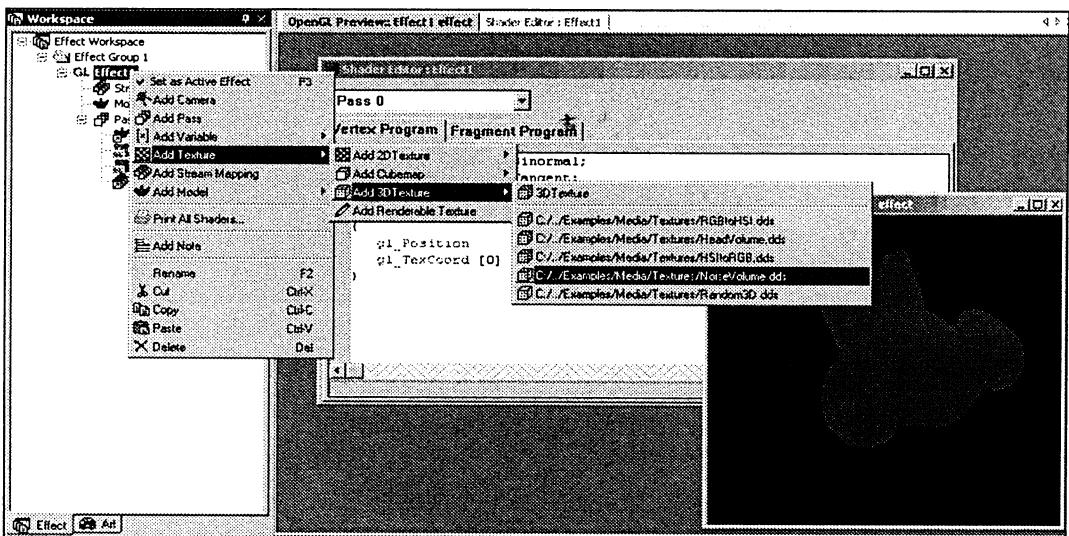


Рис. 15.11. Добавление текстуры к проекту

Теперь для того, чтобы добавленную текстуру можно было использовать в проходе **Pass 0**, необходимо в этом проходе добавить так называемый *текстурный объект* (Texture Object) (фактически это означает выделение под нее блока текстурирования).

Для этого в контекстном меню узла **Pass 0** выберите пункт **Add Texture Object**. Раскройте появившийся узел **Texture 0**.

Значки самого узла **Texture 0** и его дочернего узла **baseMap** перечеркнуты красным — это значит, что данному узлу не поставлена в соответствие ни одна текстура (рис. 15.12).

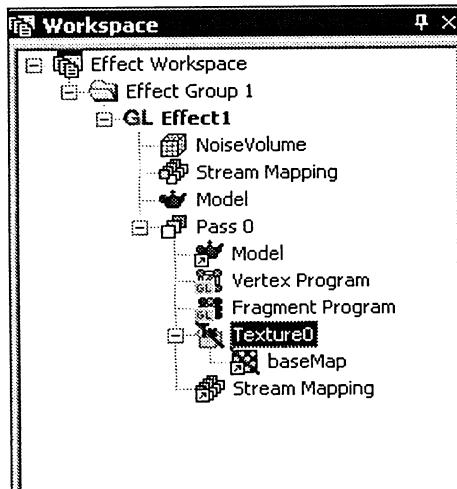


Рис. 15.12. Текстурному объекту не сопоставлена текстура

Сопоставьте узлу **Texture 0** текстуру **NoiseVolume**. Для этого откройте контекстное меню узла **baseMap** (щелчком правой кнопки мыши на узле) и выберите пункт **Reference Node \ NoiseVolume** (рис. 15.13).

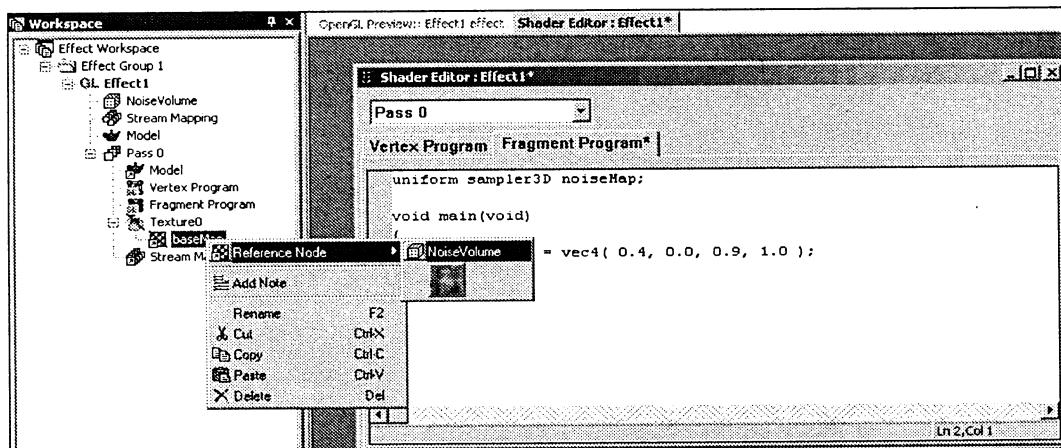


Рис. 15.13. Назначение текстуры текстурному объекту

С помощью команды **Rename** контекстного меню переименуйте узел **Texture 0** в **noiseMap**. Для доступа к этой текстуре из фрагментного шейдера добавьте в него следующую строку:

```
uniform sampler3D noiseMap;
```

Двойным щелчком на узле **noiseMap** откройте диалоговое окно для задания параметров данной текстуры (рис. 15.14).

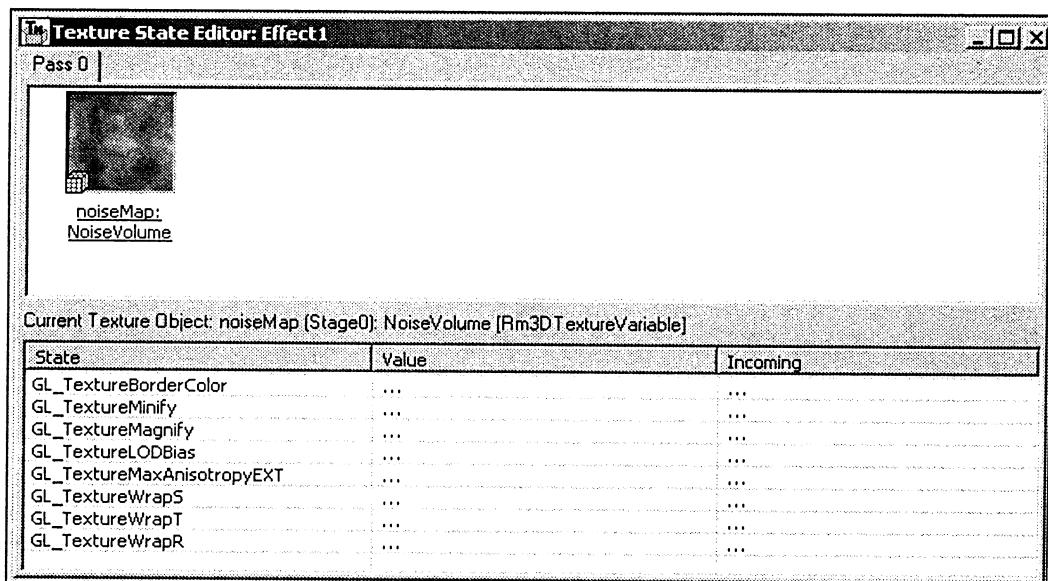


Рис. 15.14. Окно настройки параметров текстуры

Щелчком мыши на ячейке таблицы в столбце **Value** можно выбрать значение для соответствующего параметра текстуры.

Следующий возможностью среди RenderMonkey, которой мы сейчас воспользуемся, является добавление переменных. При этом можно добавлять переменные следующих типов (рис. 15.15) — **Boolean** (логическая переменная или вектор), **Integer** (целочисленная переменная или вектор), **Float** (вещественная переменная или вектор), **Matrix** (двух-, трех- или четырехмерная матрица), **Color** (цветовая переменная — RGBA-вектор).

Теперь добавим проходу **Pass 0** вещественную переменную **freqScale** — откроем контекстное меню этого узла и в нем выберем пункт **Add Variable \ Float \ Float** (рис. 15.16).

С помощью контекстного меню узла добавленной переменной переименуем ее в **freqScale** (рис. 15.17) и установим ее значение в 4.0.

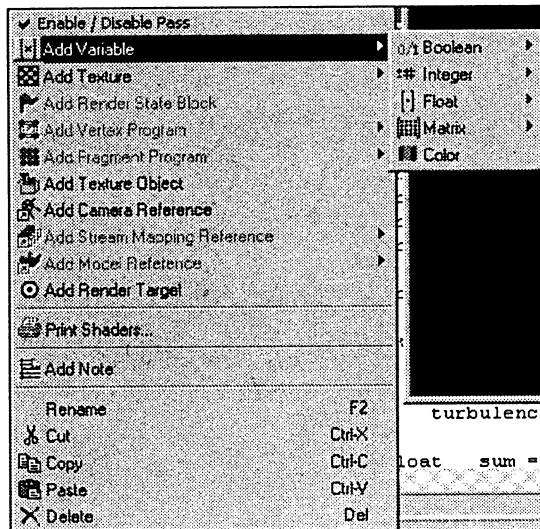
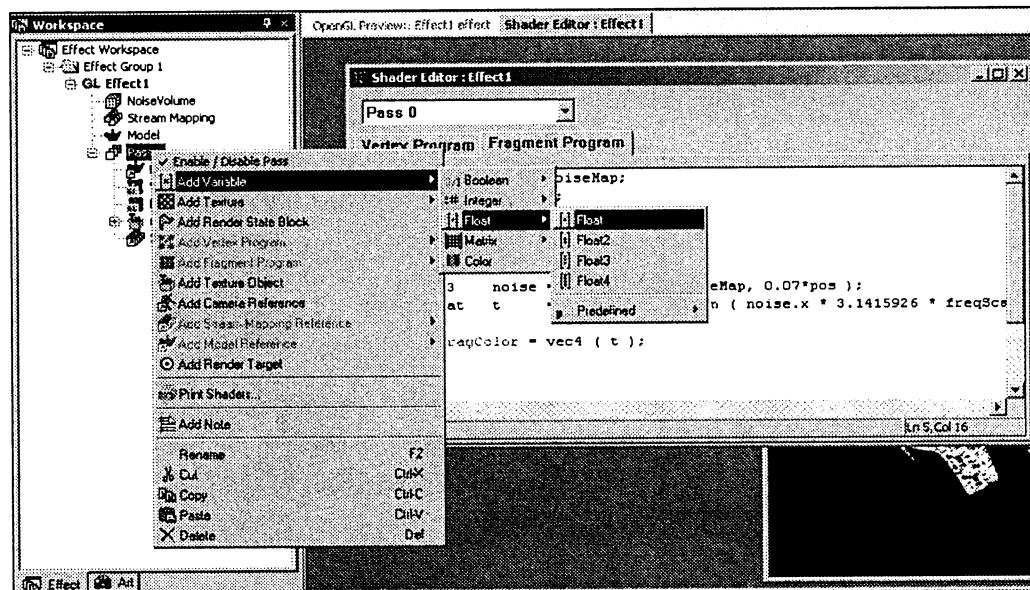
Рис. 15.15. Добавление переменных с помощью команды **Add Variable**

Рис. 15.16. Добавление вещественной переменной

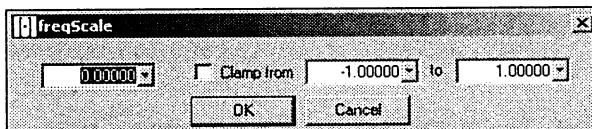


Рис. 15.17. Окно выбора значения вещественной переменной

Подправим теперь тексты шейдеров, чтобы привести их в соответствие с листингами 15.1 и 15.2.

Листинг 15.1. Вершинный шейдер

```
attribute vec3    rm_Binormal;
attribute vec3    rm_Tangent;

varying   vec3    pos;

void main(void)
{
    pos          = gl_Vertex.xyz;
    gl_Position   = ftransform();
    gl_TexCoord [0] = gl_MultiTexCoord0;
}
```

Листинг 15.2. Фрагментный шейдер

```
uniform sampler3D noiseMap;
varying   vec3    pos;
uniform   float   freqScale;

void main(void)
{
    vec3    noise = texture3D ( noiseMap, 0.07 * pos );
    float  t      = 0.5 * ( 1.0 + sin ( noise.x * 3.1415926 *
                                    freqScale ) );

    gl_FragColor = vec4 ( t );
}
```

Обратите внимание на объявление uniform-переменной `freqScale` — она автоматически связывается с узлом `freqScale` в проходе **Pass 0** при выполнении шейдера.

Откомпилировав этот эффект, мы получаем изображение слоника, покрытого странным черно-белым рисунком (рис. 15.18).

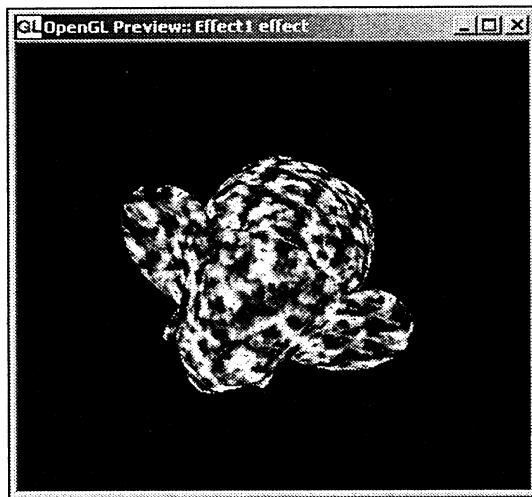


Рис. 15.18. Изображение, полученное при помощи шейдеров из листингов 15.1 и 15.2

Если сейчас открыть окно выбора значения для переменной `freqScale` и начать изменять значение этой переменной, то нам сразу же будет видно влияние этого изменения на изображение в окне просмотра результатов рендеринга.

Чтобы сделать нашего слоника цветным, добавим переменные `c1` и `c2` с помощью команды **Add Variable \ Color** контекстного меню узла **Pass 0**.

Введем эти переменные в фрагментный шейдер, добавив в него следующие две строки:

```
uniform vec4 c1;  
uniform vec4 c2;
```

Изменим возвращаемое фрагментным шейдером значение цвета на `mix(c1, c2, t)`, чтобы сделать слоника цветным.

Однако если сейчас откомпилировать наш проект, то слоник по-прежнему останется черно-белым — по умолчанию всем создаваемым цветовым переменным присваивается значение `(1, 1, 1, 1)`.

Выберем для переменной `c1` другое значение цвета — для этого достаточно лишь открыть окно выбора цвета (рис. 15.19) двойным щелчком на узле переменной `c1`.

Теперь можно свободно менять значения переменных `c1`, `c2` и `freqScale` для получения желаемого визуального эффекта.

На рис. 15.20 приведен пример того, что может получиться, — слоник как будто сделан из фантастического камня.

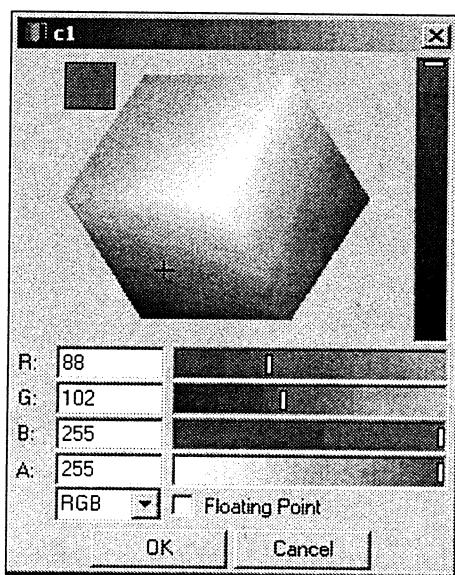


Рис. 15.19. Окно выбора цвета

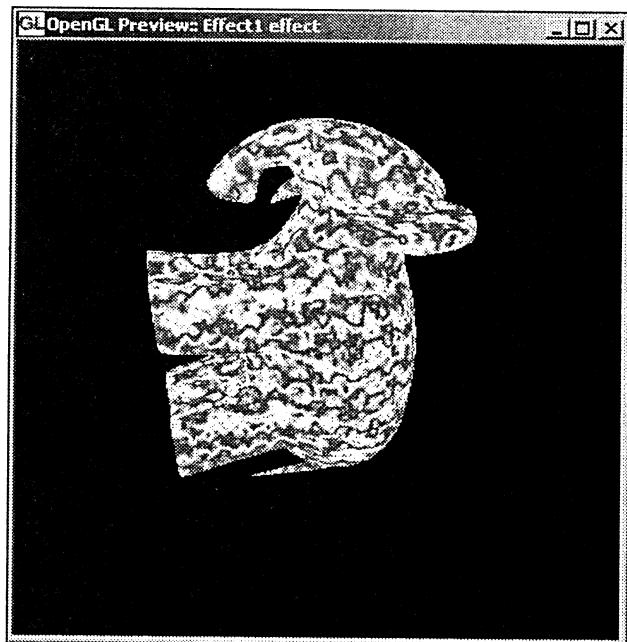


Рис. 15.20. Изображение объекта после подбора параметров

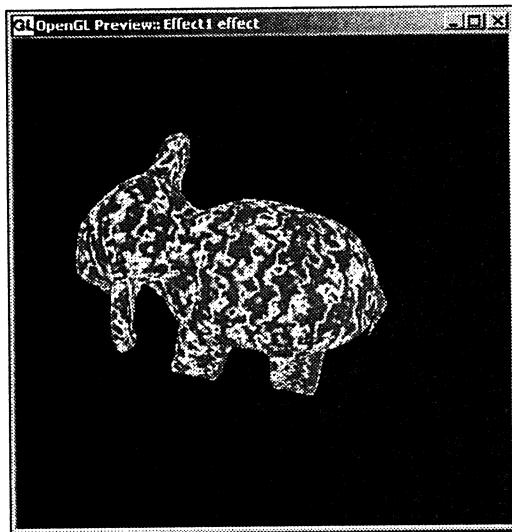


Рис. 15.21. Изображение, полученное при t^3

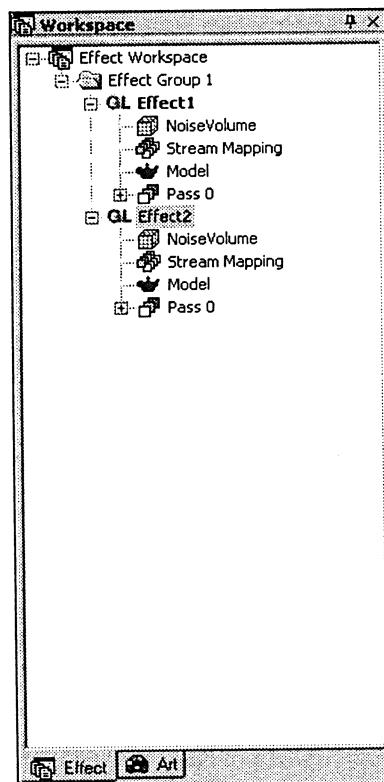


Рис. 15.22. Добавляем новый эффект

Попробуем сделать белые полоски на нем потоньше. Этого проще всего достичь путем возвведения переменной t , используемой для смешения цветов, в степень больше 1. На рис. 15.21 приведен результат возвведения этой переменной в куб.

Попробуем создать еще один эффект на той же основе. Для этого достаточно использовать стандартное копирование/вставку (Copy/Paste) для создания копии эффекта **Effect 1** в нашем проекте (правда, ему автоматически будет присвоено имя **Effect 2**) (рис. 15.22).

Примечание

Окно просмотра результатов рендеринга только одно, и оно отображает содержимое активного эффекта. Выбрать эффект как активный можно либо при помощи контекстного меню узла эффекта или выделив узел и нажав клавишу <F3>.

Добавим новые вещественные переменные `redScale` и `redPow` и изменим фрагментный шейдер для добавления в наш материал еще одного цвета — красного (листинг 15.3).

Листинг 15.3. Измененный фрагментный шейдер

```
uniform sampler3D noiseMap;
uniform float freqScale;
uniform float noiseScale;
uniform float redScale;
uniform float redPow;
uniform vec4 c1;
uniform vec4 c2;
varying vec4 pos;

void main(void)
{
    vec3 noise = 0.07 * pos.x + noiseScale * texture3D ( noiseMap,
                                                       0.07 * pos );
    float t      = 0.5 * ( 1.0 + sin ( noise.x * 3.1415926 * freqScale ) );
    float t2     = clamp ( pow ( t * redScale, redPow ), 0, 1 );

    gl_FragColor = mix ( c1, c2, t * t * t ) + vec4 ( 1, 0, 0, 1 ) * t2;
}
```

Если откомпилировать полученный шейдер, то станет видно, что мы добавили элементы красного цвета к нашему материалу.

Теперь воспользуемся еще одной возможностью программы RenderMonkey — предопределенными переменными. Среда RenderMonkey содержит большое количество переменных, таких как время с начала запуска и различные функции от него, параметры плоскостей отсечения и т. д. (табл. 15.1 и 15.2).

Таблица 15.1. Предопределенные переменные RenderMonkey, связанные с временем

Имя	Значение
fTime0_X	Вещественное значение, представляющее время в пределах временного цикла (по умолчанию величина цикла — 120 с)
fCosTime0_X	Косинус fTime0_X
fSinTime0_X	Синус fTime0_X
fTanTime0_X	Тангенс fTime0_X
fTime0_X_4f	Четырехмерный вектор, построенный из переменных fTime0_X, fCosTime0_X, fSinTime0_X и fTanTime0_X
fTime0_1	Нормированное значение величины fTime0_X (т. е. приведенное к отрезку [0, 1])
fCosTime0_1	Косинус fTime0_1
fSinTime0_1	Синус fTime0_1
fTanTime0_1	Тангенс fTime0_1
fTime0_1_4f	Четырехмерный вектор, построенный из переменных fTime0_1, fCosTime0_1, fSinTime0_1 и fTanTime0_1
fTime0_2PI	Значение fTime0_X, приведенное к отрезку [0, 2π]
fCosTime0_2PI	Косинус fTime0_2PI
fSinTime0_2PI	Синус fTime0_2PI
fTanTime0_2PI	Тангенс fTime0_2PI
fTime0_2PI_4f	Четырехмерный вектор, построенный из переменных fTime0_2PI, fCosTime0_2PI, fSinTime0_2PI и fTanTime0_2PI
fTimeCyclePeriod	Период цикла по времени (по умолчанию — 120 с)
fFPS	Число кадров в секунду (вещественное число)
fTimeElapsed	Время между соседними кадрами (в секундах)

Таблица 15.2. Предопределенные переменные *RenderMonkey*, связанные с окном рендеринга

Имя	Значение
fViewportWidth	Ширина окна в пикселях (вещественное число)
fViewportHeight	Высота окна в пикселях (вещественное число)
fInverseViewportWidth	1.0 / fViewportWidth
fInverseViewportHeight	1.0 / fViewportHeight
fFOV	Угол обзора камеры (Field Of View)
fFarClipPlane	Расстояние до ближней плоскости отсечения
fNearClipPlane	Расстояние до дальней плоскости отсечения

Среда *RenderMonkey* предоставляет в распоряжение разработчика шейдеров набор случайных величин (принимающих значения на отрезке [0, 1]), уникальных для каждого прохода и эффекта (табл. 15.3).

Таблица 15.3. Предопределенные переменные, связанные со случайными значениями

Имя	Значение
fRandomFraction1PerPass	Первое случайное значение для данного прохода
fRandomFraction2PerPass	Второе случайное значение для данного прохода
fRandomFraction3PerPass	Третье случайное значение для данного прохода
fRandomFraction4PerPass	Четвертое случайное значение для данного прохода
fRandomFraction1PerEffect	Первое случайное значение для данного эффекта
fRandomFraction2PerEffect	Второе случайное значение для данного эффекта
fRandomFraction3PerEffect	Третье случайное значение для данного эффекта
fRandomFraction4PerEffect	Четвертое случайное значение для данного эффекта

Предопределенная вещественная переменная *fPassIndex* содержит номер текущего прохода рендеринга (pass).

В табл. 15.4 приведены предопределенные переменные, связанные с мышью, в табл. 15.5 — переменные, задающие параметры вида и модели.

Таблица 15.4. Предопределенные переменные RenderMonkey, связанные с мышью

Имя	Значение
fMouseButtonStateLeft	Вещественная переменная, принимающая значение 1.0, если левая кнопка мыши нажата, и 0.0 в противном случае
fMouseButtonStateMiddle	Вещественная переменная, принимающая значение 1.0, если средняя кнопка мыши нажата, и 0.0 в противном случае
fMouseButtonStateRight	Вещественная переменная, принимающая значение 1.0, если правая кнопка мыши нажата, и 0.0 в противном случае
fMouseCoordx	Текущая x-координата курсора мыши
fMouseCoordy	Текущая y-координата курсора мыши
fMouseCoordx_NDC	Текущая x-координата курсора мыши в нормированных координатах устройства (Normalized Device Coordinates, NDC) — окончательных координатах, получаемых после перспективного деления
fMouseCoordy_NDC	Текущая y-координата курсора мыши в нормированных координатах устройства (NDC)
fMouseButtonState4f	Четырехмерный вектор, построенный из переменных fMouseButtonStateLeft, fMouseButtonStateMiddle и fMouseButtonStateRight
fMouseCoords4f	Четырехмерный вектор, построенный из текущих координат курсора мыши
fMouseCoordsXY	Текущие координаты курсора мыши в виде двухмерного вектора
fMouseCoordsXY_NDC	Текущие координаты курсора мыши (в нормированных координатах устройства, NDC) в виде двухмерного вектора

Таблица 15.5. Предопределенные переменные RenderMonkey, задающие параметры вида и модели

Имя	Значение
vViewDirection	Четырехмерный вектор, определяющий направление взгляда для камеры
vViewPosition	Четырехмерный вектор, задающий текущее положение камеры

Таблица 15.5 (окончание)

Имя	Значение
vViewSide	Четырехмерный вектор, задающий направление вбок для камеры
vViewUp	Четырехмерный вектор, задающий направление вверх для камеры
vModelBBoxTopLeft	Трехмерный вектор — координаты верхнего левого угла прямоугольного параллелепипеда (Axis Aligned Bounding Box, AABB), описанного вокруг модели, для которой выполняется шейдер
vModelBBoxBottomRight	Трехмерный вектор — координаты нижнего правого угла прямоугольного параллелепипеда (AABB), описанного вокруг модели, для которой выполняется шейдер
vModelBBoxCenter	Трехмерный вектор — центр модели прямоугольного параллелепипеда (AABB), описанного вокруг модели, для которой выполняется шейдер
vModelBoundingSphereCenter	Трехмерный вектор — центр окружности, описанной вокруг модели, для которой выполняется шейдер
vModelBoundingSphereRadius	Радиус окружности, описанной вокруг модели, для которой выполняется шейдер

Кроме этих скалярных и векторных величин в RenderMonkey есть набор предопределенных матричных переменных 4×4 , добавляемых с помощью команды **Add Variable \ Matrix \ Predefined** контекстного меню узла.

С помощью команды **Add Variable \ Float \ Predefined \ fTime0_X** контекстного меню узла **Pass 0** добавим новую переменную, представляющую время (рис. 15.23).

С помощью этой переменной мы можем "оживить" материал, например, заставить красный цвет то появляться, то исчезать (листинг 15.4). Если к этому подключить уже полученное значение из текстуры **noiseMap**, то это появление будет не таким регулярным.

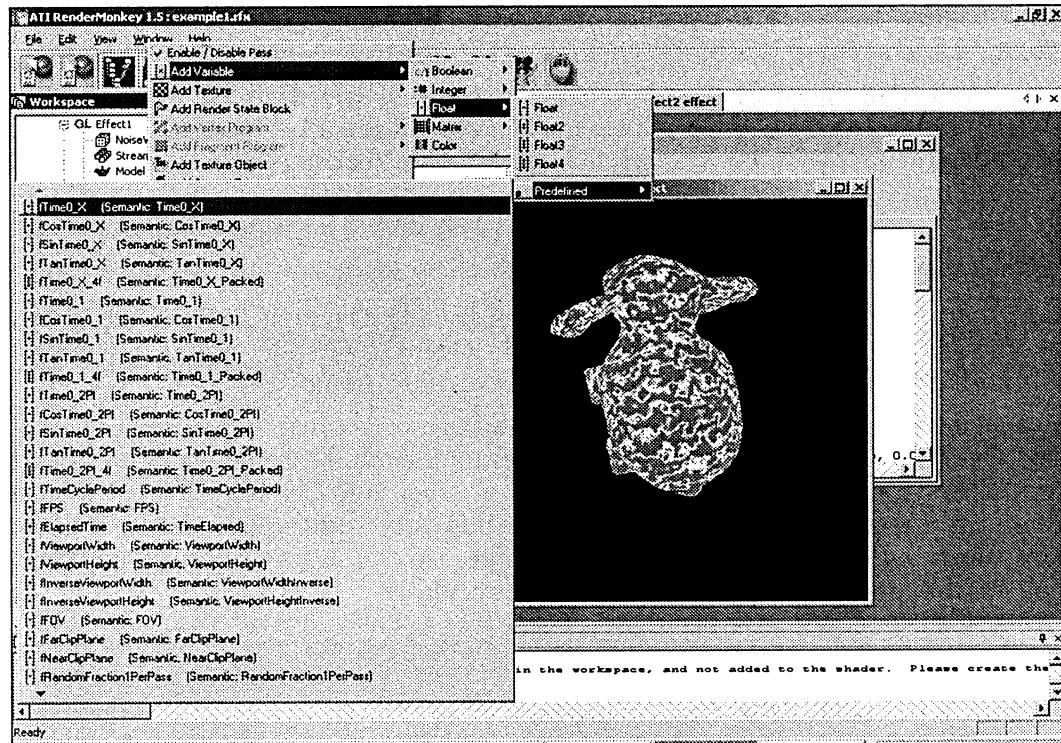


Рис. 15.23. Добавление предопределенной переменной

Листинг 15.4. Фрагментный шейдер для анимированной текстуры

```

uniform sampler3D noiseMap;
uniform float freqScale;
uniform float noiseScale;
uniform float redScale;
uniform float redPow;
uniform float fTime0_X;
uniform vec4 c1;
uniform vec4 c2;
varying vec4 pos;

void main(void)
{
    vec3 noise = 0.07*pos.x + noiseScale*texture3D( noiseMap, 0.07*pos );
    float t = 0.5 * ( 1.0 + sin( noise.x * 3.1415926 *

```

```

        freqScale ) );
float tScale = 1.0 + sin ( fTime0_X * 3.1415926 + noise.y -
                           5.0 * noise.z );
float t2      = clamp (0.5 * tScale * pow(t * redScale, redPow), 0, 1);

gl_FragColor = mix ( c1, c2, t*t*t ) + vec4 ( 1, 0, 0, 1 ) * t2;
}

```

Еще одной удобной возможностью RenderMonkey является добавление к узлу прохода (Pass) узла параметров состояния OpenGL (Render State) при помощи команды **Add Render State Block** контекстного меню.

Блок параметров состояния отвечает за многочисленные параметры состояния OpenGL, такие как параметры смешения цветов, разрешение/запрещение отдельных буферов, размеры точки и т. п. Двойной щелчок на узле **Render State** открывает окно настройки параметров состояния (рис. 15.24), в котором, щелкнув на значении в столбце **Value**, можно задать один из параметров состояния для использования при рендеринге.

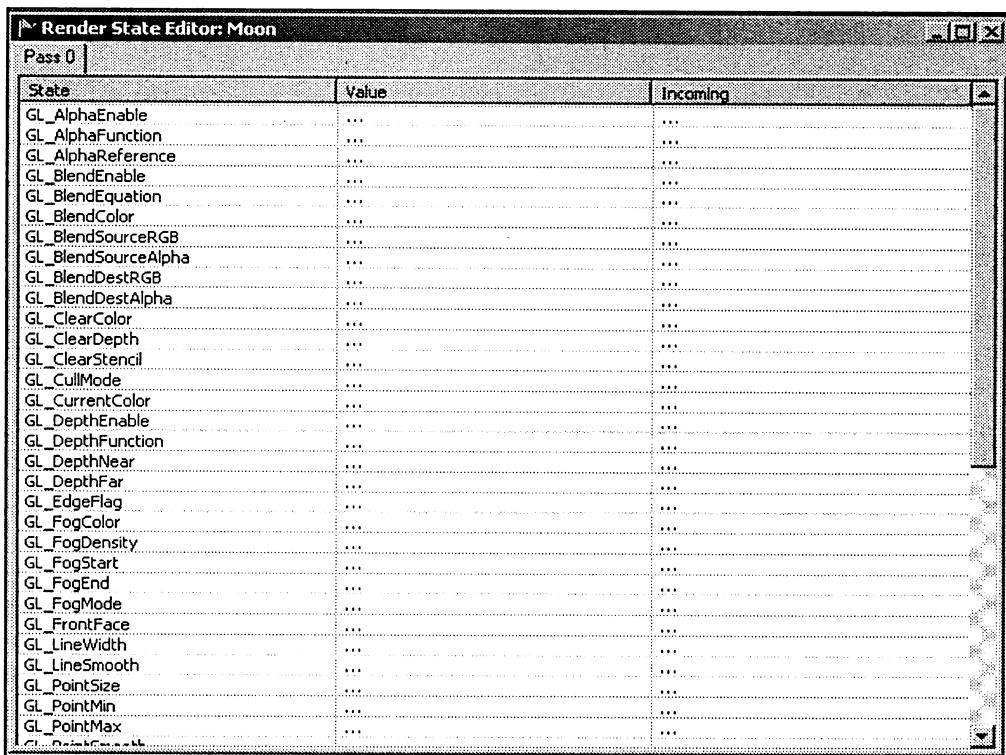


Рис. 15.24. Окно настройки параметров состояния OpenGL

Таким способом можно задать режим смешения цветов для конкретного прохода, режимы записи и т. п.

Кроме обычных текстур, в проект можно включить текстуру, в которую можно осуществлять рендеринг (Renderable Texture). Для этого предназначена команда **Add Texture \ Add Renderable Texture** всплывающего меню узла. После этого в текущем узле появляется текстура, которая может служить целью для рендеринга в одном из проходов.

Также в любом из проходов можно выбрать команду **Add Render Target** всплывающего меню для перенаправления результатов рендеринга данного прохода. Это добавляет к данному проходу узел рендеринга в текстуру, во всплывающем меню которого следует выбрать команду **Reference Node** и в ее подменю выбрать одну из текстур, в которую можно осуществлять рендеринг.

Рассмотрим использование рендеринга в текстуру на примере создания ореола вокруг объекта. Стандартный способ для этого — рендеринг объекта белым цветом в текстуру (изначально черную), после чего к данному изображению применяется эффект размытия (blur). Далее это "размытое" изображение можно наложить поверх уже нормально выведенного объекта.

Для этого проекта (рис. 15.25) нам понадобятся две камеры — первая соответствует проходу наложения размытого изображения на результаты рендеринга (на основе модели ScreenAlignedQuad.3ds), а вторая предназначена для рендеринга основной модели.

Также понадобятся две модели: первая — ScreenAlignedQuad.3ds, а вторая — объект, вокруг которого будем строить размытие (например, Skull.3ds).

Также в эффект необходимо добавить текстуру, в которую будет осуществляться рендеринг (команда **Add Texture \ Add Renderable Texture** всплывающего меню).

В первом проходе мы добавляем узел **Camera Reference** (команда **Add Camera Reference**), выбрав в качестве камеры основную камеру для рендеринга (**Camera**).

Также в этом проходе добавляется узел **Render Target** и назначается добавленная к эффекту текстура для рендеринга.

Второй проход практически аналогичен первому, только в качестве цели для рендеринга использует экран (т. е. в нем отсутствует узел **Render Target**).

Последним идет узел, "размывающий" текстуру с изображением исходного объекта и накладывающий ее на изображение, полученное в предыдущем проходе. При этом данный узел должен накладываться на предыдущее изображение с использованием смешения цветов. Для обеспечения этого в последний проход необходимо добавить объект **Render State Block**.

Чтобы не писать сложных процедур размытия (они будут рассмотрены в гл. 17), ограничимся простым использованием возможностей пирамидального

фильтрования — если в функции `texture2D` задать для параметра `bias` значение больше нуля, то изображение будет взято из одного из уровней пирамиды, полученных именно путем размытия основной текстуры.

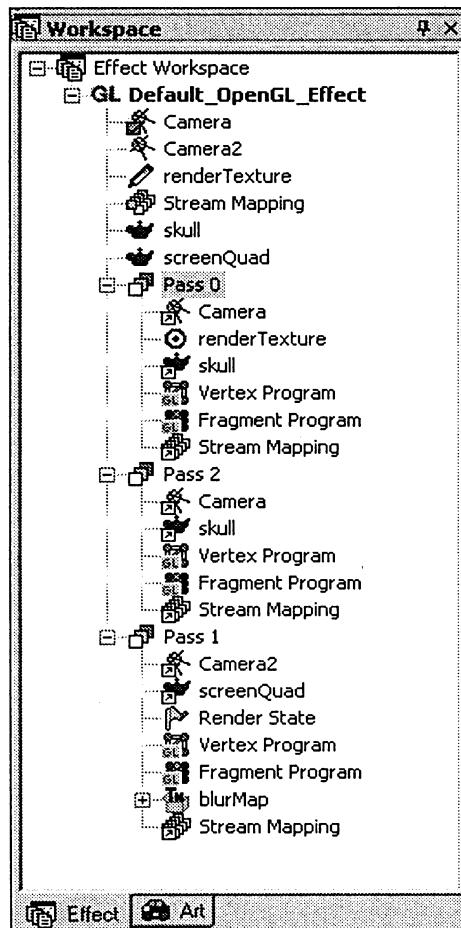


Рис. 15.25. Эффект вывода объекта с добавлением ореола

Этот эффект реализуется фрагментным шейдером, приведенным в листинге 15.5.

Листинг 15.5. Фрагментный шейдер для создания ореола вокруг объекта

```
uniform sampler2D blurMap;

void main(void)
{
```

```
vec2 tex = vec2 ( 1.0, -1.0 ) * gl_TexCoord [0].xy;
    // leave only black
vec3 clr = texture2D ( blurMap, tex ).rgb;

if ( all ( greaterThan ( clr, vec3 ( 0.5 ) ) ) )
    discard;
}

gl_FragColor = 1.3 * texture2D ( blurMap, tex, 3 );
}
```

Обратите внимание на условный оператор, отбрасывающий белые точки, — задачей данного шейдера является именно получение ореола, т. е. точек, получивших свой цвет в результате размытия исходного изображения, а не точек самого размыываемого объекта.

На рис. 15.26 изображена накладываемая текстура (та, которая размывается и используется для создания ореола).



Рис. 15.26. Текстура, используемая для размытия и создания ореола

Для точного наложения ореола на объект может понадобиться ручная настройка камер (например понадобится настройка камеры, используемой для рендеринга модели SCREENALIGNEDQUAD.3DS, чтобы ее изображение заняло все окно рендеринга). Для этого вызовите всплывающее меню одной из камер и выберите в нем пункт **Set Active Camera** для назначения данной камеры активной, т. е. камерой, параметры которой будут изменяться. Настроив одну камеру, сделаем активной вторую и тоже настроим ее.

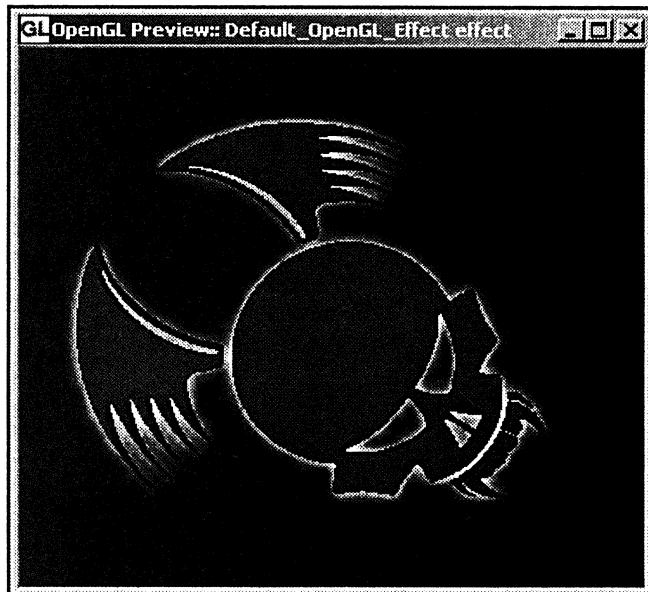
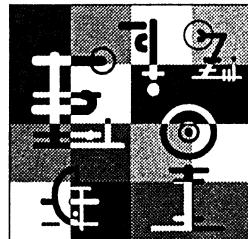


Рис. 15.27. Объект с размытием

Все рассмотренные проекты вы найдете на сопроводительном компакт-диске книги в каталоге Code\Chapter-15.



Глава 16

Использование основных моделей освещения, моделирование преломления и дифракции, обработка изображений на GLSL

В предыдущих главах мы рассмотрели как сам язык GLSL, так и ряд моделей освещения вместе с их реализаций на GLSL. В этой главе представлены различные шейдеры, начиная с основных моделей освещения и заканчивая спецэффектами и обработкой изображения. Приведены и соответствующие программы на C++, использующие данные шейдеры.

Использование шейдеров из главы 13

В главе 13 были приведены вершинные и фрагментные шейдеры для ряда распространенных моделей освещения. Рассмотрим теперь, каким образом можно использовать эти шейдеры для рендеринга различных объектов из программ на языке C++.

Для работы с шейдерами мы будем использовать класс `GlslProgram`, рассмотренный в главе 14.

В листинге 16.1 приведен исходный код программы на C++, осуществляющей рендеринг одного из стандартных объектов библиотеки GLUT — "Чайника" — с использованием модели освещения Фонга. При этом вокруг объекта вращается источник света и сам объект можно поворачивать при помощи мыши.

Листинг 16.1. Рендеринг "Чайнника" (модель освещения Фонга)

```
//  
// Example of GLSL Phong shader.  
  
#include "libExt.h"
```

```
#include <glut.h>
#include <stdio.h>
#include <stdlib.h>
#include "libTexture.h"
#include "Vector3D.h"
#include "Vector2D.h"
#include "Vector4D.h"
#include "Data.h"
#include "GlslProgram.h"

Vector3D eye ( 7, 5, 7 );           // camera position
Vector3D light ( 5, 0, 4 );         // light position
float angle = 0;
Vector3D rot ( 0, 0, 0 );
int mouseOldX = 0;
int mouseOldY = 0;

GlslProgram program;

void init ()
{
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );
    glEnable ( GL_DEPTH_TEST );
    glDepthFunc ( GL_LEQUAL );

    glHint ( GL_POLYGON_SMOOTH_HINT, GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}

void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // draw the light
    program.unbind ();

    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();

    glTranslatef ( light.x, light.y, light.z );
```

```
glColor4f      ( 1, 1, 1, 1 );
glutSolidSphere ( 0.1f, 15, 15 );
glPopMatrix     ();

glMatrixMode   ( GL_MODELVIEW );
glPushMatrix    ();

glRotatef      ( rot.x, 1, 0, 0 );
glRotatef      ( rot.y, 0, 1, 0 );
glRotatef      ( rot.z, 0, 0, 1 );

program.bind   ();
glutSolidTeapot ( 2.5 );

program.unbind ();

glPopMatrix ();
glutSwapBuffers ();

}

void reshape ( int w, int h )
{
    glViewport      ( 0, 0, (GLsizei)w, (GLsizei)h );
    glMatrixMode   ( GL_PROJECTION );
        // factor all camera ops into
        // projection matrix
    glLoadIdentity ();
    gluPerspective ( 60.0, (GLfloat)w/(GLfloat)h, 1.0, 60.0 );
    gluLookAt       ( eye.x, eye.y, eye.z, // eye
                      0, 0, 0,           // center
                      0.0, 0.0, 1.0 ); // up

    glMatrixMode   ( GL_MODELVIEW );
    glLoadIdentity ();
}

void motion ( int x, int y )
{
    rot.y -= ((mouseOldY - y) * 180.0f) / 200.0f;
```

```
rot.z -= ((mouseOldX - x) * 180.0f) / 200.0f;
rot.x = 0;

if ( rot.z > 360 )
    rot.z -= 360;

if ( rot.z < -360 )
    rot.z += 360;

if ( rot.y > 360 )
    rot.y -= 360;

if ( rot.y < -360 )
    rot.y += 360;

mouseOldX = x;
mouseOldY = y;

glutPostRedisplay ();
}

void mouse ( int button, int state, int x, int y )
{
    if ( state == GLUT_DOWN )
    {
        mouseOldX = x;
        mouseOldY = y;
    }
}

void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' ) // quit requested
        exit ( 0 );
}

void animate ()
{
    angle = 0.004f * glutGet ( GLUT_ELAPSED_TIME );
```

```
light.x = 2*cos ( angle );
light.y = 3*sin ( 1.4 * angle );
light.z = 3 + 0.5 * sin ( angle / 3 );

program.bind ();
program.setUniformVector ( "eyePos", Vector4D ( eye, 1 ) );
program.setUniformVector ( "lightPos", Vector4D ( light, 1 ) );
program.unbind ();

glutPostRedisplay ();
}

int main ( int argc, char * argv [] )
{
    // initialize glut
    glutInit      ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize ( 500, 500 );

    // create window
    glutCreateWindow ( "Example of GLSL Phong shader" );

    // register handlers
    glutDisplayFunc ( display );
    glutReshapeFunc ( reshape );
    glutKeyboardFunc ( key );
    glutMouseFunc   ( mouse );
    glutMotionFunc  ( motion );
    glutIdleFunc    ( animate );

    init          ();
    initExtensions ();
    printfInfo    ();

    assertExtensionsSupported (
        "GL_ARB_shading_language_100 GL_ARB_shader_objects" );

    if ( !program.loadShaders ( "phong.vsh", "phong.fsh" ) )
```

```

{
    printf ( "Error loading shaders:\n%s\n",
             program.getLog ().c_str () );

    return 3;
}

glutMainLoop ();

return 0;
}

```

Как видно из приведенного листинга, все, что требуется для рендеринга модели с использованием шейдеров, — это загрузка соответствующих шейдеров (с помощью метода `loadShaders`) и установка соответствия текстурных блоков именам сэмплеров (с помощью метода `setTexture`). После этого для каждого кадра задаются положения источника света и положение наблюдателя (с помощью метода `setUniformVector`), непосредственно сам рендеринг объекта заключается между вызовами функций `bind` и `unbind`.

Аналогичным образом реализуется рендеринг с использованием остальных рассмотренных шейдеров. Далее приведен код только рендеринга тора с использованием карты нормалей (листинг 16.2). Код для остальных примеров находится на сопроводительном компакт-диске книги в каталоге `Code\Chapter-16`.

Всю работу с моделью в данном случае мы будем осуществлять через библиотеку `libMesh` (рассмотренную в гл. 11). Она позволяет создать нужный объект (тор) как набор треугольных граней. При этом в каждой вершине определены не только текстурные координаты, но и вектор нормали, касательный вектор и бинормаль (т. е. полный базис касательного пространства).

Класс `Mesh` автоматически помещает все эти данные в вершинный массив (`Vertex Buffer Objects`, VBO), что позволяет избежать затрат на постоянную передачу всех этих данных GPU при каждом рендеринге тора. Однако для использования шейдеров из листингов 13.5 и 13.6 необходимо осуществить правильное назначение данных различным вершинам модели, например так:

```
torus = makeTorus ( 1, 3, 30, 30 );
```

```

torus -> addCoordAssignment ( Mesh :: tagVertex,      Mesh :: tagVertex );
torus -> addCoordAssignment ( Mesh :: tagNormal,     Mesh :: tagNormal );
torus -> addCoordAssignment ( Mesh :: tagTexCoord,   Mesh :: tagTex0   );
torus -> addCoordAssignment ( Mesh :: tagTangent,    Mesh :: tagTex1   );

```

```
torus -> addCoordAssignment ( Mesh :: tagBinormal, Mesh :: tagTex2 ) ;
torus -> createBuffers ();
```

При таком назначении и настройке текстур текстурных блоков рендеринг тора оказывается очень похожим на уже рассмотренный случай.

Листинг 16.2. Исходный код на C++ для рендеринга тора с использованием карты нормалей

```
//  
// Sample showing how to use bump mapping in GLSL programs  
  
#include "libExt.h"  
#include <glut.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include "libTexture.h"  
#include "Vector3D.h"  
#include "Vector2D.h"  
#include "Vector4D.h"  
#include "Data.h"  
#include "GlslProgram.h"  
#include "Mesh.h"  
#include "MeshUtils.h"  
  
Vector3D eye ( 7, 5, 7 );           // camera position  
Vector3D light ( 5, 0, 4 );         // light position  
float angle = 0;  
Vector3D rot ( 0, 0, 0 );  
int mouseOldX = 0;  
int mouseOldY = 0;  
unsigned diffuseMap;  
unsigned bumpMap;  
Mesh * torus;  
GlslProgram program;  
  
void init ()  
{  
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );  
    glEnable ( GL_DEPTH_TEST );
```

```
glDepthFunc ( GL_LESS );  
  
glHint ( GL_Polygon_SMOOTH_HINT, GL_NICEST );  
glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );  
}  
  
void display ()  
{  
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );  
  
    // draw the light  
    program.unbind ();  
  
    glMatrixMode ( GL_MODELVIEW );  
    glPushMatrix ();  
  
    glTranslatef ( light.x, light.y, light.z );  
    glColor4f ( 1, 1, 1, 1 );  
    glutSolidSphere ( 0.1f, 15, 15 );  
    glPopMatrix ();  
  
    glMatrixMode ( GL_MODELVIEW );  
    glPushMatrix ();  
  
    glRotatef ( rot.x, 1, 0, 0 );  
    glRotatef ( rot.y, 0, 1, 0 );  
    glRotatef ( rot.z, 0, 0, 1 );  
  
    glActiveTextureARB ( GL_TEXTURE0_ARB );  
    glBindTexture ( GL_TEXTURE_2D, bumpMap );  
  
    glActiveTextureARB ( GL_TEXTURE1_ARB );  
    glBindTexture ( GL_TEXTURE_2D, diffuseMap );  
  
    program.bind ();  
    torus->render ();  
    program.unbind ();  
  
    glPopMatrix ();
```

```
glutSwapBuffers ();
}

void reshape ( int w, int h )
{
    glViewport      ( 0, 0, (GLsizei)w, (GLsizei)h );
    glMatrixMode   ( GL_PROJECTION );
        // factor all camera ops into
        // projection matrix
    glLoadIdentity ();
    gluPerspective ( 60.0, (GLfloat)w/(GLfloat)h, 1.0, 60.0 );
    gluLookAt       ( eye.x, eye.y, eye.z, // eye
                      0, 0, 0,           // center
                      0.0, 0.0, 1.0 ); // up

    glMatrixMode   ( GL_MODELVIEW );
    glLoadIdentity ();
}

void motion ( int x, int y )
{
    rot.y -= ((mouseOldY - y) * 180.0f) / 200.0f;
    rot.z -= ((mouseOldX - x) * 180.0f) / 200.0f;
    rot.x = 0;

    if ( rot.z > 360 )
        rot.z -= 360;

    if ( rot.z < -360 )
        rot.z += 360;

    if ( rot.y > 360 )
        rot.y -= 360;

    if ( rot.y < -360 )
        rot.y += 360;

    mouseOldX = x;
    mouseOldY = y;
```

```
glutPostRedisplay ();  
}  
  
void mouse ( int button, int state, int x, int y )  
{  
    if ( state == GLUT_DOWN )  
    {  
        mouseOldX = x;  
        mouseOldY = y;  
    }  
}  
  
void key ( unsigned char key, int x, int y )  
{  
    if ( key == 27 || key == 'q' || key == 'Q' ) // quit requested  
        exit ( 0 );  
}  
  
void animate ()  
{  
    angle = 0.004f * glutGet ( GLUT_ELAPSED_TIME );  
  
    light.x = 2*cos ( angle );  
    light.y = 2*sin ( angle );  
    light.z = 3 + 0.3 * sin ( angle / 3 );  
  
    program.bind ();  
    program.setUniformVector ( "eyePos", Vector4D ( eye, 1 ) );  
    program.setUniformVector ( "lightPos", Vector4D ( light, 1 ) );  
    program.unbind ();  
  
    glutPostRedisplay ();  
}  
  
int main ( int argc, char * argv [] )  
{  
    // initialize glut  
    glutInit ( &argc, argv );  
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
```

```
glutInitWindowSize ( 500, 500 );

        // create window
glutCreateWindow (
    "More complex example of GLSL specular perpixel shader" );

        // register handlers
glutDisplayFunc ( display );
glutReshapeFunc ( reshape );
glutKeyboardFunc ( key );
glutMouseFunc     ( mouse );
glutMotionFunc   ( motion );
glutIdleFunc     ( animate );

init         ();
initExtensions ();

assertExtensionsSupported (
    "GL_ARB_shading_language_100 GL_ARB_shader_objects" );

if ( !program.loadShaders ( "bump.vsh", "bump.fsh" ) )
{
    printf ( "Error loading shaders:\n%s\n",
            program.getLog ().c_str () );

    return 3;
}

diffuseMap = createTexture2D ( true, "../../../Textures/oak.bmp" );
bumpMap    = createTexture2D ( false,
                            "../../../Textures/Bumpmaps/normal2.bmp" );

        // install program object as part of
        // current state
program.bind ();

program.setTexture ( "bumpMap",      0 );
program.setTexture ( "diffuseMap", 1 );
```

```

torus = makeTorus ( 1, 3, 30, 30 );

torus -> addCoordAssignment ( Mesh :: tagVertex,
                               Mesh :: tagVertex );
torus -> addCoordAssignment ( Mesh :: tagNormal,
                               Mesh :: tagNormal );
torus -> addCoordAssignment ( Mesh :: tagTexCoord,
                               Mesh :: tagTex0 );
torus -> addCoordAssignment ( Mesh :: tagTangent,
                               Mesh :: tagTex1 );
torus -> addCoordAssignment ( Mesh :: tagBinormal,
                               Mesh :: tagTex2 );
torus -> createBuffers
();

glutMainLoop ();

return 0;
}

```

Использование шейдеров для анимации

Все рассмотренные ранее примеры использовали шейдеры для вычисления освещенности поверхностей объектов, не изменяя самой геометрии объектов. Однако за счет использования вершинных шейдеров можно реализовать анимацию объектов, т. е. изменять их геометрию непосредственно на стадии рендеринга, точнее, на стадии T&L (Transform & Lighting).

Простейшим примером этого является использование шейдеров для анимации *системы частиц* (particle system).

Пусть у нас есть источник частиц, равномерно создающий новые частицы, которые после создания двигаются под действием силы тяжести. Каждая такая частица может быть охарактеризована начальным положением, скоростью и моментом создания. После этого всегда можно определить положение частицы для любого текущего момента времени t по следующей формуле:

$$p(t) = p_0 + v_0 \cdot (t - t_0) + 0,5 a \cdot (t - t_0)^2. \quad (16.1)$$

В формуле (16.1) параметрами p_0 , v_0 и t_0 обозначены начальное положение частицы, начальная скорость частицы и момент ее создания (рождения). Параметром a обозначен вектор ускорения свободного падения, т. е. $(0, 0, -g)$.

Если с каждой частицей связать набор (p_0, v_0, t_0) , то расчет текущего положения частицы можно производить при помощи вершинного шейдера — достаточно для каждой частицы передать ее начальные параметры и текущее

время. При этом, поскольку для каждой частицы параметры (p_0, v_0, t_0) не изменяются с течением времени, с одной стороны, не требуются ресурсы для пересылки данных GPU, а с другой, нет необходимости тратить время CPU на расчет положения частицы в каждый момент времени, т. к. все это берет на себя вершинная программа.

Тем самым мы получаем выигрыш как в передаче данных, так и в экономии времени CPU.

В листинге 16.3 приведен вершинный шейдер, осуществляющий анимацию частицы по ее начальным данным. При этом значения (v_0, t_0) передаются в виде четырехмерного вектора текстурных координат для первого текстурного блока.

Листинг 16.3. Простейший вершинный шейдер для анимации системы частиц

```
uniform float time;
varying vec4 color;

void main(void)
{
    vec3 vel      = 3*gl_MultiTexCoord1.xyz;
    vec3 pos      = gl_Vertex.xyz;
    float phase = gl_MultiTexCoord1.w;
    float t      = time - phase;

    // now compute new position using t and org
    vec3 org      = pos + t*vel - 0.5*t*t*vec3 ( 0.0, 0.0, 1.0 );

    gl_Position     = gl_ModelViewProjectionMatrix * vec4 (org, 1.0);
    gl_TexCoord [0] = gl_MultiTexCoord0;
}
```

Предложенное решение можно еще упростить — пока что центральный процессор должен как создавать новые частицы, так и уничтожать те частицы, время жизни которых уже закончено. Хотя вершинные шейдеры и не могут создавать новые вершины или уничтожать уже существующие (т. е. изменять топологию объекта), в данном случае можно обойтись и без этого.

Будем считать, что у всех частиц продолжительность жизни одинакова (и равна T), причем, как только одна частица уничтожается, сразу же создается новая, т. е. времена создания частиц являются кратными T величинами.

Если считать, что новая частица создается с теми же начальными положением и скоростью, что и только что уничтоженная, то процесс рождения

новой частицы (и гибели старой) заключается лишь в увеличении момента рождения частицы на T .

Исходя из этого можно вообще не изменять набор атрибутов частиц, создав их однажды, а для вычисления текущего положения частицы использовать формулу (16.2):

$$p(t) = p_0 + v_0 \cdot \text{mod}(t - t_0, T) + 0,5 a \cdot \text{mod}(t - t_0, T)^2. \quad (16.2)$$

Примечание

Функция $\text{mod}(t, T)$ возвращает остаток от деления t на T .

Тогда работа CPU вообще сводится к минимуму — создать один раз набор начальных параметров частиц, поместить их в вершинный массив и просто выводить его, передавая для каждого кадра текущее время через `uniform`-переменную.

В листинге 16.4 приведена реализация соответствующего вершинного шейдера.

Листинг 16.4. Вершинный шейдер для анимации системы частиц

```
//  
// Simple particle system vertex shader  
//  
  
uniform float time;  
varying vec4 color;  
  
void main(void)  
{  
    vec3 vel      = 3*gl_MultiTexCoord1.xyz;  
    vec3 pos      = gl_Vertex.xyz;  
    float phase  = gl_MultiTexCoord1.w;  
    float t       = mod ( time + phase, 4.0 );  
  
    // now compute new position using t and org  
    vec3 org = pos + t*vel - 0.5*t*t*vec3 ( 0.0, 0.0, 1.0 );  
  
    color = mix ( vec4 ( 1.0, 0.0, 0.0, 1.0 ),  
                 vec4 ( 0.3, 0.3, 0.0, 1.0 ), t * 0.25 );  
    gl_Position     = gl_ModelViewProjectionMatrix * vec4 (org, 1.0);  
    gl_TexCoord [0] = gl_MultiTexCoord0;  
}
```

Если мы хотим использовать расширение ARB_point_sprite [1], то следует внести коррекции в фрагментный шейдер для правильного вывода изображения частицы с использованием текстуры (листинг 16.5).

Листинг 16.5. Фрагментный шейдер для вывода анимированной системы частиц

```
//  
// Simple particle system fragment shader  
  
//  
  
varying vec4 color;  
uniform sampler2D decalMap;  
  
void main (void)  
{  
    gl_FragColor = color * texture2D ( decalMap, gl_TexCoord [0].xy );  
}
```

Соответствующий код на C++ приведен в листинге 16.6 (обратите внимание, что здесь данные не помещаются в вершинный массив или дисплейный список, хотя это возможно).

Листинг 16.6. Программа на C++ для построения системы частиц, анимированной на GPU

```
//  
// Sample showing how to use GLSL programs to create particle systems  
  
//  
#include "libExt.h"  
#include <glut.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
#include "libTexture.h"  
#include "Vector3D.h"  
#include "Vector2D.h"  
#include "Vector4D.h"  
#include "GlslProgram.h"  
  
#define NUM_PARTICLES 5000 // total number of particles
```

```
Vector3D eye ( 7, 5, 7 );           // camera position
float angle = 0;
Vector3D rot ( 0, 0, 0 );
int mouseOldX = 0;
int mouseOldY = 0;
unsigned decalMap;

Vector4D partAttr [NUM_PARTICLES]; // particle attributes
// (initial xyz and phase)

GlslProgram program;

void init ()
{
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );
    glEnable ( GL_DEPTH_TEST );
    glDepthFunc ( GL_LEQUAL );

    glHint ( GL_POLYGON_SMOOTH_HINT, GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}

void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glPushAttrib ( GL_ENABLE_BIT | GL_POINT_BIT | GL_COLOR_BUFFER_BIT
        | GL_DEPTH_BUFFER_BIT );

    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();

    glRotatef ( rot.x, 1, 0, 0 );
    glRotatef ( rot.y, 0, 1, 0 );
    glRotatef ( rot.z, 0, 0, 1 );

    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glBindTexture ( GL_TEXTURE_2D, decalMap );

    glDepthMask ( GL_FALSE );
```

```
glEnable      ( GL_BLEND );
glBlendFunc  ( GL_ONE, GL_ONE );
glPointSize ( 5 );

float quadratic [] = { 1.0f, 0.0f, 0.01f };

glEnable          ( GL_POINT_SPRITE_ARB );
glPointParameterfvARB ( GL_POINT_DISTANCE_ATTENUATION_ARB,
                        quadratic );
glPointParameterfARB ( GL_POINT_FADE_THRESHOLD_SIZE_ARB, 20.0f );

glTexEnvf ( GL_POINT_SPRITE_ARB, GL_COORD_REPLACE_ARB, GL_TRUE );

program.bind ();

glBegin      ( GL_POINTS );

for ( int i = 0; i < NUM_PARTICLES; i++ )
{
    glMultiTexCoord4fv ( GL_TEXTURE1_ARB, partAttr [i] );
    glVertex3fv       ( partAttr [i] );
}

glEnd ();

program.unbind ();

glPopAttrib ();
glPopMatrix ();

glutSwapBuffers ();
}

void reshape ( int w, int h )
{
    glViewport      ( 0, 0, (GLsizei)w, (GLsizei)h );
    glMatrixMode   ( GL_PROJECTION );
        // factor all camera ops into
        // projection matrix
```

```
glLoadIdentity ();
gluPerspective ( 60.0, (GLfloat)w/(GLfloat)h, 1.0, 60.0 );
gluLookAt      ( eye.x, eye.y, eye.z, // eye
                  0, 0, 0,           // center
                  0.0, 0.0, 1.0 );   // up

glMatrixMode   ( GL_MODELVIEW );
glLoadIdentity ();
}

void motion ( int x, int y )
{
    rot.y -= ((mouseOldY - y) * 180.0f) / 200.0f;
    rot.z -= ((mouseOldX - x) * 180.0f) / 200.0f;
    rot.x = 0;

    if ( rot.z > 360 )
        rot.z -= 360;

    if ( rot.z < -360 )
        rot.z += 360;

    if ( rot.y > 360 )
        rot.y -= 360;

    if ( rot.y < -360 )
        rot.y += 360;

    mouseOldX = x;
    mouseOldY = y;

    glutPostRedisplay ();
}

void mouse ( int button, int state, int x, int y )
{
    if ( state == GLUT_DOWN )
    {
        mouseOldX = x;
```

```
mouseOldY = y;
}

}

void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' ) // quit requested
        exit ( 0 );
}

void animate ()
{
    float time = 0.001f * glutGet ( GLUT_ELAPSED_TIME );

    program.bind ();
    program.setUniformFloat ( "time", time );
    program.unbind ();

    glutPostRedisplay ();
}

int main ( int argc, char * argv [] )
{
    // initialize glut
    glutInit ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize ( 500, 500 );

    // create window
    glutCreateWindow ( "Particle animation via GLSL shader" );

    // register handlers
    glutDisplayFunc ( display );
    glutReshapeFunc ( reshape );
    glutKeyboardFunc ( key );
    glutMouseFunc ( mouse );
    glutMotionFunc ( motion );
    glutIdleFunc ( animate );

    init ();
}
```

```
initExtensions ();

assertExtensionsSupported (
    "GL_ARB_shading_language_100 GL_ARB_shader_objects" );

if ( !program.loadShaders ( "particles.vsh", "particles.fsh" ) )
{
    printf ( "Error loading shaders:\n%s\n",
        program.getLog ().c_str () );

    return 3;
}

decalMap = createTexture2D ( true, "../../../Textures/Fire.bmp" );

        // initialize particles
for ( int i = 0; i < NUM_PARTICLES; i++ )
{
    Vector3D xyz = Vector3D :: getRandomVector ( 0.5 );
    float      phase = 5 * (float) rand () / (float) RAND_MAX;

    partAttr [i] = Vector4D ( xyz, phase );
}

program.bind      ();
program.setTexture ( "decalMap", 0 );
program.unbind    ();

glutMainLoop ();

return 0;
}
```

Моделирование преломления

Одним из красивых и в то же время простых эффектов, легко моделируемых при помощи шейдеров, является рендеринг прозрачных объектов с учетом преломления. Простейший вариант заключается в том, что для прозрачного объекта задается *кубическая карта окружения* (cubic environment map). Тогда

для произвольной точки на поверхности прозрачного объекта можно найти отраженный r и преломленный t векторы по вектору нормали n и единичному вектору направления на наблюдателя v (рис. 16.1).

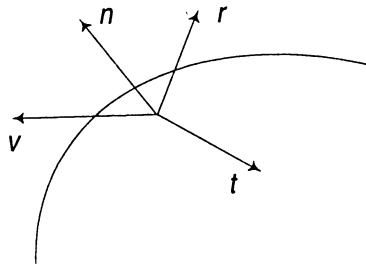


Рис. 16.1. Отраженный и преломленный векторы

Оба этих вектора (r и t) можно использовать для доступа к кубической карте окружения и затем объединить получившиеся цвета для получения итогового цвета. Естественно, что это лишь очень грубое приближение, однако во многих случаях оно дает вполне приемлемые результаты (рис. 16.2).



Рис. 16.2. Изображение преломляющего объекта

Для правильного смешения отраженного и преломленного цветов (C_{refl} и C_{refr}) следует использовать вводимый в курсе физики коэффициент Френеля (Fresnel coefficient). Данный коэффициент $F(\theta, \lambda)$ зависит от угла падения θ и длины волны λ и определяет долю отразившейся световой энергии. Тогда величина $1 - F(\theta, \lambda)$ задает долю преломившейся световой энергии. Тем самым данный коэффициент является параметром для смешивания отраженного и преломленного цветов.

К сожалению, зависимость коэффициента Френеля от длины волны и угла падения довольно сложна, поэтому на практике обычно используют различные приближенные способы его вычисления. Далее приведены две таких формулы, используемых на практике.

$$F = \text{clamp}(0, 1, 0, 9, |\cos \theta|), \quad (16.3)$$

$$F = r_0 + (1 - r_0) \cdot (1 - \cos \theta)^5, \quad (16.4)$$

$$F = \frac{1}{(1 + \cos \theta)^8}. \quad (16.5)$$

Параметром r_0 в формуле (16.4) обозначена величина $\left(\frac{1-\eta}{1+\eta}\right)^2$, где η — относительный коэффициент преломления одной среды относительно другой.

Используя формулу (16.3) и стандартные функции для нахождения отраженного и преломленного векторов, приходим к следующим шейдерам (листинги 16.7 и 16.8).

Листинг 16.7. Вершинный шейдер для рендеринга объекта "Стеклянный чайник" с эффектом преломления

```
//  
// Refractive glass vertex shader  
//  
uniform vec4 eyePos;  
uniform mat4 cubeMapMatrix;  
  
varying vec3 n;  
varying vec3 e;  
  
void main(void)  
{  
    vec4 pos = gl_ModelViewMatrix * gl_Vertex;  
  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

```
n = normalize ( gl_NormalMatrix * gl_Normal *
                mat3 ( cubeMapMatrix ) );
e = normalize ( ( (pos - eyePos)*cubeMapMatrix ).xyz );
}
```

Листинг 16.8. Фрагментный шейдер для рендеринга объекта "Стеклянный чайник" с эффектом преломления

```
//  
// Refractive glass fragment shader  
//  
  
uniform samplerCUBE envMap;  
  
varying vec3 n;  
varying vec3 e;  
  
void main (void)  
{  
    const float eta = 0.9;  
  
    // Compute reflection vector  
    vec3 en = normalize ( e );  
    vec3 nn = normalize ( n );  
    vec3 r = reflect ( en, nn );  
  
    // Do a lookup into the environment map.  
    vec3 envColor = textureCube ( envMap, r ).rgb;  
  
    // calc fresnels term.  
    float fresnel = abs ( dot ( en, nn ) );  
  
    fresnel = clamp ( fresnel, 0.1, 0.9 );  
  
    // calc refraction (transmitted ray)  
    vec3 t = refract ( en, nn, eta );  
  
    vec3 refractionColor = textureCube ( envMap, t ).rgb;
```

```

envColor = mix ( envColor, refractionColor, fresnel );

gl_FragColor = vec4 ( envColor, 1.0 );
}

```

Данный подход (изображение на рис. 16.2 построено при помощи этих шейдеров) хотя и выглядит красиво, но обладает одним недостатком — обычно преломление света приводит к появлению различных радугоподобных эффектов, т. е. происходит разделение света по длинам волн. Это связано с зависимостью коэффициентов Френеля от длины волны света.

Довольно простым способом моделирования такого эффекта разделения является отдельный расчет компонентов R, G и B для преломленного цвета — каждому из них сопоставляется свой коэффициент преломления, рассчитывается свой преломленный вектор, по нему извлекается свой цвет из кубической карты окружения.

Для получения итогового преломленного цвета из каждого из трех преломленных цветов берется по одному компоненту — взятые вместе, они и составляют итоговый цвет.

Для реализации этого эффекта никакие изменения в вершинном шейдере не требуются, поэтому здесь приводится лишь соответствующий фрагментный шейдер (листинг 16.9).

Листинг 16.9. Фрагментный шейдер для рендеринга преломления с разделением цветов

```

//  

// Refractive glass-2 fragment shader  

//  

uniform samplerCUBE envMap;  

varying vec3 n;  

varying vec3 e;  

void main (void)  

{  

    const vec3 eta = vec3 ( 0.9, 0.94, 0.97 );  

    // Compute reflection vector  

    vec3 en = normalize ( e );  

    vec3 nn = normalize ( n );

```

```
vec3 r = reflect ( en, nn );

// Do a lookup into the environment map.
vec3 envColor = textureCube ( envMap, r ).rgb;

// calc fresnels term
float fresnel = abs ( dot ( en, nn ) );

fresnel = clamp ( fresnel, 0.1, 0.9 );

// calc refraction (transmitted ray) for
// every component
vec3 tRed = refract ( en, nn, eta.r );
vec3 tGreen = refract ( en, nn, eta.g );
vec3 tBlue = refract ( en, nn, eta.b );

vec3 refractionColorRed = textureCube ( envMap, tRed ).rgb;
vec3 refractionColorGreen = textureCube ( envMap, tGreen ).rgb;
vec3 refractionColorBlue = textureCube ( envMap, tBlue ).rgb;
vec3 refractionColor = vec3 ( refractionColorRed.r,
                             refractionColorGreen.g,
                             refractionColorBlue.b );

envColor = mix ( envColor, refractionColor, fresnel );

gl_FragColor = vec4 ( envColor, 1.0 );
}
```

Моделирование дифракции света

Наверное, все с детства помнят мыльные пузыри, переливающиеся всеми цветами радуги. Подобные переливы видны и на поверхности компакт-диска. Появление красивых переливов цветов и в том и в другом случае связано с волновыми свойствами света, а именно — с наложением двух световых волн друг на друга.

Рассмотрим сначала случай мыльного пузыря толщиной d (рис. 16.3).

При падении света на тонкий слой отражение происходит как от внешней границы слоя (вектор r'), так и от внутренней границы (вектор r). При этом происходит наложение световых волн, отразившихся от внутренней границы, на световые волны, отразившиеся от внешней границы. Если толщина границы соизмерима с типичными длинами волн, то может произойти взаимное усиление или ослабление волн, зависящее от фазового сдвига волн. Если расстояние, пройденное волнами, кратно длине волны, то происходит их взаимное усиление. В противном случае происходит их ослабление.

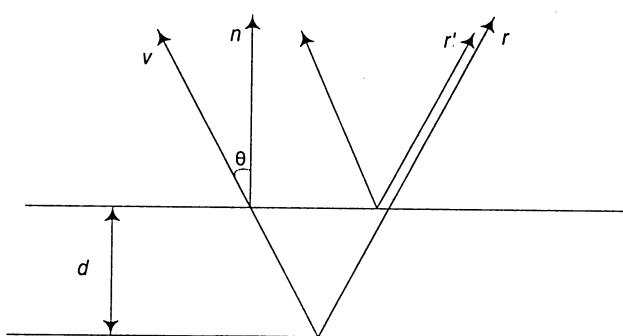


Рис. 16.3. Отражение в тонком слое

Таким образом, в пучке белого света, представляющего собой набор волн самых различных длин волн, происходит ослабление/усиление волн в зависимости от их длины волны.

Определяющим фактором здесь является разница в расстоянии, пройденном налагающимися волнами. Как видно из рис. 16.3, эта разница равна $2 \cdot d / \cos\theta$, где через θ обозначен угол падения (т. е. $\cos\theta = (n, v)$).

Данную величину можно вычислить для каждой вершины и проинтерполировать на всей грани. После этого она определяет основной цвет, для которого происходит усиление. Вместо его вычисления прямо в шейдере можно заранее рассчитать эту зависимость и запомнить ее в одномерной текстуре.

Данный подход был взят из примера из NVIDIA SDK, только здесь такая вспомогательная текстура рассчитывается отдельно при помощи скрипта (написанного на языке Python), который вы можете найти в каталоге CODE\SCRIPTS сопроводительного компакт-диска книги. Полученный из этой текстуры цвет используется для модулирования отраженного цвета, даваемого кубической картой окружения. Реализация данного подхода приведена в листингах 16.10—16.12.

Листинг 16.10. Вершинный шейдер для рендеринга объектов "Мыльный пузырь"

```
//  
// Soap bubble vertex shader  
  
uniform vec4 eyePos;  
uniform vec3 r0;           // components for R, G and B  
  
varying float viewDepth;  
varying vec3 pos;  
varying vec3 v;  
varying vec3 r;  
varying vec3 fresnel;  
  
void main()  
{  
    const float filmDepth = 0.1;  
    const vec3 one = vec3 ( 1.0 );  
  
    gl_Position = ftransform ();  
  
    // transformed point to world space  
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );  
  
    // transform normal from model-space  
    // to view-space  
    vec3 n = normalize ( gl_NormalMatrix * gl_Normal );  
  
    v      = normalize ( eyePos.xyz - p );  
    r      = reflect ( v, n );  
    viewDepth = filmDepth / dot ( n, v );  
    pos    = gl_Vertex.xyz;  
  
    // compute approximation to Fresnel  
    fresnel = r0 + (one - r0) * pow (1.0 - abs (dot (n,v)), 5.0);  
}
```

Листинг 16.11. Фрагментный шейдер для рендеринга объектов "Мыльные пузыри"

```
//  
// Soap bubble fragment shader  
  
uniform sampler1D    fringeMap;  
uniform samplerCube   reflectionMap;  
  
uniform float        time;  
varying float         viewDepth;  
varying vec3          pos;  
varying vec3          v;  
varying vec3          r;  
varying vec3          fresnel;  
  
void main()  
{  
    float depthScale = 0.4;  
  
    // lookup fringe value based on view depth  
    vec3 fringeColor = texture1D ( fringeMap,  
                                    viewDepth.x*depthScale ).rgb;  
  
    // lookup reflection in cube map  
    vec3 reflColor = textureCube ( reflectionMap, r ).rgb;  
  
    // modulate reflection by fringe color  
    gl_FragColor = vec4 ( reflColor * fringeColor, 0.4 );  
}
```

Листинг 16.12. Код на C++ для рендеринга мыльных пузырей

```
#include    "libExt.h"  
#include    <glut.h>  
#include    <stdio.h>  
#include    <stdlib.h>  
#include    "libTexture.h"  
#include    "TypeDefs.h"
```

```
#include "Vector3D.h"
#include "Vector2D.h"
#include "boxes.h"
#include "PBuffer.h"
#include "GlslProgram.h"

Vector3D eye    ( -0.5, -0.5, 1.5 ); // camera position
unsigned decalMap;                  // decal (diffuse) texture
unsigned stoneMap;
unsigned teapotMap;
unsigned fringeMap;
unsigned reflectionMap;
float angle     = 0;
float rot       = 0;
Vector3D eta   ( 1.05, 1.1, 1.15 );
Vector3D r0;
bool useFilter = true;
bool useRotation = true;

GlslProgram program;

void init ()
{
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );
    glEnable   ( GL_DEPTH_TEST );
    glEnable   ( GL_TEXTURE_2D );
    glDepthFunc ( GL_LEQUAL      );

    glHint ( GL_POLYGON_SMOOTH_HINT,           GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}

void displayBoxes ()
{
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();

    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable        ( GL_TEXTURE_2D );
```

```
glDisable      ( GL_TEXTURE_CUBE_MAP );

glRotatef ( rot, 0, 0, 1 );
drawBox   ( Vector3D ( -5, -5, 0 ), Vector3D ( 10, 10, 3 ),
            stoneMap, false );
drawBox   ( Vector3D ( 3, 2, 0.5 ), Vector3D ( 1, 2, 2 ),
            decalMap );

glActiveTextureARB ( GL_TEXTURE1_ARB );
glBindTexture      ( GL_TEXTURE_1D, fringeMap );
glActiveTextureARB ( GL_TEXTURE2_ARB );
glBindTexture      ( GL_TEXTURE_CUBE_MAP, reflectionMap );
glActiveTextureARB ( GL_TEXTURE0_ARB );

 glEnable      ( GL_BLEND );
glBlendFunc ( GL_SRC_ALPHA, GL_ONE );

if ( useFilter )
    program.bind ();

glBindTexture ( GL_TEXTURE_2D, teapotMap );
glPushMatrix ();
glTranslatef ( 0.2, 1, 1.5 );
glRotatef     ( angle * 45.3, 1, 0, 0 );
glRotatef     ( angle * 57.2, 0, 1, 0 );
glutSolidTeapot ( 0.3 );

glPopMatrix  ();
glPushMatrix  ();
glTranslatef ( 2.7, 2, 0.5 );

glutSolidSphere ( 0.5, 20, 20 );

glPopMatrix  ();

if ( useFilter )
    program.unbind ();

glDisable      ( GL_BLEND );
```

```
glActiveTextureARB ( GL_TEXTURE1_ARB );
glBindTexture      ( GL_TEXTURE_3D, 0 );
glActiveTextureARB ( GL_TEXTURE2_ARB );
glBindTexture      ( GL_TEXTURE_1D, 0 );
glActiveTextureARB ( GL_TEXTURE3_ARB );
glBindTexture      ( GL_TEXTURE_CUBE_MAP, 0 );
glPopMatrix ();

}

void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    displayBoxes ();

    glutSwapBuffers ();
}

void reshape ( int w, int h )
{
    glViewport      ( 0, 0, (GLsizei)w, (GLsizei)h );
    glMatrixMode    ( GL_PROJECTION );
    glLoadIdentity ();
    gluPerspective ( 60.0, (GLfloat)w / (GLfloat)h, 1.0, 60.0 );
    glMatrixMode    ( GL_MODELVIEW );
    glLoadIdentity ();
    gluLookAt       ( eye.x, eye.y, eye.z,           // eye
                      3, 3, 1,                  // center
                      0, 0, 1 );                // up
}

void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
        exit ( 0 );

    if ( key == ' ' )
        useRotation = !useRotation;
}
```

```
void    specialKey ( int key, int x, int y )
{
    if ( key == GLUT_KEY_RIGHT )
        rot += 5;
    else
        if ( key == GLUT_KEY_LEFT )
            rot -= 5;

    glutPostRedisplay ();
}

void    animate ()
{
    static float lastTime = 0.0;
    float    time       = 0.001f * glutGet ( GLUT_ELAPSED_TIME );

    if ( useRotation )
        angle += 2 * (time - lastTime);

    lastTime = time;

    program.bind ();
    program.setUniformVector ( "eyePos", eye );
    program.setUniformVector ( "r0", r0 );
    program.setUniformFloat ( "time", time );
    program.unbind ();

    glutPostRedisplay ();
}

int main ( int argc, char * argv [] )
{
    // initialize glut
    glutInit          ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize ( 512, 512 );

    // create window
    glutCreateWindow ( "OpenGL soap bubbles" );
}
```

```
// register handlers
glutDisplayFunc  ( display      );
glutReshapeFunc  ( reshape      );
glutKeyboardFunc ( key         );
glutSpecialFunc  ( specialKey );
glutIdleFunc     ( animate      );

init              ();
initExtensions ();

if ( !GlslProgram :: isSupported () )
{
printf ( "GLSL not supported.\n" );

return 1;
}

if ( !program.loadShaders ( "bubble.vsh", "bubble.fsh" ) )
{
printf ( "Error loading shaders:\n%s\n",
        program.getLog ().c_str () );

return 3;
}

const char * faces [6] =
{
"../../Textures/Cubemaps/cm_left.tga",
"../../Textures/Cubemaps/cm_right.tga",
"../../Textures/Cubemaps/cm_top.tga",
"../../Textures/Cubemaps/cm_bottom.tga",
"../../Textures/Cubemaps/cm_back.tga",
"../../Textures/Cubemaps/cm_front.tga",
};

reflectionMap = createCubeMap ( true, faces );
fringeMap = createTexture1D ( true, "fringe.png" );
decalMap  = createTexture2D ( true, "../../Textures/oak.bmp" );
stoneMap  = createTexture2D ( true, "../../Textures/block.bmp" );
```

```

teapotMap = createTexture2D ( true, ".../Textures/Oxidated.jpg" );

program.bind ();
program.setTexture ( "fringeMap", 1 );
program.setTexture ( "reflectionMap", 2 );
program.unbind ();

r0.x = (1 - eta.x)/(1 + eta.x);
r0.y = (1 - eta.y)/(1 + eta.y);
r0.z = (1 - eta.z)/(1 + eta.z);
r0 *= r0;

printf ( "Space toggles rotation on/off\n" );

glutMainLoop ();

return 0;
}

```

Если при рендеринге "мыльных пузырей" рассматривалось наложение отраженных волн (т. е. угол падения был равен углу отражения), то для рендеринга поверхности компакт-диска это уже неверно. Для такой поверхности рассматривается ее непосредственное освещение (источником света), поэтому угол между нормалью и направлением на наблюдателя может отличаться от угла между нормалью и направлением на источник света. Кроме того, в случае поверхности компакт-диска происходит наложение световых волн, отразившихся от соседних дорожек (рис. 16.4).

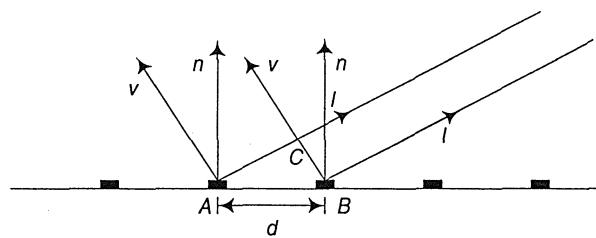


Рис. 16.4. Дифракция на поверхности компакт-диска

Если расстояние между соседними дорожками равно d , то разность путей, пройденных лучами, отразившимися от соседних дорожек, равна $AC - BC$.

Если обозначить через θ_1 и θ_2 угол между нормалью и направлением на наблюдателя и угол между нормалью и направлением на источник света, то эту разность можно приближенно записать как $d \cdot u$, где $u = \sin \theta_1 - \sin \theta_2$.

Тогда для света с длиной волны λ усиление будет происходить в случае $|u| \cdot d = n \lambda$, где n — произвольное натуральное число (т. е. когда $|u| \cdot d$ кратно λ). Исходя из того, что человеческий глаз различает только свет с длиной волны от 0,1 до 1 микрона, это позволит ограничить перебор возможных значения для n . Тем не менее, все равно получается довольно большая неопределенность, поэтому удобнее сразу ограничить перебор возможных значений для n несколькими первыми натуральными числами. Для каждого такого номера n из таблицы (т. е. одномерной текстуры) по величине $d \cdot |u| / n$ будет извлекаться цвет, соответствующий усиливающейся длине волны. Такие цвета будут суммироваться по набору перебираемых n .

В качестве основной модели освещения для поверхности компакт-диска удобнее всего использовать одну из стандартных моделей анизотропного освещения, например модель Уорда.

В листингах 16.13 и 16.14 приведены соответствующие вершинный и фрагментный шейдеры. Результат рендеринга показан на рис. 16.5.

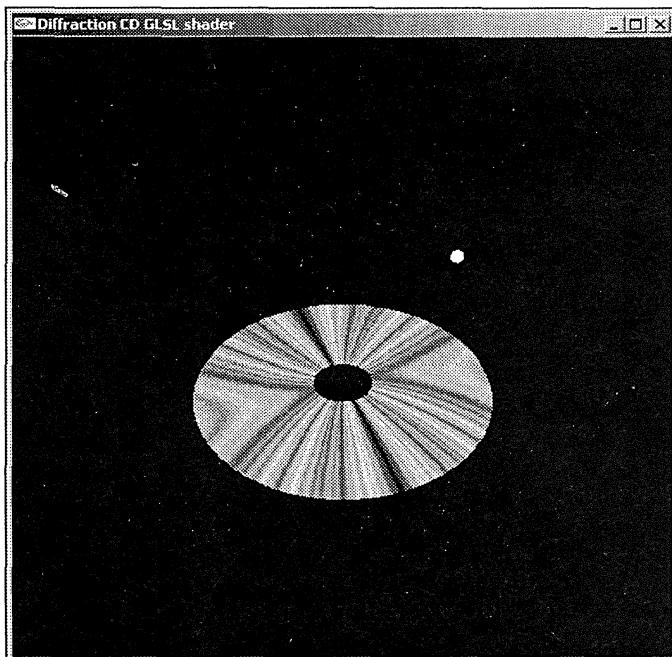


Рис. 16.5. Результат рендеринга "компакт-диска"

Примечание

Обратите внимание на то, что в фрагментном шейдере диск выводится как квадрат (отбрасываются все точки квадрата, не лежащие внутри кольца с радиусами 0,1 и 0,5).

Листинг 16.13. Вершинный шейдер для рендеринга объекта "Компакт-диск" с учетом дифракции

```

//  

// Simple diffraction and anisotropic Ward lighting vertex shader  

// Code based on chapter 8 from GPU Gems  

//  

varying    vec3 lt;  

varying    vec3 vt;  

varying    vec3 ht;  

uniform    vec4 lightPos;  

uniform    vec4 eyePos;  

void main(void)  

{  

    vec3    p = vec3      ( gl_ModelViewMatrix * gl_Vertex );  

    vec3    l = normalize ( vec3 ( lightPos ) - p );  

    vec3    v = normalize ( vec3 ( eyePos ) - p );  

    vec3    h = normalize ( l + v );  

    vec3    n = gl_NormalMatrix * gl_Normal;  

    vec3    t = gl_NormalMatrix * gl_MultiTexCoord1.xyz;  

    vec3    b = gl_NormalMatrix * gl_MultiTexCoord2.xyz;  

        // now remap v, l, and h into tangent space  

    vt = vec3 ( dot ( v, t ), dot ( v, b ), dot ( v, n ) );  

    lt = vec3 ( dot ( l, t ), dot ( l, b ), dot ( l, n ) );  

    ht = vec3 ( dot ( h, t ), dot ( h, b ), dot ( h, n ) );  

    gl_Position      = gl_ModelViewProjectionMatrix * gl_Vertex;  

    gl_TexCoord [0] = gl_MultiTexCoord0;  

}

```

Листинг 16.14. Фрагментный шейдер для рендеринга объекта "Компакт-диск" с учетом дифракции

```

//  

// Simple diffraction with anisotropic Ward lighting fragment shader  

// Code based on chapter 8 from GPU Gems  

//  


```

```
varying vec3 vt;
varying vec3 lt;
varying vec3 ht;
uniform sampler2D tangentMap;
uniform sampler2D decalMap;
uniform sampler1D rainbowMap;

void main (void)
{
    const vec4 specColor = vec4 ( 0, 0, 1, 0 );
    const vec3 n         = vec3 ( 0, 0, 1 );
    const float roughness = 30;
    const float d         = 40;

    float r = length ( gl_TexCoord [0].xy - vec2 ( 0.5 ) );

    if ( r > 0.5 || r < 0.1 )
        discard;

    vec3 tang      = normalize ( 2.0 * texture2D ( tangentMap,
                                                gl_TexCoord [0].xy ).xyz - vec3 ( 1 ) );
    float u         = dot ( ht, tang );
    float w         = dot ( ht, n );
    float e         = roughness * u / w;
    vec3 diffColor = vec3 ( 0.0 );

    u = d * abs ( u );

    for ( int n = 1; n <= 8; n++ )
        diffColor += texture1D ( rainbowMap, u / n ).rgb;

    gl_FragColor.rgb = specColor.rgb * exp ( -e*e ) + diffColor/4;
    gl_FragColor.a   = 1;
}
```

Примечание

Используемая в этом примере текстура RAINBOW.BMP строится при помощи скрипта на языке Python из каталога CODE\SCRIPTS на сопроводительном компакт-диске книги.

Обработка изображений средствами GLSL

Еще одним очень интересным применением GLSL (и GPU) является обработка изображений. При этом исходное (подлежащее обработке) изображение помещается в текстуру и осуществляется рендеринг прямоугольника с этой текстурой.

Сам рендеринг осуществляется тоже в текстуру (мы будем для этого использовать класс `FrameBuffer`).

Поддержка расширения `ARB_texture_non_power_of_two` [1] позволяет свободно использовать текстуры любого размера (в том числе и не являющиеся степенью числа 2). Возможно одновременное использование сразу нескольких входных текстур (рис. 16.6).

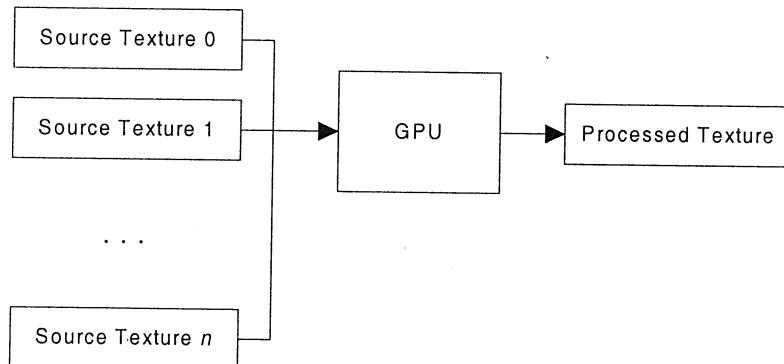


Рис. 16.6. Схема обработки изображений на GPU

Возможности современных GPU позволяют реализовать даже сложный эффект обработки изображений в один проход. Таким образом, применение GPU позволяет практически мгновенно обрабатывать изображения (даже если при этом не задействован центральный процессор). Это позволяет использовать обработку изображений на GPU для организации нелинейного видеомонтажа в реальном времени.

В последней версии операционной системы Mac OS X Tiger имеется встроенная библиотека обработки изображений, основанная на использовании GPU. Кроме большого количества стандартных эффектов, пользователь может легко добавлять новые эффекты, реализуя их на некотором подмножестве языка GLSL.

Рассмотрим теперь, как можно реализовать основные эффекты (фильтры) при помощи GLSL.

Фильтр Sepia

Одним из наиболее простых эффектов является так называемый фильтр Sepia ("сепия") — все пиксели изображения переводятся в оттенки одного цвета в зависимости от их яркости.

Для определения интенсивности пикселя по его RGB-компонентам обычно используется следующая формула:

$$I = 0,3r + 0,59g + 0,11b. \quad (16.6)$$

Примечание

Обратите внимание: расчет интенсивности можно реализовать как всего лишь одно скалярное произведение двух трехмерных векторов — $(0,3, 0,59, 0,11)$ и (r, g, b) .

Для получения оттенков заданного цвета достаточно умножить его RGB-представление на интенсивность.

Данный эффект реализуется двумя шейдерами, приведенными в листингах 16.15 и 16.16.

Листинг 16.15. Вершинный шейдер для фильтра Sepia

```
//  
// Simple sepia effect  
//  
  
void main(void)  
{  
    gl_Position      = gl_ModelViewProjectionMatrix * gl_Vertex;  
    gl_TexCoord [0] = gl_MultiTexCoord0;  
}
```

Листинг 16.16. Фрагментный шейдер для фильтра Sepia

```
//  
// Simple sepia fragment shader  
//  
uniform sampler2D mainTex;  
  
void main (void)  
{
```

```
const vec3 luminance = vec3 ( 0.3, 0.59, 0.11 );
const vec3 sepiaColor = vec3 ( 1, 0.89, 0.54 );

vec4 color = texture2D ( mainTex, gl_TexCoord [0].xy );
gl_FragColor = vec4 ( dot ( color.rgb, luminance ) * sepiaColor, 1 );
}
```

Соответствующий код на C++ приведен в листинге 16.17.

Обратите внимание на функции `startOrtho` и `endOrtho`, обеспечивающие использование параллельного проектирования и работы в координатах окна.

Листинг 16.17. Код на C++, реализующий фильтр Sepia

```
//  
// Sample to image postprocessing via framebuffer and GLSL programs  
  
//  
#include "libExt.h"  
#include <glut.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include "libTexture.h"  
#include "TypeDefs.h"  
#include "Vector3D.h"  
#include "Vector2D.h"  
#include "boxes.h"  
#include "FrameBuffer.h"  
#include "GlslProgram.h"  
  
Vector3D eye ( -0.5, -0.5, 1.5 ); // camera position  
unsigned decalMap; // decal (diffuse) texture  
unsigned stoneMap;  
unsigned teapotMap;  
unsigned screenMap;  
float angle = 0;  
float rot = 0;  
bool useFilter = true;  
  
GlslProgram program;  
FrameBuffer buffer ( 512, 512, FrameBuffer :: depth32 );
```

```
void    renderToBuffer () ;

void init ()
{
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );
    glEnable     ( GL_DEPTH_TEST );
    glEnable     ( GL_TEXTURE_2D );
    glDepthFunc  ( GL_LEQUAL      );

    glHint ( GL_POLYGON_SMOOTH_HINT,           GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}

void displayBoxes ()
{
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();

    glRotatef    ( rot, 0, 0, 1 );

    drawBox ( Vector3D ( -5, -5, 0 ), Vector3D ( 10, 10, 3 ), stoneMap,
              false );
    drawBox ( Vector3D ( 3, 2, 0.5 ), Vector3D ( 1, 2, 2 ), decalMap );

    glBindTexture ( GL_TEXTURE_2D, teapotMap );
    glTranslatef   ( 0.2, 1, 1.5 );
    glRotatef     ( angle * 45.3, 1, 0, 0 );
    glRotatef     ( angle * 57.2, 0, 1, 0 );
    glutSolidTeapot ( 0.3 );
    glPopMatrix   ();
}

void    startOrtho ()
{
    glMatrixMode   ( GL_PROJECTION ); // select the projection matrix
    glPushMatrix   ();             // store the projection matrix
    glLoadIdentity ();             // reset the projection matrix
        // set up an ortho screen
    glOrtho        ( 0, 512, 0, 512, -1, 1 );
}
```

```
glMatrixMode ( GL_MODELVIEW ); // select the modelview matrix
glPushMatrix (); // store the modelview matrix
glLoadIdentity (); // reset the modelview matrix
}

void endOrtho ()
{
    glMatrixMode ( GL_PROJECTION ); // select the projection matrix
    glPopMatrix (); // restore the old projection matrix
    glMatrixMode ( GL_MODELVIEW ); // select the modelview matrix
    glPopMatrix (); // restore the old projection matrix
}

void display ()
{
    renderToBuffer ();
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    startOrtho ();
    glBindTexture ( GL_TEXTURE_2D, screenMap );
    if ( useFilter )
        program.bind ();
    glBegin ( GL_QUADS );
    glTexCoord2f ( .0, 0 );
    glVertex2f ( 0, 0 );
    glTexCoord2f ( 1, 0 );
    glVertex2f ( 512, 0 );
    glTexCoord2f ( 1, 1 );
    glVertex2f ( 512, 512 );
    glTexCoord2f ( 0, 1 );
    glVertex2f ( 0, 512 );
    glEnd ();
    if ( useFilter )
        program.unbind ();
    endOrtho ();
    glutSwapBuffers ();
}

void reshape ( int w, int h )
{
```

```
glViewport      ( 0, 0, (GLsizei)w, (GLsizei)h );
glMatrixMode   ( GL_PROJECTION );
glLoadIdentity ();
gluPerspective ( 60.0, (GLfloat)w/(GLfloat)h, 1.0, 60.0 );
glMatrixMode   ( GL_MODELVIEW );
glLoadIdentity ();
gluLookAt      ( eye.x, eye.y, eye.z,      // eye
                  3, 3, 1,           // center
                  0, 0, 1 );        // up
}

void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )          // quit requested
        exit ( 0 );

    if ( key == 'f' || key == 'F' )
        useFilter = !useFilter;
}

void specialKey ( int key, int x, int y )
{
    if ( key == GLUT_KEY_RIGHT )
        rot += 5;
    else
        if ( key == GLUT_KEY_LEFT )
            rot -= 5;
    glutPostRedisplay ();
}

void animate ()
{
    angle = 0.004f * glutGet ( GLUT_ELAPSED_TIME );
    program.bind ();
    program.setUniformFloat ( "time", angle );
    program.unbind ();
    glutPostRedisplay ();
}
```

```
void    renderToBuffer ()
{
    glBindTexture ( GL_TEXTURE_2D, 0 );
    buffer.bind ();
    glClearColor ( 0, 0, 1, 1 );
    glClear      ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    reshape ( buffer.getWidth (), buffer.getHeight () );
    displayBoxes ();
    buffer.unbind ();
}

int main ( int argc, char * argv [] )
{
    // initialize glut
    glutInit          ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH );
    glutInitWindowSize ( 512, 512 );
    // create window
    glutCreateWindow ( "OpenGL image postprocessing - sepia effect" );
    // register handlers
    glutDisplayFunc   ( display      );
    glutReshapeFunc   ( reshape      );
    glutKeyboardFunc   ( key         );
    glutSpecialFunc   ( specialKey );
    glutIdleFunc      ( animate      );

    init ();
    initExtensions ();

    assertExtensionsSupported ( "GL_ARB_shading_language_100 GL_ARB_shader_objects
EXT_framebuffer_object" );
    if ( !program.loadShaders ( "sepia.vsh", "sepia.fsh" ) )
    {
        printf ( "Error loading shaders:\n%s\n",
                program.getLog ().c_str () );
        return 3;
    }

    decalMap = createTexture2D ( true, "../../../Textures/oak.bmp" );
}
```

```
stoneMap = createTexture2D ( true, "../../../Textures/block.bmp" );
teapotMap = createTexture2D ( true, "../../../Textures/Oxidated.jpg" );
screenMap = buffer.createColorTexture ();

buffer.create ();
buffer.bind ();
buffer.attachColorTexture ( GL_TEXTURE_2D, screenMap );

if ( !buffer.isOk () )
    printf ( "Error with framebuffer\n" );

buffer.unbind ();
program.bind ();
program.setTexture ( "mainTex", 0 );
program.unbind ();

printf ( "Press F key to turn filtering on/off\n" );

glutMainLoop ();

return 0;
}
```

В данном примере осуществляется рендеринг в текстуру, реализованную в виде объекта класса `FrameBuffer`. Для этого сначала с помощью метода `create` создается фреймбуфер, затем с помощью метода `createColorTexture` создается текстура и подключается в качестве цветового буфера.

Во время работы программы можно использовать клавишу `<F>` для включения/выключения эффекта.

Гамма-коррекция

Еще один интересный эффект — так называемая *гамма-коррекция* (γ -коррекция) — каждый компонент преобразуется по закону $f(t) = t^{1/\gamma}$. Соответствующий фрагментный шейдер приведен в листинге 16.18.

Примечание

Поскольку соответствующие вершинные шейдеры и программы на C++ для остальных эффектов практически совпадают с используемыми для фильтра Sepia, для них приводятся только фрагментные шейдеры.

Листинг 16.18. Фрагментный шейдер для γ -коррекции

```
//  
// Gamma correction effect fragment shader  
//  
uniform sampler2D mainTex;  
void main (void)  
{  
    const vec3 gamma = vec3 ( 0.6 );  
    vec3 color = texture2D ( mainTex, gl_TexCoord [0].xy );  
    gl_FragColor = vec4 ( pow ( color, 1.0 / gamma ), 1 );  
}
```

Фильтр выделения границ

Еще один интересный фильтр — фильтр выделения границ (edge detect). Данный фильтр выделяет повышенной яркостью те участки изображения, где резко меняется интенсивность пикселов.

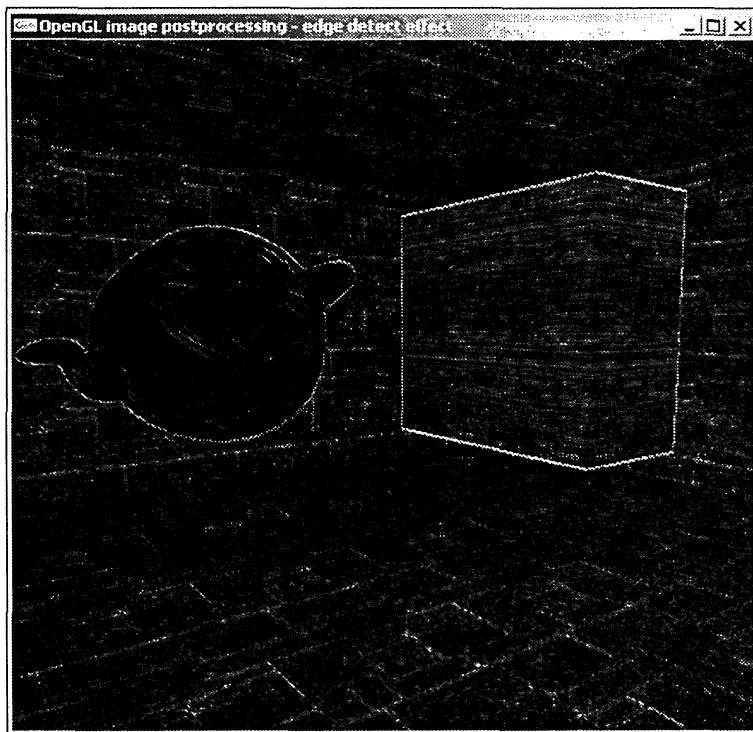


Рис. 16.7. Результат работы фильтра выделения границ

Если через $I_{i,j}$ обозначить интенсивность пикселя (i,j) , то работа фильтра выражается формулой (16.7):

$$C_{i,j} = k \cdot (|I_{i=1,j} - I_{i-1,j}| + |I_{i,j+1} - I_{i,j-1}|). \quad (16.7)$$

В этой формуле параметром k обозначен масштабирующий коэффициент, применяемый ко всему изображению.

На рис. 16.7 приведен результат работы данного фильтра.

Листинг 16.19. Фрагментный шейдер, реализующий фильтр выделения границ

```
//  
// Edge detect effect fragment shader  
//  
  
uniform sampler2D mainTex;  
  
void main (void)  
{  
    const vec3 luminance = vec3 ( 0.3, 0.59, 0.11 );  
    const vec3 sepiaColor = vec3 ( 1, 0.89, 0.54 );  
    const vec2 d01 = vec2 ( 0, 1.0 / 512.0 );  
    const vec2 d10 = vec2 ( 1.0 / 512.0, 0 );  
    const float scale = 1.0;  
  
    float c1 = dot ( luminance, texture2D ( mainTex,  
                                           gl_TexCoord [0].xy + d01 ).rgb );  
    float c2 = dot ( luminance, texture2D ( mainTex,  
                                           gl_TexCoord [0].xy - d01 ).rgb );  
    float c3 = dot ( luminance, texture2D ( mainTex,  
                                           gl_TexCoord [0].xy + d10 ).rgb );  
    float c4 = dot ( luminance, texture2D ( mainTex,  
                                           gl_TexCoord [0].xy - d10 ).rgb );  
  
    gl_FragColor = vec4 ( vec3 ( scale * ( abs ( c1 - c2 ) +  
                                abs ( c2 - c3 ) ) ), 1 );  
}
```

Если слегка модифицировать данный шейдер, то получится фильтр, строящий карту нормалей (bump map) по карте высот (height map).

Коррекция цвета в пространстве HSV

Еще одним интересным фильтром является коррекция цвета в цветовом пространстве HSV.

Пространство HSV использует еще один способ представления цвета, который (в отличие от RGB-представления) больше понятен человеку: цвет описывается тремя характеристиками — *интенсивностью* (V, Value), *тоном* (H, Hue) и *насыщенностью* (S, Saturation).

Изменение параметра V изменяет только яркость, не затрагивая остальные характеристики цвета. Изменение параметра S изменяет насыщенность цвета — можно сделать его более сочным, насыщенным или, наоборот, блеклым и серым.

И хотя преобразование цвета из RGB- в HSV-пространство и обратно не является линейным, его можно легко реализовать в виде шейдера на GLSL (рис. 16.8).

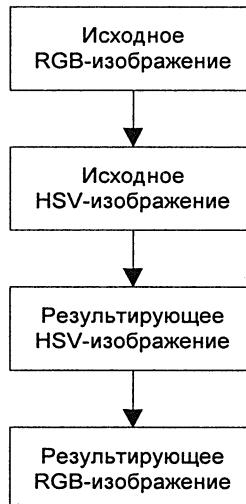


Рис. 16.8. Коррекция изображения при помощи преобразования в HSV-пространство

Для коррекции цвета цвет каждого пикселя переводится в HSV-представление, компоненты которого домножаются на величину коррекции (листинг 16.20).

Листинг 16.20. Фрагментный шейдер для коррекции цвета в HSV-пространстве

```

//  

// HSV correction image filter fragment shader  

//  


```

```
uniform sampler2D mainTex;
uniform float hScale;
uniform float sScale;
uniform float vScale;

vec3 rgbToHsv ( const in vec3 c )
{
    float mn = min ( min ( c.r, c.g ), c.b );
    float mx = max ( max ( c.r, c.g ), c.b );
    float delta = mx - mn;
    float h, s, v;

    v = mx;
    if ( mx > 0.001 )
    {
        s = delta / mx;
        if ( c.r == mx )
            h = ( c.g - c.b ) / delta;
        else
            if ( c.g == mx )
                h = 2 + ( c.b - c.r ) / delta;
            else
                h = 4 + ( c.r - c.g ) / delta;
    }
    else
    {
        s = 0;
        h = 0;
    }

    return vec3 ( h / 6.0, s, v );
}

vec3 hsvToRgb ( const in vec3 c )
{
    float r, b, g;
    float h = c.x;
    float s = c.y;
    float v = c.z;
```

```
vec3 res;

if ( s < 0.001 )
    res = vec3 ( v, v, v );
else
{
    h *= 6;
    int i = floor ( h );
    float f = h - i;
    float p = v * ( 1 - s );
    float q = v * ( 1 - s * f );
    float t = v * ( 1 - s * ( 1 - f ) );
    if ( i == 0 )
        res = vec3 ( v, t, p );
    else
        if ( i == 1 )
            res = vec3 ( q, v, p );
        else
            if ( i == 2 )
                res = vec3 ( p, v, t );
            else
                if ( i == 3 )
                    res = vec3 ( p, q, v );
                else
                    if ( i == 4 )
                        res = vec3 ( t, p, v );
                    else
                        res = vec3 ( v, p, q );
}
return res;
}

void main (void)
{
    vec3 color = texture2D ( mainTex, gl_TexCoord [0].xy );
    vec3 hsv = rgbToHsv ( color );
    hsv *= vec3 ( hScale, sScale, vScale );
    gl_FragColor = vec4 ( hsvToRgb ( hsv ), 1.0 );
}
```

На сопроводительном компакт-диске книги в каталоге Code\Chapter-16 имеется полный текст соответствующих вершинного шейдера и программы на C++.

Примечание

С помощью клавиш $<+>$ и $<->$, $<*>$ и $</>$, $<[>$ и $<]>$ можно изменять (масштабировать) компоненты H, S и V и наблюдать то, какие изменения в цвете вызовет такое изменение коэффициентов.

Размытие изображения

Еще один интересный (и часто используемый) эффект — размытие (blur) изображения. Работа этого фильтра выражается формулой (16.8):

$$C_{\text{out},i,j} = \sum_{l,m} \alpha_{l,m} \cdot C_{i+1,j+m}. \quad (16.8)$$

Параметром $C_{i,j}$ обозначены цвета пикселов исходного изображения, параметром $\alpha_{i,j}$ — так называемые коэффициенты свертки (ядро), а параметром $C_{\text{out},i,j}$ — цвета результирующего изображения.

Ясно, что при большом размере ядра свертки расчет размытия требует очень большого числа обращений к текстуре. Поэтому обычно используется следующий подход, позволяющий заметно сократить число текстурных обращений.

Для определенного класса ядер свертки (например $\alpha_{i,j} = 1$ или $\alpha_{i,j} = \exp(-i^2 - j^2)$), называемых *разделяемыми* (separable), можно осуществлять размытие в два этапа. Сначала проводится размытие только по горизонтали, а затем только по вертикали. В результате получается правильное изображение.

В листингах 16.21 и 16.22 приведены фрагментные шейдеры для размытия по горизонтали и по вертикали для ядра размером 7×7 .

Листинг 16.21. Фрагментный шейдер для x-размытия

```
//  
// Simple x-blur fragment shader  
//  
  
uniform sampler2D mainTex;  
  
void main (void)  
{
```

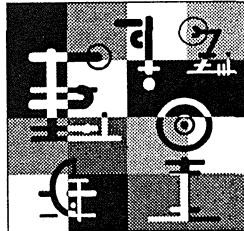
```
vec3 sum = vec3 ( 0 );
vec2 dx = vec2 ( 1.0 / 512.0, 0 );
vec2 tx = gl_TexCoord [0].xy - 3*dx;

for ( int i = 0; i < 7; i++ )
{
    sum += texture2D ( mainTex, tx ).rgb;
    tx += dx;
}

gl_FragColor = vec4 ( sum / 7.0, 1 );
}
```

Листинг 16.22. Фрагментный шейдер для у-размытия

```
//  
// Simple y-blur fragment shader  
//  
  
uniform sampler2D mainTex;  
  
void main (void)  
{  
    vec3 sum = vec3 ( 0 );
    vec2 dx = vec2 ( 0, 1.0 / 512.0 );
    vec2 tx = gl_TexCoord [0].xy - 3*dx;  
  
    for ( int i = 0; i < 7; i++ )
    {
        sum += texture2D ( mainTex, tx ).rgb;
        tx += dx;
    }
  
    gl_FragColor = vec4 ( sum / 7.0, 1 );
}
```



Глава 17

Использование шумовой функции в моделировании. Моделирование облаков, волн и материалов

Большинство реальных объектов (будь то облака на небе, структура прожилок камня, рябь на воде) несут в себе определенный элемент случайности и непредсказуемости. Хотя у дерева всегда четко прослеживается структура годовых колец, тем не менее каждый кусок дерева по-своему уникален, причем эта уникальность проявляется именно в отклонениях от идеальной структуры.

Ясно, что использование просто генератора случайных чисел (даже в виде текстуры) здесь совершенно неприемлемо — подобная случайность "слишком случайна" и непредсказуема. Повторный рендеринг того же объекта с незначительными изменениями даст совершенно другие значения, т. е. нет непрерывной зависимости результатов от входных данных — крайне малое изменение данных может привести к огромному скачку в значениях.

Именно как средство введения в шейдеры контролируемой случайности было введение Кеном Перлином (Ken Perlin) в 1985 году так называемой *шумовой функции* (noise, Perlin noise). Предложенная им функция (как и ее реализация в GLSL) удовлетворяет следующим требованиям:

- значения принимаются на отрезке $[-1, 1]$;
- средним значением является 0;
- одному и тому же значению аргумента всегда соответствует одно и то же значение функции, т. е. это не генератор случайных чисел;
- статистическая инвариантность относительно поворотов;
- статистическая инвариантность относительно переноса;
- C^1 -непрерывность.

На рис. 17.1 приведено изображение, генерируемое двухмерной шумовой функцией (значения из отрезка $[-1, 1]$ переводятся в оттенки серого цвета).

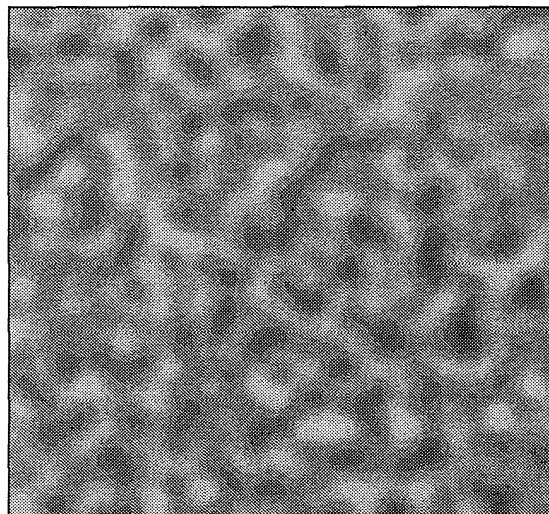


Рис. 17.1. Изображение, полученное с помощью двухмерной шумовой функции

Из рисунка видно, что использованная функция действительно удовлетворяет сформулированным требованиям.

Есть варианты шумовой функции от одной, двух, трех переменных. Несложно построить шумовую функцию от произвольного числа вещественных аргументов.

Точно так же можно построить шумовую функцию, принимающую векторное значение, — каждый компонент такой функции удовлетворяет сформулированным требованиям.

Поскольку далеко не во всех реализациях GLSL и далеко не для всех драйверов присутствует нормальная поддержка встроенных шумовых функций, вместо такой функции мы будем использовать трехмерную RGB-текстуру (файл TEXTURES\NOISE\NOISE-3D.DDS на сопроводительном диске книги), построенную на основе трехмерной шумовой функции.

Важным свойством этой текстуры является ее периодичность — значения на одной грани полностью совпадают со значениями на противоположной грани. Это необходимо для того, чтобы, введя закон приведения текстурных координат `GL_REPEAT`, мы могли бы рассматривать эту текстуру как непрерывную во всем пространстве функцию текстурных координат (s, t, r).

Без условия периодичности при переходе одного из компонентов текстурных координат через целое значение функция имела бы разрыв.

На сопроводительном компакт-диске книги в каталоге Code\utils находятся и реализация шумовой функции в виде класса языка C++ и реализация класса `noise3D`, описание которого приведено в листинге 17.1.

Листинг 17.1. Описание класса noise3D

```
//  
// Noise texture generator class  
  
#include    "Vector2D.h"  
#include    "Vector3D.h"  
  
class    NoiseTexture  
{  
private:  
    enum  
    {  
        n = 8           // size of matrix  
    };  
  
    Vector3D * g;  
  
    float    drop ( float t ) const  
    {  
        float    ta = (float) fabs ( t );  
  
        if ( ta <= 1 )  
            return 1 - ta*ta*ta*(10 - 15*ta + 6*ta*ta);  
  
        return 0;  
    }  
  
    const Vector3D&    n2D ( int x, int y, int z ) const  
    {  
        x &= n - 1;  
        y &= n - 1;  
        z &= n - 1;  
  
        return g [z*n*n + y*n + x];  
    }  
  
    Vector3D omega ( long i, long j, long k, const Vector3D& pt ) const  
    {
```

```
return n2D(i, j, k)*drop(pt.x)*drop(pt.y)*drop(pt.z);
}

public:
    NoiseTexture ( int seed = -1 );
    ~NoiseTexture ();

float    noise ( const Vector3D& pt ) const
{
    long      ip   = (long) floor ( pt.x );
    long      jp   = (long) floor ( pt.y );
    long      kp   = (long) floor ( pt.z );
    float     sum = 0;

    for ( register long i = ip; i <= ip + 1; i++ )
        for ( register long j = jp; j <= jp + 1; j++ )
            for ( register long k = kp; k <= kp + 1; k++ )
                sum += omega ( i, j, k, Vector3D ( pt.x - i,
                                                pt.y - j, pt.z - k ) ).x;

    return sum;
}

Vector3D    noise2D ( const Vector2D& pt ) const
{
    long      ip   = (long) floor ( pt.x );
    long      jp   = (long) floor ( pt.y );
    Vector3D  sum ( 0, 0, 0 );

    for ( register long i = ip; i <= ip + 1; i++ )
        for ( register long j = jp; j <= jp + 1; j++ )
            sum += omega ( i, j, 0, Vector3D ( pt.x - i, pt.y - j, 0 ) );

    return sum;
}

Vector3D    noise3D ( const Vector3D& pt ) const
{
    long      ip   = (long) floor ( pt.x );
```

```

long      jp  = (long) floor ( pt.y );
long      kp  = (long) floor ( pt.z );
Vector3D  sum ( 0, 0, 0 );

for ( register long i = ip; i <= ip + 1; i++ )
    for ( register long j = jp; j <= jp + 1; j++ )
        for ( register long k = kp; k <= kp + 1; k++ )
            sum += omega(i, j, k, Vector3D(pt.x-i, pt.y-j, pt.z-k));

return sum;
}

unsigned createNoiseTexture2D ( int width, int height ) const;
unsigned createNoiseTexture3D ( int width, int height,
                               int depth ) const;
bool     saveNoiseTexture3D ( int width, int height, int depth,
                           const char * fileName ) const;

unsigned createAbsNoiseTexture3D ( int width, int height,
                                   int depth ) const;
bool     saveAbsNoiseTexture3D ( int width, int height,
                               int depth,
                           const char * fileName ) const;
};


```

Данный класс предназначен для создания периодических трехмерных текстур на основе шумовой функции. Также он позволяет сохранять подобные текстуры в виде DDS-файлов.

Обратите внимание: в отличие от настоящей шумовой функции, функция, получаемая при помощи текстуры, является периодической. Однако путем очень простого приема можно на ее основе получить функцию со сколь угодно большим периодом.

В самом деле, пусть функция $f(t)$ имеет период T . Тогда функция $g(t) = f(1,001t)$ имеет период, равный $T / 1,001$.

Практически любая нетривиальная комбинация этих функций будет иметь период, равный $1000T$, поскольку:

$$f(1,001(t + 1000T)) = f(1,001t + 1001T) = f(1,001t).$$

Тем самым, комбинируя функции с очень близкими периодами, можно получать функции, период которых будет сколь угодно велик (ясно, что периода, равного $1001T$, вполне достаточно в большинстве случаев).

Использование шумовой функции для рендеринга "мыльных пузырей"

Рассмотрим применение шумовой функции в приведенном ранее шейдере "мыльных пузырей". Основным недостатком этого шейдера является постоянная установка толщины слоя, из-за чего в "пузыре" отсутствуют те самые случайные и непредсказуемые переливы цветов. Чтобы их добавить, достаточно в фрагментном шейдере изменять полученную из вершинного шейдера величину $d / (n, v)$ при помощи следующей функции:

$$\text{depthScale} = 1 + k \cdot \text{noise}(p + p_0 \cdot t). \quad (17.1)$$

Параметр p в формуле (17.1) обозначает пространственные координаты соответствующей точки в системе координат объекта (чтобы они не изменились при преобразованиях объекта при помощи модельной матрицы). Параметр k — это некоторый коэффициент, принимающий значения из интервала $(0, 1)$ и определяющий степень случайности в изменении цветов.

Параметром p_0 обозначен некоторый вектор, позволяющий учитывать время t при обращении к шумовой функции, — чем он больше, тем быстрее происходит изменение цветов в пузыре.

Величина p , необходимая для доступа к шумовой функции, должна вычисляться в вершинном шейдере и передаваться через `varying`-переменную.

В листингах 17.2 и 17.3 приведены измененные вершинный и фрагментный шейдеры для рендеринга "мыльных пузырей".

Листинг 17.2. Вершинный шейдер для рендеринга анимированных "мыльных пузырей"

```
//  
// Soap bubble vertex shader  
  
  
uniform vec4 eyePos;  
uniform vec3 r0; // components for R, G and B  
  
varying float viewDepth;  
varying vec3 pos;  
varying vec3 v;  
varying vec3 r;  
varying vec3 fresnel;  
  
void main()
```

```
{  
    const float filmDepth = 0.1;  
    const vec3 one = vec3 ( 1.0 );  
  
    gl_Position = ftransform ();  
  
    // transformed point to world space  
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );  
  
    // transform normal from model-space to view-space  
    vec3 n = normalize ( gl_NormalMatrix * gl_Normal );  
  
    v      = normalize ( eyePos.xyz - p );  
    r      = reflect   ( v, n );  
    viewDepth = filmDepth / dot ( n, v );  
    pos    = gl_Vertex.xyz;  
    fresnel = r0 + (one - r0) * pow (1.0-abs ( dot ( n,v ) ), 5.0);  
  
    // output texture coordinates for diffuse map  
    gl_TexCoord [0] = gl_MultiTexCoord0;  
}
```

Листинг 17.3. Фрагментный шейдер для рендеринга анимированных "мыльных пузырей"

```
//  
// Soap bubble fragment shader with animated noise  
//  
  
uniform sampler1D    fringeMap;  
uniform samplerCube   reflectionMap;  
uniform sampler3D     noiseMap;  
  
uniform float    time;  
varying float    viewDepth;  
varying vec3     pos;  
varying vec3     v;  
varying vec3     r;
```

```

varying vec3 fresnel;

void main()
{
    // get noise to adjust viewDepth
    vec3 ncoord      = vec3 ( pos.x, pos.y + time, pos.z ) * 0.3;
    vec3 noise       = texture3D ( noiseMap, ncoord ).rgb;
    float depthScale = (1.0 + noise.r)*0.4;

    // lookup fringe value based on view depth
    vec3 fringeColor = texture1D ( fringeMap, viewDepth.x *
                                    depthScale ).rgb;

    // lookup reflection in cube map
    vec3 reflColor = textureCube ( reflectionMap, r ).rgb;

    // modulate reflection by fringe color
    gl_FragColor = vec4 ( lerp ( fringeColor, reflColor, fresnel ), 0.4 );
}

```

В листинге 17.4 приведен код на C++, использующий шейдеры из листингов 17.2 и 17.3.

Листинг 17.4. Программа для рендеринга анимированных "мыльных пузырей"

```

// 
// Example of animated soap bubbles via GLSL programs
//
#include "libExt.h"

#include <glut.h>
#include <stdio.h>
#include <stdlib.h>
#include "libTexture.h"
#include "TypeDefs.h"
#include "Vector3D.h"
#include "Vector2D.h"
#include "boxes.h"
#include "noise3D.h"

```

```
#include "GlslProgram.h"

Vector3D eye  (-0.5, -0.5, 1.5); // camera position
Unsigned decalMap;           // decal (diffuse) texture
unsigned stoneMap;
unsigned teapotMap;
unsigned fringeMap;
unsigned reflectionMap;
unsigned noiseMap;
float angle      = 0;
float rot        = 0;
Vector3D eta ( 1.05, 1.1, 1.15 );
Vector3D r0;
Bool useFilter   = true;
bool useRotation = true;

GlslProgram program;
NoiseTexture noise;

void init ()
{
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );
    glEnable     ( GL_DEPTH_TEST );
    glEnable     ( GL_TEXTURE_2D );
    glDepthFunc ( GL_LEQUAL );

    glHint ( GL_POLYGON_SMOOTH_HINT, GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}

void displayBoxes ()
{
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();

    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable         ( GL_TEXTURE_2D );
    glDisable        ( GL_TEXTURE_CUBE_MAP );
```

```
glRotatef      ( rot, 0, 0, 1 );

drawBox  ( Vector3D ( -5, -5, 0 ), Vector3D ( 10, 10, 3 ),
           stoneMap, false );
drawBox  ( Vector3D ( 3, 2, 0.5 ), Vector3D ( 1, 2, 2 ),
           decalMap );

glActiveTextureARB ( GL_TEXTURE1_ARB );
glBindTexture      ( GL_TEXTURE_1D, fringeMap );
glActiveTextureARB ( GL_TEXTURE2_ARB );
glBindTexture      ( GL_TEXTURE_CUBE_MAP, reflectionMap );
glActiveTextureARB ( GL_TEXTURE0_ARB );

 glEnable      ( GL_BLEND );
glBlendFunc ( GL_SRC_ALPHA, GL_ONE );

if ( useFilter )
    program.bind ();

glBindTexture      ( GL_TEXTURE_2D, teapotMap );
glPushMatrix      ();
glTranslatef      ( 0.2, 1, 1.5 );
glRotatef        ( angle * 45.3, 1, 0, 0 );
glRotatef        ( angle * 57.2, 0, 1, 0 );

glutSolidTeapot ( 0.3 );
glPopMatrix      ();

glPushMatrix      ();
glTranslatef      ( 2.7, 2, 0.5 );
glutSolidSphere ( 0.5, 20, 20 );
glPopMatrix      ();

if ( useFilter )
    program.unbind ();

glDisable      ( GL_BLEND );
glActiveTextureARB ( GL_TEXTURE1_ARB );
glBindTexture      ( GL_TEXTURE_3D, 0 );
```

```
glActiveTextureARB ( GL_TEXTURE2_ARB );
glBindTexture      ( GL_TEXTURE_1D, 0 );
glActiveTextureARB ( GL_TEXTURE3_ARB );
glBindTexture      ( GL_TEXTURE_CUBE_MAP, 0 );
glPopMatrix ();
}

void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    displayBoxes ();
    glutSwapBuffers ();
}

void reshape ( int w, int h )
{
    glViewport      ( 0, 0, (GLsizei)w, (GLsizei)h );
    glMatrixMode   ( GL_PROJECTION );
    glLoadIdentity ();
    gluPerspective ( 60.0, (GLfloat)w/(GLfloat)h, 1.0, 60.0 );
    glMatrixMode   ( GL_MODELVIEW );
    glLoadIdentity ();
    gluLookAt       ( eye.x, eye.y, eye.z,           // eye
                      3, 3, 1,                  // center
                      0, 0, 1 );                // up
}

void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' ) // quit requested
        exit ( 0 );

    if ( key == ' ' )
        useRotation = !useRotation;
}

void     specialKey ( int key, int x, int y )
{
    if ( key == GLUT_KEY_RIGHT )
```

```
    rot += 5;
else
if ( key == GLUT_KEY_LEFT )
    rot -= 5;

glutPostRedisplay ();
}

void      animate ()
{
    static float lastTime = 0.0;
    float      time      = 0.001f * glutGet ( GLUT_ELAPSED_TIME );

if ( useRotation )
    angle += 2 * (time - lastTime);

lastTime = time;

program.bind ();
program.setUniformVector ( "eyePos", eye );
program.setUniformVector ( "r0", r0 );
program.setUniformFloat ( "time", time );
program.unbind ();

glutPostRedisplay ();
}

int main ( int argc, char * argv [] )
{
                                // initialize glut
glutInit             ( &argc, argv );
glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
glutInitWindowSize ( 512, 512 );

                                // create window
glutCreateWindow ( "OpenGL soap bubbles" );

                                // register handlers
glutDisplayFunc ( display );
```

```
glutReshapeFunc  ( reshape      );
glutKeyboardFunc ( key         );
glutSpecialFunc  ( specialKey  );
glutIdleFunc     ( animate     );

init            ();
initExtensions ();

if ( !GlslProgram :: isSupported () )
{
    printf ( "GLSL not supported.\n" );

    return 1;
}

if ( !program.loadShaders ( "bubble.vsh", "bubble.fsh" ) )
{
    printf ( "Error loading shaders:\n%s\n",
             program.getLog () .c_str () );

    return 3;
}

const char * faces [6] =
{
    ".../Textures/Cubemaps/cm_left.tga",
    ".../Textures/Cubemaps/cm_right.tga",
    ".../Textures/Cubemaps/cm_top.tga",
    ".../Textures/Cubemaps/cm_bottom.tga",
    ".../Textures/Cubemaps/cm_back.tga",
    ".../Textures/Cubemaps/cm_front.tga",
};

reflectionMap = createCubeMap ( true, faces );

fringeMap = createTexture1D ( true, "fringe.png" );
decalMap  = createTexture2D ( true, ".../Textures/oak.bmp" );
stoneMap  = createTexture2D ( true, ".../Textures/block.bmp" );
teapotMap = createTexture2D ( true, ".../Textures/Oxidated.jpg" );
```

```

noiseMap = noise.createNoiseTexture3D ( 64, 64, 64 );

program.bind ();
program.setTexture ( "fringeMap", 1 );
program.setTexture ( "reflectionMap", 2 );
program.unbind ();

r0.x = (1 - eta.x)/(1 + eta.x);
r0.y = (1 - eta.y)/(1 + eta.y);
r0.z = (1 - eta.z)/(1 + eta.z);
r0 *= r0;

printf ( "Space toggles rotation on/off\n" );

glutMainLoop ();

return 0;
}

```

Как и ранее, при помощи клавиши <Пробел> можно остановить вращение "мыльных" объектов, но сейчас даже при неподвижных объектах будут видны переливы цвета, изменяющиеся случайным образом с течением времени.

Искажение нормали при помощи шумовой функции

Возьмем рассмотренный уже шейдер, реализующий модель освещения Блинна, и при помощи шумовой функции искажим вектор нормали для каждого фрагмента (листинги 17.5 и 17.6).

Как видно на рис. 17.2, получилась иллюзия "хорошо помятого чайника".

Листинг 17.5. Вершинный шейдер для эффекта "мятого чайника"

```

//
// Simple noise-based bump mapping
//
varying vec3 l;
varying vec3 h;
varying vec3 n;

```

```
varying vec3 v;
varying vec3 pp;

uniform vec4 lightPos;
uniform vec4 eyePos;

void main(void)
{
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );

    l = normalize ( vec3 ( lightPos ) - p );
    v = normalize ( vec3 ( eyePos ) - p );
    h = normalize ( l + v );
    n = normalize ( gl_NormalMatrix * gl_Normal );
    pp = gl_Vertex;

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Листинг 17.6. Фрагментный шейдер для эффекта "мятого чайника"

```
//  
// Simple noise-based bump mapping  
  
//  
varying vec3 l;  
varying vec3 h;  
varying vec3 v;  
varying vec3 n;  
varying vec3 pp;  
  
uniform sampler3D noiseMap;  
  
void main (void)  
{  
    const vec4 diffColor = vec4 ( 0.5, 0.0, 0.0, 1.0 );  
    const vec4 specColor = vec4 ( 0.7, 0.7, 0.0, 1.0 );  
    const float specPower = 30;  
  
    vec3 n2 = normalize(n + 2*texture3D(noiseMap, 2*pp).rgb - vec3(1));
```

```
vec3  l2    = normalize ( 1 );
vec3  h2    = normalize ( h );
vec4  diff = diffColor * max ( dot ( n2, l2 ), 0.0 );
vec4  spec = specColor * pow ( max(dot(n2, h2), 0.0), specPower );

gl_FragColor = diff + spec;
}
```



Рис. 17.2. "Чайник" с искаженной при помощи шумовой функции нормалью

Обратите внимание: для искажения нормали используется не `texture3D(noiseMap, 2*pp)`, а `2*texture3D(noiseMap, 2*pp) - vec3(1.0)`. Это связано с тем, что все компоненты текстуры принимают значения на отрезке $[0, 1]$, в то время как шумовая функция принимает значения на отрезке $[-1, 1]$. Это преобразование и служит для перевода одного отрезка в другой.

Турбулентность

Чаще всего шумовая функция не применяется непосредственно, а служит основой для создания новых функций. Случайные искажения во многих природных материалах и процессах носят не просто случайный характер, но и очень часто фрактальный (т. е. самоподобный) характер. Так, увеличивая изображение жилки внутри мрамора, можно открывать все новые и новые детали.

К сожалению, шумовая функция подобным свойством не обладает. На рис. 17.1 сразу заметен характерный масштаб деталей, и увеличение новых деталей эта функция обнаружить уже не может.

Однако если рассмотреть функцию *noise(2p)*, то мы заметим, что характерный масштаб деталей уменьшился вдвое (рис. 17.3).

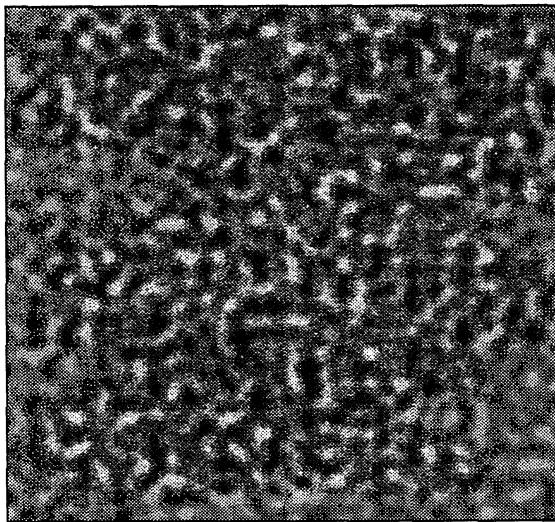


Рис. 17.3. Изображение функции *noise(2p)*

Если рассмотреть график функции *noise(4p)*, то мы увидим, что характерный размер деталей опять уменьшился вдвое (рис. 17.4).

Теперь если сложить все эти функции с весами 0,5, 0,25 и 0,125, то получится другое изображение (рис. 17.5).

Легко увидеть, что в данном изображении присутствуют детали сразу нескольких характерных размеров (точнее, всего трех). Таким образом, данное изображение можно в определенной степени считать фрактальным. Ясно, что увеличивая число слагаемых, мы будем добавлять все новые характерные размеры деталей, "увеличивая" тем самым фрактальность.

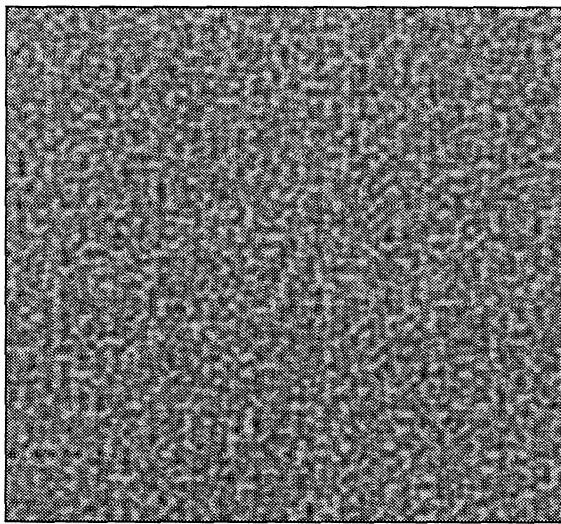


Рис. 17.4. Изображение функции $\text{noise}(4p)$

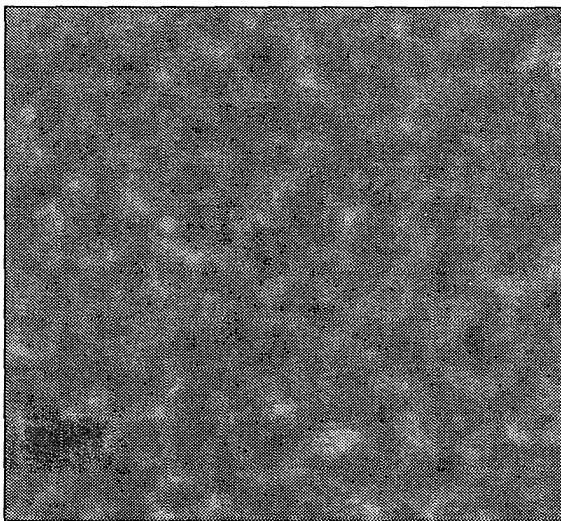


Рис. 17.5. Изображение для суммы шумовых функций

Таким образом, мы приходим к функции fBm :

$$fBm(p, f) = \sum_{k=0}^n \text{noise}\left(f^k \cdot p\right) \frac{1}{f^k}. \quad (17.2)$$

Название fBm этой функции является сокращением от fractal Brownian motion (фрактальное броуновское движение), т. к. оказалось, что броуновское движение частиц можно описывать при помощи этой функции.

Кроме нее часто рассматривают следующую функцию, называемую *турбулентностью*:

$$\text{turbulence}(p, f) = \sum_{k=0}^n \left| \text{noise}\left(f^k \cdot p\right) \right| \frac{1}{f^k}. \quad (17.3)$$

На рис. 17.6 приведено изображение "чайника", для искажения нормали которого была использована функция турбулентности.



Рис. 17.6. Чайник, для искажения нормали которого была использована функция турбулентности

Полезной особенностью функции *turbulence*, отличающей ее от *fBm*, является то, что благодаря использованию модуля, ее производные имеют точки разрыва.

Кроме функций *fBm* и *turbulence* иногда используется следующая функция:

$$\text{multifractal}(p, f) = \prod_{k=0}^n \left(\text{offs} + \text{noise}\left(f^k \cdot p\right) \cdot f \right).$$

Для дальнейшего использования поместим данные функции в специальный файл — ProcFuncs.h (листинг 17.7), который будет включаться в ряд исполь-

зуемых в дальнейшем шейдеров. Кроме того, поместим в этот файл ряд полезных периодических функций, часто используемых при построении процедурных текстур.

Листинг 17.7. Файл ProcFuncs.h

```
//  
// Small set of functions for procedural texturing  
//  
#ifndef __PROC_FUNCS__  
#define __PROC_FUNCS__  
//  
// Soft-tooth func, C1 in (n,n+1) with derivatives breaks  
// in integer points  
//  
float smoothToothWave ( in float t )  
{  
    float x = fract ( t );  
  
    return 4.0 * x * ( 1.0 - x );  
}  
//  
// Saw-tooth wave.  
// Periodic func with values in [0,1] and period of 1  
// Derivatives breaks at 0, 0.5, 1, 1.5, 2, ....  
//  
float sawWave ( in float t )  
{  
    t -= floor ( t );  
  
    return t >= 0.5f ? 2 * ( 1 - t ) : 2 * t;  
}  
//  
// Simple C1 periodic func with values in [0,1] and period of 1  
//  
float sineWave ( in float t )  
{  
    return 0.5 * ( 1.0 + sin ( t * 3.1415926 ) );  
}  
//  
// Stripes function
```

```
//  
vec3 stripes ( in vec3 x, in float f )  
{  
    vec3 t = 0.5 * ( 1.0 + sin ( f * x * 2 * 3.1415926 ) );  
    return t * t - 0.5;  
}  
//  
// takes values in [-1,1] range  
//  
vec3 turbulence ( const in vec3 p, const in float freqScale )  
{  
    float sum = 0;  
    vec3 t = vec3 ( 0 );  
    float f = 1;  
  
    for ( int i = 0; i <= 3; i++ )  
    {  
        t += abs ( 2 * texture3D ( noiseMap, p * f ).rgb -  
                   vec3 ( 1 ) ) / f;  
        sum += 1 / f;  
        f *= freqScale;  
    }  
    // remap from [0,sum] to [-1,1]  
    return 2 * t / sum - vec3 ( 1 );  
}  
//  
// Fractal Brownian motion func  
//  
vec3 fBm ( const in vec3 p, const in float freqScale )  
{  
    float sum = 0;  
    vec3 t = vec3 ( 0 );  
    float f = 1;  
  
    for ( int i = 0; i <= 3; i++ )  
    {  
        t += ( 2 * texture3D ( noiseMap, p * f ).rgb - vec3 ( 1.0 ) ) / f;  
        sum += 1 / f;  
        f *= freqScale;  
    }  
    // remap from [-sum,sum] to [-1,1]
```

```

    return t / sum;
}
//
// Multifractal function
//
vec3 multifractal ( const in vec3 p, const in float offset,
                     const in float freqScale )
{
    float sum = 0;
    vec3 t = vec3 ( 1 );
    float f = 1;

    for ( int i = 0; i <= 3; i++ )
    {
        t *= offset + ( 2 * texture3D ( noiseMap, p * f ).rgb -
                         vec3 ( 1 ) ) * freqScale;
        f *= freqScale;
    }
    return t;
}
#endif

```

На рис. 17.7 приведены графики некоторых периодических функций, введенных в листинге 17.7.

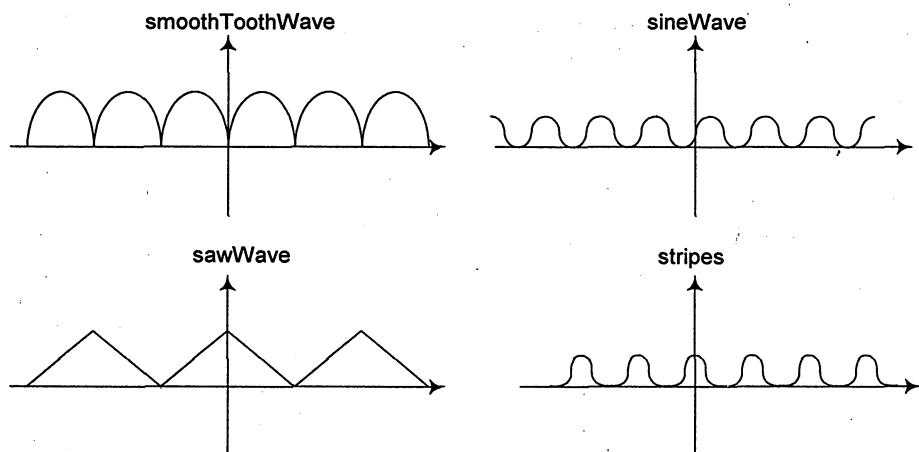


Рис. 17.7. Графики вспомогательных периодических функций из файла ProcFuncs.h

В листинге 17.8 приведен фрагментный шейдер, использованный для построения изображения с рис. 17.6.

Листинг 17.8. Фрагментный шейдер, использующий функцию turbulence для искажения нормали объекта

```
//  
// Fragment shader using turbulence to bent normal  
  
varying vec3 l;  
varying vec3 h;  
varying vec3 v;  
varying vec3 n;  
varying vec3 pp;  
  
uniform sampler3D noiseMap;  
  
#include "ProcFuncs.h"  
  
void main (void)  
{  
    const vec4 diffColor = vec4 ( 0.5, 0.0, 0.0, 1.0 );  
    const vec4 specColor = vec4 ( 0.7, 0.7, 0.0, 1.0 );  
    const float specPower = 30;  
  
    vec3 n2 = normalize ( n + 0.5*turbulence ( pp, 2.01379 ) );  
    vec3 l2 = normalize ( l );  
    vec3 h2 = normalize ( h );  
    vec4 diff = diffColor * ( 0.2 + max ( dot ( n2, l2 ), 0.0 ) );  
    vec4 spec = specColor * pow ( max ( dot ( n2, h2 ), 0.0 ),  
                                specPower );  
  
    gl_FragColor = diff + spec;  
}
```

Обратите внимание на то, что в качестве параметра f для функции `turbulence` использовалось число, слегка отличающееся от двух, — использование такого числа позволяет избежать возможных накладок, связанных с периодичностью шумовой функции.

Ржавление

В качестве еще одного примера использования шумовой функции рассмотрим шейдер, моделирующий разъедание объекта ржавчиной.

Простейший подход заключается в вычислении для каждого фрагмента функции `turbulence`, и если это значение оказывается меньше некоторого порогового значения, то данный фрагмент считается "проржавевшим насквозь" и отбрасывается (команда `discard`).

В листингах 17.9 и 17.10 приведены соответствующие вершинный и фрагментный шейдеры, демонстрирующие процесс ржавления объекта.

Листинг 17.9. Вершинный шейдер для моделирования процесса ржавления

```
//  
// Vertex shader for rust effect  
  
varying vec3 l;  
varying vec3 h;  
varying vec3 n;  
varying vec3 v;  
varying vec3 pp;  
  
uniform vec4 lightPos;  
uniform vec4 eyePos;  
  
void main(void)  
{  
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );  
  
    l = normalize ( vec3 ( lightPos ) - p );  
    v = normalize ( vec3 ( eyePos ) - p );  
    h = normalize ( l + v );  
    n = normalize ( gl_NormalMatrix * gl_Normal );  
    pp = gl_Vertex.xyz;  
  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

Листинг 17.10. Фрагментный шейдер для моделирования процесса ржавления

```
//  
// Fragment shader for rust effect  
  
varying vec3 l;  
varying vec3 h;  
varying vec3 v;  
varying vec3 n;  
varying vec3 pp;  
  
uniform sampler3D noiseMap;  
uniform float offs;  
  
#include "ProcFuncs.h"  
  
void main (void)  
{  
    const vec4 diffColor = vec4 ( 0.5, 0.0, 0.0, 1.0 );  
    const vec4 specColor = vec4 ( 0.7, 0.7, 0.0, 1.0 );  
    const float specPower = 30;  
  
    vec3 ns      = turbulence ( pp, 2.07193 );  
    float factor = (ns.x + ns.y + ns.z) / 3;  
  
    if ( factor < offs )  
        discard;  
  
    vec3 n2      = normalize ( n );  
    vec3 l2      = normalize ( l );  
    vec3 h2      = normalize ( h );  
    vec4 diff = max ( dot ( n2, l2 ), 0.0 ) + 0.3;  
    vec4 spec = specColor * pow ( max ( dot ( n2, h2 ), 0.0 ),  
                                specPower );  
  
    gl_FragColor = diff * diffColor + spec;  
}
```

В листинге 17.11 приведен соответствующий код на C++.

Примечание

Обратите внимание на использование uniform-переменной `offs`, позволяющей управлять степенью "ржавости" объекта. При помощи клавиш `<+>` и `<->` можно изменять значение этого параметра и, соответственно, управлять степенью "ржавости" объекта.

Листинг 17.11. Программа на C++, демонстрирующая "ржавление" объекта

```
//  
// Example of using rust shader  
  
#include "libExt.h"  
#include <glut.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include "libTexture.h"  
#include "libTexture3D.h"  
#include "Vector3D.h"  
#include "Vector2D.h"  
#include "Vector4D.h"  
#include "Data.h"  
#include "GlslProgram.h"  
#include "noise3D.h"  
  
Vector3D eye ( 7, 5, 7 ); // camera position  
Vector3D light ( 5, 0, 4 ); // light position  
Vector3D rot ( 0, 0, 0 );  
float angle = 0;  
float offs = -0.7;  
int mouseOldX = 0;  
int mouseOldY = 0;  
unsigned noiseMap;  
  
GlslProgram program;  
NoiseTexture noise;  
  
void init ()
```

```
{  
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );  
    glEnable      ( GL_DEPTH_TEST );  
    glDepthFunc   ( GL_LESS );  
  
    glHint ( GL_Polygon_SMOOTH_HINT,           GL_NICEST );  
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );  
}  
  
void display ()  
{  
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );  
  
    // draw the light  
    program.unbind ();  
  
    glMatrixMode     ( GL_MODELVIEW );  
    glPushMatrix ();  
    glTranslatef     ( light.x, light.y, light.z );  
    glColor4f        ( 1, 1, 1, 1 );  
    glutSolidSphere ( 0.1f, 15, 15 );  
    glPopMatrix ();  
    glMatrixMode     ( GL_MODELVIEW );  
    glPushMatrix ();  
  
    glRotatef       ( rot.x, 1, 0, 0 );  
    glRotatef       ( rot.y, 0, 1, 0 );  
    glRotatef       ( rot.z, 0, 0, 1 );  
  
    glActiveTextureARB ( GL_TEXTURE0_ARB );  
    glBindTexture     ( GL_TEXTURE_3D, noiseMap );  
    glEnable         ( GL_TEXTURE_3D );  
  
    program.bind ();  
    glutSolidTeapot ( 3 );  
    program.unbind ();  
  
    glPopMatrix ();  
    glutSwapBuffers ();
```

```
}

void reshape ( int w, int h )
{
    glViewport      ( 0, 0, (GLsizei)w, (GLsizei)h );
    glMatrixMode   ( GL_PROJECTION );
    glLoadIdentity ();
    gluPerspective ( 60.0, (GLfloat)w/(GLfloat)h, 1.0, 60.0 );
    gluLookAt       ( eye.x, eye.y, eye.z, // eye
                      0, 0, 0,           // center
                      0.0, 0.0, 1.0 ); // up

    glMatrixMode   ( GL_MODELVIEW );
    glLoadIdentity ();
}

void motion ( int x, int y )
{
    rot.y -= ((mouseOldY - y) * 180.0f) / 200.0f;
    rot.z -= ((mouseOldX - x) * 180.0f) / 200.0f;
    rot.x = 0;

    if ( rot.z > 360 )
        rot.z -= 360;

    if ( rot.z < -360 )
        rot.z += 360;

    if ( rot.y > 360 )
        rot.y -= 360;

    if ( rot.y < -360 )
        rot.y += 360;

    mouseOldX = x;
    mouseOldY = y;

    glutPostRedisplay ();
}
```

```
void mouse ( int button, int state, int x, int y )
{
    if ( state == GLUT_DOWN )
    {
        mouseOldX = x;
        mouseOldY = y;
    }
}

void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' ) // quit requested
        exit ( 0 );

    if ( key == '+' )
        offs += 0.01;
    else
        if ( key == '-' )
            offs -= 0.01;
}

void animate ()
{
    angle = 0.004f * glutGet ( GLUT_ELAPSED_TIME );

    light.x = 5*cos ( angle );
    light.y = 4*sin ( 1.4 * angle );
    light.z = 5 + 0.7 * sin ( angle / 3 );

    program.bind ();
    program.setUniformVector ( "eyePos", Vector4D ( eye, 1 ) );
    program.setUniformVector ( "lightPos", Vector4D ( light, 1 ) );
    program.setUniformFloat ( "offs", offs );
    program.unbind ();

    glutPostRedisplay ();
}

int main ( int argc, char * argv [] )
```

```
{  
    // initialize glut  
    glutInit      ( &argc, argv );  
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );  
    glutInitWindowSize ( 500, 500 );  
  
    // create window  
    glutCreateWindow ( "Example of GLSL eroded shader" );  
  
    // register handlers  
    glutDisplayFunc ( display );  
    glutReshapeFunc ( reshape );  
    glutKeyboardFunc ( key );  
    glutMouseFunc   ( mouse );  
    glutMotionFunc  ( motion );  
    glutIdleFunc    ( animate );  
  
    init          ();  
    initExtensions ();  
    printfInfo    ();  
  
    noiseMap = createTexture3D ( false,  
                                ".../Textures/Noise/noise-3d.dds" );  
    assertExtensionsSupported ( "GL_ARB_shading_language_100 GL_ARB_shader_objects" );  
    if ( !program.loadShaders ( "eroded.vsh", "eroded.fsh" ) )  
    {  
        printf ( "Error loading shaders:\n%s\n",  
                 program.getLog().c_str () );  
        return 3;  
    }  
  
    program.bind      ();  
    program.setTexture ( "noiseMap", 0 );  
    program.unbind    ();  
  
    printf (
```

```
"Eroded shader\nUse + and - keys to change erosion offset\n" );  
  
glutMainLoop ();  
  
return 0;  
}
```

Несмотря на то что данные шейдеры дают красивую картину постепенного разрушения объекта, края объектов не имеют никаких следов "ржавчины". Это можно слегка изменить — достаточно в тех случаях, когда значение функции *turbulence* приближается к пороговому значению на некоторое минимальное расстояние, просто сделать цвет темнее, чем ближе значение к пороговому.

Именно этот подход и реализован в следующем фрагментном шейдере, приведенном в листинге 17.12.

Листинг 17.12. Фрагментный шейдер улучшенной процедуры "ржавления"

```
//  
// Eroded fragment shader  
//  
  
varying vec3 l;  
varying vec3 h;  
varying vec3 v;  
varying vec3 n;  
varying vec3 pp;  
  
uniform sampler3D noiseMap;  
uniform float offs;  
  
#include "ProcFuncs.h"  
  
void main (void)  
{  
    const vec4 diffColor = vec4 ( 0.5, 0.0, 0.0, 1.0 );  
    const vec4 specColor = vec4 ( 0.7, 0.7, 0.0, 1.0 );  
    const float specPower = 30;  
    const float delta     = 0.3;
```

```
vec3 ns      = turbulence ( pp, 2.07193 );
float factor = (ns.x + ns.y + ns.z) / 3;

if ( factor < offs )
    discard;

vec3 n2      = normalize ( n );
vec3 l2      = normalize ( l );
vec3 h2      = normalize ( h );
vec4 diff = max ( dot ( n2, l2 ), 0.0 ) + 0.3;
vec4 spec = specColor * pow ( max ( dot ( n2, h2 ), 0.0 ),
                               specPower );
float f      = smoothstep ( offs, offs + delta, factor );

gl_FragColor = f * ( diff * diffColor + spec );
}
```

Изображение "ржавого чайника" на рис. 17.8 получено с помощью шейдера из листинга 17.12.

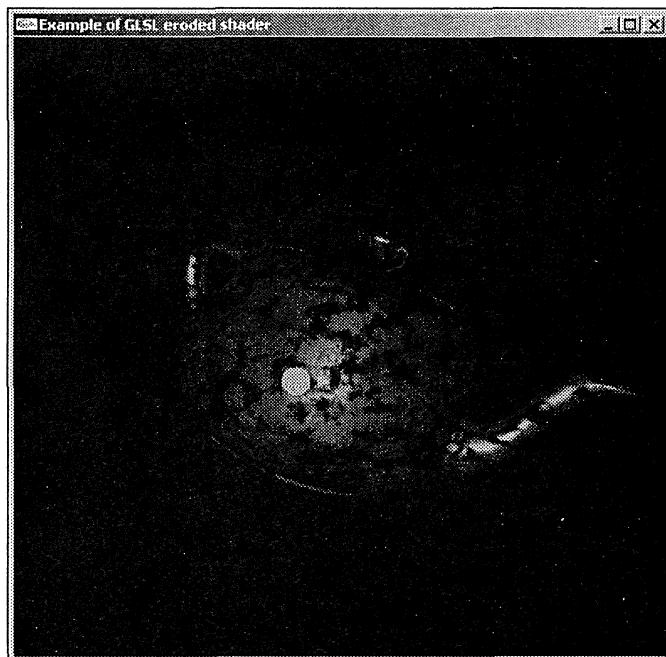


Рис. 17.8. "Ржавый чайник"

Рябь на воде

Еще одним красивым приложением для функции `turbulence` является создание эффекта ряби на воде. Для этого, подобно анимированию "мыльных пузырей", необходимо сделать аргумент функции `turbulence` зависящим как от пространственных координат, так и от времени — это обеспечит эффект движущихся волн. Полученное значение этой функции используется для искажения вектора нормали к поверхности воды для каждого фрагмента. По отклоненному вектору нормали строится отраженный вектор, по которому осуществляется обращение к кубической карте.

Единственным моментом, отличающим конкретную реализацию, приведенную в листингах 17.13 и 17.14, является необходимость повернуть текстурную карту, для перемещения "воды" "вниз". Этот поворот должен быть компенсирован в шейдерах.

Листинг 17.13. Вершинный шейдер, реализующий "рябь на воде"

```
//  
// Vertex shader for ripple effect  
//  
uniform vec4 eyePos;  
varying vec3 n;  
varying vec3 e;  
varying vec3 p;  
  
void main(void)  
{  
    // we have to compensate for texture coordinate transform  
    mat4 rot = mat4( 1, 0, 0, 0,  
                    0, 0, -1, 0,  
                    0, 1, 1, 0,  
                    0, 0, 0, 1 );  
  
    vec3 pos = vec3( gl_ModelViewMatrix * gl_Vertex );  
  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
    n = normalize( gl_NormalMatrix * gl_Normal ) * mat3( rot );  
    e = mat3( rot ) * normalize( eyePos - gl_Vertex.xyz );  
    p = gl_Vertex.xyz;  
}
```

Листинг 17.14. Фрагментный шейдер, реализующий "рябь на воде"

```
//  
// Fragment shader for ripple effect  
  
uniform samplerCUBE envMap;  
uniform sampler3D noiseMap;  
uniform float time;  
varying vec3 n;  
varying vec3 e;  
varying vec3 p;  
  
#include "ProcFuncs.h"  
  
void main (void)  
{  
    const float eta = 0.9;  
  
    // Compute reflection vector  
    vec3 arg = 0.5*p+vec3 ( 0.04101, -0.0612149, 0.071109 )*time;  
    vec3 ns = turbulence ( arg, 2.17 );  
    vec3 en = normalize ( e );  
    vec3 nn = normalize ( n + 0.3 * ns );  
    vec3 r = reflect ( en, nn );  
  
    r.x = -r.x;  
  
    // Do a lookup into the environment map.  
    vec3 envColor = textureCube ( envMap, r ).rgb;  
  
    gl_FragColor = vec4 ( envColor, 1.0 );  
}
```

Соответствующий текст программы на C++ находится на сопроводительном компакт-диске книги в каталоге Code\Chapter-17.

Облака

Еще одним частым применением функции *turbulence* является моделирование облаков на небе. Для этого обычно для каждого фрагмента вычисляется

значение этой функции (зависящее как от положения, так и от текущего времени). Если полученное значение меньше некоторого порога (*stage1*), то считается, что в этой точке небо чистое и облаков нет. Если значение функции больше второго порогового значения (*stage2*), то считается, что данная точка полностью закрыта облаками. Во всех остальных случаях считается, что фрагмент частично закрыт облаками и его цвет является взвешенной суммой цвета неба и облаков.

Таким образом, цвет задается следующей формулой:

$$C(p, t) = \text{mix}(C_{\text{sky}}, C_{\text{cloud}}, \text{smoothstep}(\text{stage1}, \text{stage2}, \text{turbulence}(p, t))).$$

Обратите внимание, что за счет изменения значений *stage1* и *stage2* можно управлять плотностью облачного покрова.

Сам цвет неба можно получить путем интерполяции между значениями нормального цвета неба и цвета неба в точке солнца при помощи косинуса угла между направлениями на фрагмент и на солнце.

Этот подход реализуется шейдерами, приведенными в листингах 17.15 и 17.16.

Листинг 17.15. Вершинный шейдер для рендеринга "облаков на небе"

```
//  
// Vertex shader for clouds and sky  
  
varying vec3 v;  
varying vec3 l;  
varying vec3 pos;  
  
uniform vec4 sunPos;  
uniform vec4 eyePos;  
  
void main(void)  
{  
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );  
  
    v = vec3 ( gl_ModelViewMatrix * eyePos ) - p;  
    l = vec3 ( gl_ModelViewMatrix * sunPos ) - p;  
    pos = 0.1 * gl_Vertex.xyz;  
  
    gl_Position      = gl_ModelViewProjectionMatrix * gl_Vertex;  
    gl_TexCoord [0] = gl_MultiTexCoord0;  
}
```

Листинг 17.16. Фрагментный шейдер для рендеринга "облаков на небе"

```
//  
// Fragment shader for clouds and sky  
  
varying vec3 pos;  
varying vec3 v;  
varying vec3 l;  
  
uniform float cloudDensity;  
uniform float time;  
uniform sampler3D noiseMap;  
  
#include "ProcFuncs.h"  
  
vec4 getSkyColor ( in vec3 v )  
{  
    const vec4 skyColorLight = vec4 ( 0.6, 0.9, 1.0, 1.0 );  
    const vec4 skyColorNormal = vec4 ( 0.0, 0.5, 0.6, 1.0 );  
    const vec3 topDir = vec3 ( 0.0, 0.0, 1.0 );  
  
    float f1 = max ( 0.0, dot ( v, topDir ) );  
    float f2 = max ( 0.0, dot ( v, normalize ( l ) ) );  
    vec4 skyColor = mix ( skyColorNormal, skyColorLight,  
                         pow ( f2, 2.0 ) );  
  
    return skyColor;  
}  
  
void main ()  
{  
    const vec4 cloudColor = vec4 ( 1.0, 1.0, 1.0, 1.0 );  
    const vec3 maxLight = vec3 ( 1.0, 0.7, 2.1 );  
  
    vec3 arg = pos+vec3 ( 0.007*time, 0.006*time, 0.009*time );  
    vec3 turb = 0.5*(turbulence ( arg, 2.137 ) + 1.0);  
    vec4 skyColor = getSkyColor ( normalize ( v ) );  
  
    // now compute clouds based on n
```

```
float factor = smoothstep ( 0.6, 0.9, turb.x + cloudDensity );
vec4 cloud = mix ( skyColor, cloudColor, factor );

gl_FragColor = cloud;
}
```

На рис. 17.9 отображено "небо с облаками", полученное при помощи этих шейдеров.



Рис. 17.9. "Небо с облаками"

Мрамор

Простейший подход к моделированию каменной (мраморной, гранитной и т. п.) поверхности заключается в использовании функции *turbulence* (или *fBm*, или *multifractal*) для смешивания двух цветов:

$$C_{\text{out}} = \text{mix} (C_1, C_2, \text{turbulence}(p)). \quad (17.4)$$

Именно этот подход и реализован в следующих двух шейдерах, приведенных в листингах 17.17 и 17.18.

Листинг 17.17. Простейший вершинный шейдер для получения "мрамора"

```
//  
// Simplest marble vertex shader  
  
varying vec3 l;  
varying vec3 h;  
varying vec3 n;  
varying vec3 v;  
varying vec3 pp;  
  
uniform vec4 lightPos;  
uniform vec4 eyePos;  
  
void main(void)  
{  
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );  
  
    l = normalize ( vec3 ( lightPos ) - p );  
    v = normalize ( vec3 ( eyePos ) - p );  
    h = normalize ( l + v );  
    n = normalize ( gl_NormalMatrix * gl_Normal );  
    pp = gl_Vertex.xyz;  
  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

Листинг 17.18. Простейший фрагментный шейдер для получения "мрамора"

```
//  
// Simplest marble fragment shader  
  
varying vec3 l;  
varying vec3 h;  
varying vec3 v;  
varying vec3 n;  
varying vec3 pp;  
  
uniform sampler3D noiseMap;
```

```
#include "ProcFuncs.h"

void main (void)
{
    const vec4 specColor = vec4 ( 1.0 );
    const float specPower = 30;

    vec3 turb = turbulence ( 0.2*pp, 2.17 );
    float f = (turb.x + 1.0) / 0.8;
    vec3 color = f * vec3 ( 0.5, 0.4, 0.5 ) +
        (1 - f) * vec3 ( 0.1, 0.2, 0.1 );
    vec3 n2 = normalize ( n + 0.6*turb );
    vec3 l2 = normalize ( l );
    vec3 h2 = normalize ( h );
    float diff = 0.6 + max ( dot ( n2, l2 ), 0.0 );
    vec4 spec = specColor * pow ( max ( dot ( n2, h2 ), 0.0 ),
        specPower );

    gl_FragColor.rgb = diff * color + spec.rgb * f;
    gl_FragColor.a = 1;
}
```

Более интересный подход заключается в некотором усложнении формулы (17.4) — в качестве параметра для смешения цветов выступает некоторая периодическая функция (с значениями из промежутка [0, 1]) от комбинации координат и значения функции *turbulence*. Ниже приводится один из вариантов усложненной формулы:

$$t = \sin(\text{Wave}(p.x + \text{turbScale} \cdot \text{turbulence}(p))) \quad (17.5)$$

$$C = \text{mix}(C_1, C_2, t).$$

С помощью параметра *turbScale* можно управлять степенью "случайности" получающегося материала.

Именно этот подход и реализован в следующих двух шейдерах, приведенных в листингах 17.19 и 17.20 (рис. 17.10).

Листинг 17.19. Более сложный вершинный шейдер для получения "мрамора"

```
// Advanced vertex shader for marble
//
varying vec3 l;
```

```
varying vec3 h;
varying vec3 n;
varying vec3 v;
varying vec3 pp;

uniform vec4 lightPos;
uniform vec4 eyePos;

void main(void)
{
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );

    l = normalize ( vec3 ( lightPos ) - p );
    v = normalize ( vec3 ( eyePos ) - p );
    h = normalize ( l + v );
    n = normalize ( gl_NormalMatrix * gl_Normal );
    pp = gl_Vertex.xyz;

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Листинг 17.20. Более сложный фрагментный шейдер для получения "мрамора"

```
//  
// Advanced fragment shader for marble  
//  
varying vec3 l;  
varying vec3 h;  
varying vec3 v;  
varying vec3 n;  
varying vec3 pp;  
uniform float turbScale;  
  
uniform sampler3D noiseMap;  
  
#include "ProcFuncs.h"  
  
void main (void)  
{  
    const vec4 specColor = vec4 ( 1.0 );  
    const float specPower = 30;
```

```
const float displacement = 0.7935;

vec3 turb = turbulence ( 0.2*pp, 2.17 );
float t = pp.x + turbScale * turb.x - displacement;
float f = 0.5 * ( 1.0 + sin ( 3.141516923 * t ) );
vec3 color = f * vec3 ( 0.5, 0.4, 0.5 ) +
             (1 - f) * vec3 ( 0.1, 0.2, 0.1 );
vec3 n2 = normalize ( n + 0.6*turb );
vec3 l2 = normalize ( l );
vec3 h2 = normalize ( h );
float diff = 0.6 + max ( dot ( n2, l2 ), 0.0 );
vec4 spec = specColor * pow ( max ( dot ( n2, h2 ), 0.0 ),
                               specPower );

gl_FragColor.rgb = diff * color + spec.rgb * f;
gl_FragColor.a = 1;
}
```

Как видно из листинга 17.20, кроме управления цветом, полученное значение функции `turbulence` используется для искажения вектора нормали и управления яркостью бликов.

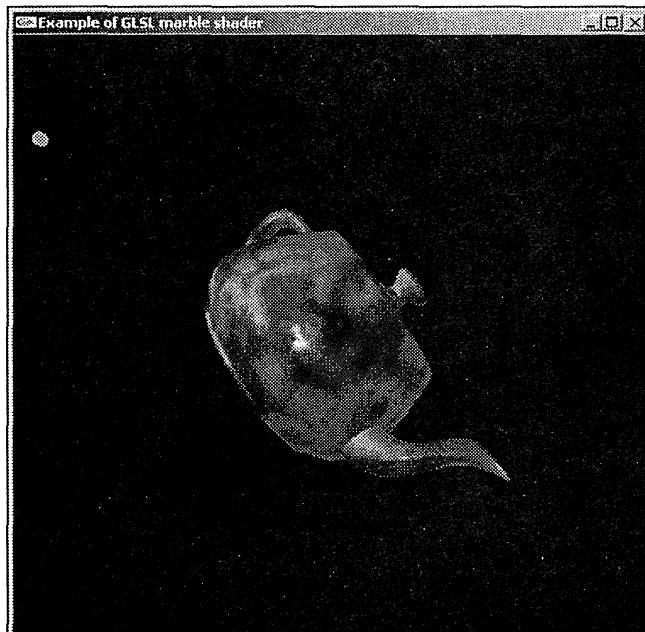


Рис. 17.10. "Улучшенный мрамор"

Если в функции (17.5) параметр t перед его использованием для смешения цветов возвести в некоторую (больше 1) степень, то это приведет к уменьшению доли одного из цветов — он будет присутствовать в материале в виде тонких прожилок. Именно этот эффект используется в следующем фрагментном шейдере для создания "мрамора с прожилками".

Листинг 17.21. Фрагментный шейдер для получения "мрамора с прожилками"

```

//  

// Fragment shader for "veined" marble  

//  

varying vec3 l;  

varying vec3 h;  

varying vec3 n;  

varying vec3 v;  

varying vec3 pp;  

uniform sampler3D noiseMap;  

#include "ProcFuncs.h"  

void main (void)  

{  

    const vec3 baseColor = vec3 ( 0.062745, 0.082386, 0.440000 );  

    const vec3 veinColor = vec3 ( 0.953333, 0.915831, 0.837338 );  

    const vec4 specColor = vec4 ( 1.0 );  

    const float specPower = 30;  

    vec3 turb = turbulence ( 0.4 * pp, 2.0913 );  

    float t = 0.5 * ( 1.0 + sin ( turb.x * 3.1415926 * 5 ) );  

    vec3 n2 = normalize ( n + 0.6*turb );  

    vec3 l2 = normalize ( l );  

    vec3 h2 = normalize ( h );  

    float diff = 0.6 + max ( dot ( n2, l2 ), 0.0 );  

    vec4 spec = specColor * pow ( max ( dot ( n2, h2 ), 0.0 ),  

                                specPower );  

    vec3 color = mix ( baseColor, veinColor, t*t*t*t );  

    gl_FragColor = vec4 ( diff * color + spec.rgb, 1 );
}

```



Рис. 17.11. "Мрамор с прожилками", полученный при помощи шейдера из листинга 17.21

Более впечатляющие результаты можно получить за счет использования не двух цветов — C_1 и C_2 , а сразу целой группы цветов. Именно так и делалось в языке *RenderMan Shading Language* — в нем есть встроенная функция *colorspline*, позволяющая по заданному упорядоченному набору пар (*color, param*) находить путем интерполяции значения для произвольного скалярного параметра. В языке GLSL подобной функции нет, но она и не нужна — самым быстрым способом вычисления цветового сплайна является задание его как одномерной (1D) текстуры. Тогда получение необходимого значения цвета сводится к обычной операции обращения к текстуре.

На сопроводительном компакт-диске книги в каталоге Code\scripts находится скрипт на языке Python, строящий по набору пар (*color, param*) соответствующую одномерную текстуру.

Дерево

При моделировании деревянной поверхности используется почти тот же подход, что и при моделировании поверхности камня, только при этом

необходимо учесть структуру дерева в виде годовых колец. Простейший вариант при этом реализуется следующей формулой:

$$t = \sqrt{x^2 + y^2} + turbScale \cdot turbulence(p).$$

После вычисления значения параметра t по этой формуле, к нему применяется какая-либо периодическая функция (с областью значений, совпадающей с отрезком $[0, 1]$), при необходимости осуществляется возведение в степень и полученный таким образом результат используется для смешения двух цветов.

Параметр $turbScale$ позволяет управлять величиной искажений, вносимых в структуру годовых колец дерева.

Именно этот подход и реализован в листингах 17.22 и 17.23.

Листинг 17.22. Вершинный шейдер для рендеринга "деревянной" поверхности

```
//  
// Vertex shader for wood  
  
varying vec3 l;  
varying vec3 h;  
varying vec3 n;  
varying vec3 v;  
varying vec3 pp;  
  
uniform vec4 lightPos;  
uniform vec4 eyePos;  
  
void main(void)  
{  
    vec3 p = vec3 ( gl_ModelViewMatrix * gl_Vertex );  
  
    l = normalize ( vec3 ( lightPos ) - p );  
    v = normalize ( vec3 ( eyePos ) - p );  
    h = normalize ( l + v );  
    n = normalize ( gl_NormalMatrix * gl_Normal );  
    pp = gl_Vertex.xyz;  
  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

Листинг 17.23. Фрагментный шейдер для рендеринга "деревянной" поверхности

```
//  
// Fragment shader for wood  
  
varying vec3 l;  
varying vec3 h;  
varying vec3 v;  
varying vec3 n;  
varying vec3 pp;  
  
uniform sampler3D noiseMap;  
  
#include "ProcFuncs.h"  
  
void main (void)  
{  
    const vec4 specColor = vec4 ( 1.0 );  
    const vec3 darkWood = vec3 ( 0.47, 0.25, 0.11 );  
    const vec3 liteWood = vec3 ( 0.75, 0.58, 0.38 );  
    const float specPower = 30;  
    const float freq = 7.0;  
    const float noiseFreq = 2.5;  
    const float squeeze = 3;  
  
    vec3 snoise = 2.0*texture3D ( noiseMap, pp ).rgb - vec3 ( 1.0 );  
    float ring = smoothToothWave ( freq * length ( pp.yz ) +  
                                  noiseFreq * snoise.x );  
    float f = pow ( ring, squeeze ) + snoise.x;  
  
    vec3 color = mix ( darkWood, liteWood, f );  
    vec3 n2 = normalize ( n );  
    vec3 l2 = normalize ( l );  
    vec3 h2 = normalize ( h );  
    float diff = 0.6 + max ( dot ( n2, l2 ), 0.0 );  
    vec4 spec = specColor * pow ( max ( dot ( n2, h2 ), 0.0 ),  
                                specPower );  
  
    gl_FragColor.rgb = diff * color + spec.rgb*(1-f);  
    gl_FragColor.a = 1;  
}
```

На рис. 17.12 отображен "Чайник", полученный при помощи этих шейдеров.

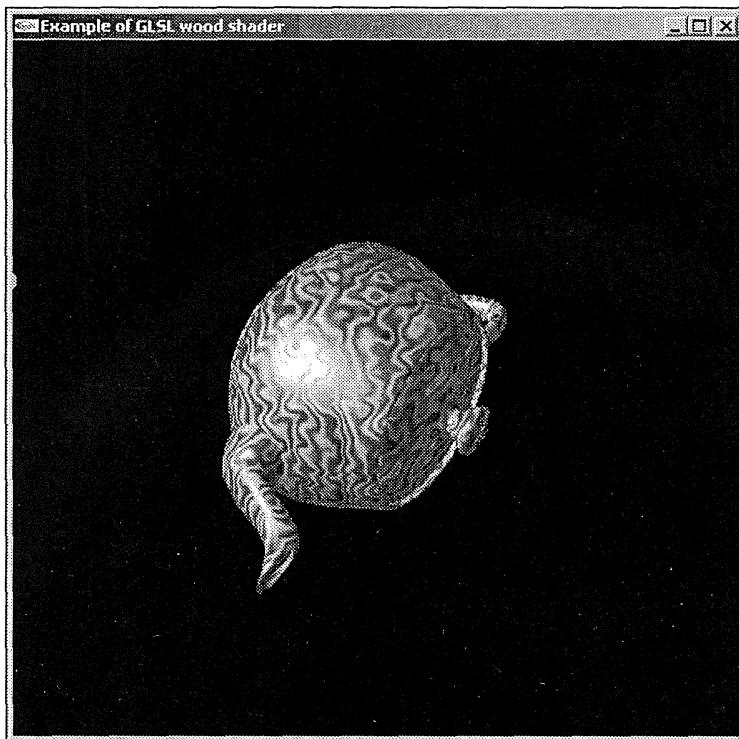


Рис. 17.12. "Деревянный чайник," полученный при помощи шейдеров из листингов 17.22 и 17.23

Система "сверкающих" частиц

За счет использования шумовой функции можно довольно легко реализовать еще один красивый эффект — заставлять отдельные частицы системы случайным образом изменять свою интенсивность — то ярко вспыхивать, то почти полностью гаснуть. Все, что для этого требуется, — получить от вершинного шейдера текущие координаты частицы и текущее время. Применив к ним после этого какую-нибудь суперпозицию, можно извлечь из шумовой текстуры значение.

Далее, если x -компонент (на самом деле подойдет любой компонент или их комбинация) полученного значения окажется меньше некоторого порогового значения, то соответствующая частица считается невидимой (либо отбрасывается, либо значение ее цветового или альфа-компонента устанавливается в ноль). Если x -компонент оказывается больше второго

порогового значения, то частица считается полностью видимой. Для всех промежуточных значений интенсивность частицы вычисляется путем линейной интерполяции.

Тем самым мы приходим к следующей формуле, дающей интенсивность частицы с учетом "сверкания":

$$k = smoothstep(val1, val2, noise(p, t).x).$$

Введение подобного эффекта потребует минимального изменения вершинного и фрагментного шейдеров для системы частиц, рассмотренной в гл. 16 (листинги 17.24 и 17.25).

Листинг 17.24. Вершинный шейдер для системы "сверкающих" частиц

```
//  
// Vertex shader for "shiny" particles  
  
/  
uniform float time;  
varying vec4 color;  
  
void main(void)  
{  
    vec3 vel    = 3*gl_MultiTexCoord1.xyz;  
    vec3 pos    = gl_Vertex.xyz;  
    float phase = gl_MultiTexCoord1.w;  
    float t      = mod ( time + phase, 4.0 );  
  
    // now compute new position using t and org.  
    vec3 org = pos + t*vel - 0.5*t*t*vec3 ( 0.0, 0.0, 1.0 );  
  
    color      = mix ( vec4 ( 1.0, 0.0, 0.0, 1.0 ),  
                      vec4 ( 0.3, 0.3, 0.0, 1.0 ), t * 0.25 );  
    gl_Position = gl_ModelViewProjectionMatrix*vec4 ( org, 1.0 );  
    gl_TexCoord [0] = gl_MultiTexCoord0;  
}
```

Листинг 17.25. Фрагментный шейдер для системы "сверкающих" частиц

```
//  
// Fragment shader for "shiny" particles  
//
```

```
uniform sampler2D decalMap;
uniform sampler3D noiseMap;
uniform float time;
varying vec3 pos;

void main (void)
{
    vec4 noise = texture3D ( noiseMap, 3*pos +
                           vec3 ( 0.802, 0.7005, 0.711 ) * time );
    float t      = noise.x;

    t = smoothstep ( 0.3, 0.9, t );

    gl_FragColor = 2.0 * t * texture2D ( decalMap, gl_TexCoord [0].xy );
}
```

Огонь

Функцию *turbulence* можно также с успехом использовать и для моделирования огня. Обычно огонь представляется при помощи прямоугольника, всегда ориентированного параллельно плоскости экрана (*billboard*). Задача заключается в написании правильного шейдера, точнее, в правильном подборе "температуры пламени" для каждой точки. Располагая температурой, можно из одномерной текстуры извлечь соответствующий цвет.

Функция *turbulence* хорошо подходит для внесения случайности в "пламя", однако обычно степень ее влияния ограничивается с учетом текущей координаты по вертикали — это позволяет снизить случайность в наиболее ярких областях и повысить ее в верхней части "пламени".

Таким образом, мы вначале берем простой цветовой градиент — плавный переход температуры от максимума (внизу) до минимума (вверху) и искааем "температуру" при помощи контролируемого применения функции *turbulence*.

На рис. 17.13 приведен пример "пламени", полученного таким способом. Соответствующие шейдеры приведены в листингах 17.26 и 17.27.



Рис. 17.13. Процедурное "пламя"

Листинг 17.26. Вершинный шейдер для моделирования "пламени"

```
//  
// Vertex flame shader  
//  
varying vec3 p;  
  
void main(void)  
{  
    p = vec3 ( gl_ModelViewMatrix * gl_Vertex );  
    gl_TexCoord [0] = gl_MultiTexCoord0;  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

Листинг 17.27. Фрагментный шейдер для моделирования "пламени"

```

//  

// Fragment flame shader  

//  

varying    vec3      p;  

uniform    sampler3D noiseMap;  

uniform    sampler1D flameMap;  

uniform    float      time;  

#include  "ProcFuncs.h"  

void main (void)  

{  

    const vec4    specColor = vec4 ( 1.0 );  

    const float   specPower = 30;  

    vec3  pos   = vec3 ( 2*p.x / 512.0, p.y / 512.0, 0.12759 );  

    vec2  tex   = gl_TexCoord [0].xy;  

    vec3  turb = turbulence ( pos + 4*  

                            vec3 ( 0.03*time, 0.009*time, 0.07*time ), 2.17 );  

    float wobble = 0.3 + 0.9*tex.y;  

    float heat   = 1.1 - tex.y + wobble * turb.x;  

    vec3  color  = texture1D ( flameMap, min ( 0.9, heat ) );  

    if ( heat < 0.01 )  

        color = vec3 ( 0.0 );  

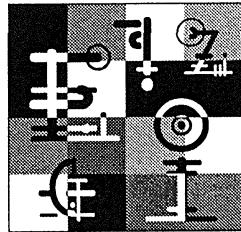
  

    gl_FragColor = vec4 ( color.rgb, 1.0 );  

}

```

Обратите внимание на постоянное использование одного и того же приема — по пространственным координатам (поэтому данный подход иногда называется solid texturing) вычисляются значения шумовой функции (или функции *turbulence*, или функции *fBm*) и по ним (зачастую вместе с некоторой характерной для материала функцией координат) получается скалярное значение. Данное скалярное значение преобразуется в цвет (посредством простого смешения двух цветов или использования одномерной текстуры).



Приложение

Описание компакт-диска

Сопроводительный компакт-диск книги содержит все исходные тексты программ и библиотек, а также необходимые текстуры и модели для примеров, приведенных в книге.

В каталоге WINDOWS-BIN находятся откомпилированные выполнимые файлы всех примеров, готовые к выполнению.

В каталоге LINUX-BIN находятся откомпилированные версии программ для платформы Linux в виде файла TAR.GZ. После его разархивирования программы могут сразу же быть выполнены.

Все примеры рассчитаны на компиляцию и выполнение под управлением как Microsoft Windows, так и Linux. Для сборки примеров под Windows можно использовать как файлы MAKEFILE.NMAKE, так и проекты для MS Visual C++ 6.0.

Для сборки примеров под Windows с использованием файла MAKEFILE.NMAKE следует в каталоге, относящемся к соответствующей главе книги, выполнить команду:

```
make - f Makefile.nmake
```

Сборка всех примеров для платформы Linux выполняется с помощью make-файлов. Для этого нужно выполнить команду:

```
make
```

Примечание

Для выполнения ряда примеров могут понадобиться последние версии драйверов для вашего графического процессора и операционной системы.

Последние версии всех программ и примеров, а также исправления и дополнения вы можете найти на сайте автора книги www.steps3d.narod.ru.

Связаться с автором можно по электронному адресу steps3d@narod.ru.

Список литературы и интернет-ресурсов

1. Боресков А. В. Расширения OpenGL. — СПб.: БХВ-Петербург, 2005.
2. Боресков А. В. Графика трехмерной компьютерной игры на основе OpenGL. — М.: Диалог-МИФИ, 2004.
3. Гайдуков С. А. OpenGL. Профессиональное программирование трехмерной графики на C++. — СПб.: БХВ-Петербург, 2004.
4. Шикин Е. В., Боресков А. В. Компьютерная графика. Полигональные модели. М.: Диалог-МИФИ, 2000.
5. Ву М. OpenGL. Официальное руководство программиста. — СПб: ДиаСофтЮП, 2002.
6. OpenGL.Официальный справочник. — СПб.: ДиаСофтЮП, 2002.
7. Хилл Ф. OpenGL. Программирование компьютерной графики. — СПб.: Питер, 2002.
8. Тихомиров Ю. В. Программирование трехмерной графики. — СПб.: ВНВ — Санкт-Петербург, 1998.
9. Краснов М. OpenGL. Графика в проектах Delphi. — СПб.: ВНВ — Санкт-Петербург, 2000.
10. Эйнджеял Э. Интерактивная компьютерная графика. Вводный курс на базе OpenGL. — М.: Вильямс, 2001.
11. Роджерс Д. Математические основы машинной графики. — М.: Мир, 2001.
12. Ким Г. Д., Ильин В. А. Линейная алгебра и аналитическая геометрия. — М.: Изд-во Московского университета, 1998.
13. Никиulin Е. А. Компьютерная геометрия и алгоритмы машинной графики. — СПб.: ВНВ — Санкт-Петербург, 2003.
14. Горнаков С. Г. Инструментальные средства программирования и отладчики шейдеров в DirectX и OpenGL. — СПб.: БХВ-Петербург, 2005.

15. <http://www.opengl.org>
16. <http://developer.nvidia.com>
17. <http://www.ati.com/developer>
18. <http://www.steps3d.narod.ru>
19. <http://www.gamedev.ru>
20. <http://www.gamedev.net>
21. <http://nehe.gamedev.net>
22. <http://www.frustum.org>
23. <http://www.shadertech.com>
24. <http://www.clockworkcoders.com/oglsl/tutorials.html>
25. <http://www.3dshaders.com>
26. <http://www.humus.ca>
27. <http://www.paulsprojects.net>

Предметный указатель

3

3DS 4, 235

A

Aliasing artifacts 15

Amy Gooch 299

ARB_depth_texture 17, 21

ARB_draw_buffers 70

ARB_fragment_program 3, 24, 45

ARB_fragment_shader 19, 68

ARB_multisample 15, 21

ARB_multitexture 15, 21

ARB_occlusion_query 19, 22

ARB_point_parameters 21, 262

ARB_point_sprite 70, 389

ARB_shader_objects 19, 68

ARB_shading_language_100, 68

ARB_shadow 17, 21

ARB_texture_border_clamp 16, 21

ARB_texture_compression 15

ARB_texture_cube_map 15, 21

ARB_texture_env_add 16, 21

ARB_texture_env_combine 16, 21

ARB_texture_env_crossbar 18, 22

ARB_texture_env_dot3 16, 21

ARB_texture_mirrored_repeat 19, 22

ARB_texture_non_power_of_two 70, 412

ARB_transpose_matrix 16, 21

ARB_vertex_buffer_object 19, 22, 216

ARB_vertex_program 3, 24, 30, 45

ARB_vertex_shader 19, 68

ARB_window_pos 19, 22

ARB-расширения 15

ASE 4, 119, 234

Aspect ratio 186

ATI 71

ATI_separate_stencil 70

Axis Aligned Bounding Box,
AABB 368

B

BGRA, формат 13

Billboard 474

Bind 155, 167

Blending 14, 24

Blur 371, 425

BMP 142

Bump map 287, 421

C

Callback-функции 84

Cg (C for graphics) 3, 44, 243

Cg CgGL- и pthread-библиотеки 47

Cggl.lib 47

Cglib 47

Chunk 235

Convolution 14

Copy-конструктор 116, 118, 186

Cube texture maps 142

Cubic environment map 394

Current menu 75

Current window 75

D

Dataflow 1
 DDS 142, 147, 148, 431
 DevIL-библиотека 142
 DIB, формат 13
 DMA, Direct Memory Access 11
 Dot 254, 269

E

Edge detect 420
 EXT_bgra 13, 20
 EXT_blend_color 14, 21
 EXT_blend_func_separate 17, 21
 EXT_blend_logic_op 11, 20
 EXT_blend_minmax 14, 21
 EXT_blend_subtract 14, 21
 EXT_color_subtable 14, 20
 EXT_color_table 14, 20
 EXT_convolution 14, 20
 EXT_copy_texture 12, 20
 EXT_draw_range_element 14
 EXT_fog_coord 17, 21
 EXT_framebuffer_object 151, 165, 173
 EXT_packed_pixels 13, 20
 EXT_polygon_offset 11, 20
 EXT_rescale_normal 13, 20
 EXT_secondary_color 17, 21
 EXT_separate_specular_color 20
 EXT_shadow_funcs 19, 22
 EXT_stencil_two_side 70
 EXT_stencil_wrap 18, 21
 EXT_subtexture 12
 EXT_texture 12, 20
 EXT_texture_lod_bias 19, 22
 EXT_texture_object 12, 20
 EXT_texture3D 13, 20
 EXT_vertex_array 11, 20

F

FBO 166
 FOV, Field Of View 179, 366
 Fractal Brownian motion 444

Fragment program 2

Fragment shader 2

Framebuffer 149

Framebuffer object FBO 166

Fresnel coefficient 396

Frustum, класс 198

G

GL, расширения 140
 GL_ARB_fragment_program 3
 GL_ARB_fragment_shader 310
 GL_ARB_vertex_program 3
 GL_ARB_vertex_shader 308
 GL_COLOR_MATRIX 14
 GL_EXT_fog_coord 139
 GL_RGBA8 12
 GL_SGIS_generate_mipmap 155
 GLAUX библиотека 142
 GLEW libExt библиотеки 140
 GLH, OpenGL Helper Library 96
 GLSL, OpenGL Shading
 Language 67, 276, 348
 GLUT-библиотека 4, 75
 Glut_interactor-класс 106
 GLX_SGIX_fbconfig 160
 GLX_SGIX_pbuffer 160
 GLX-расширения 141
 Gooch Amy 299
 GPGPU (Generic Programming
 on GPU) 1
 GPU (Graphics Processing Unit) 1

H

Height map 421
 HLSL 348
 HP,_convolution_border_
 modes 14, 20

I

IDE 348
 Idle-обработчик 87
 idSoftware, компания 9

J, K

JPEG 142
JPG 119, 142
Ken Perlin 427

L

Level Of Detail, LOD 271
libTexture и libTexture3D,
библиотеки 142
Log 305
Lookup 14
LWO 4

M

MD3 4, 236
MD5 4, 239
Md5-mesh 239
Mipmap 13
Mipmap-пирамида 169
Modelview 24
Multisample buffer 15

N

Noise 427
Normalized Device Coordinates,
NDC 367
NV_blend_square 17, 21
NV_MATH, библиотека 114
NV_UTIL, библиотека 119
NVIDIA 3, 4, 44, 71, 96, 400
NVIDIA SDK 4
NVIDIA, компания 3

O

OBJ 4
Occlusion queries 19
OpenGL Shading Language,
GLSL 3, 67
OpenGL Utility Toolkit, GLUT 75
OpenGL, библиотека 3
Output, окно 348

P

Particle system 17, 386
Pass 353
p-buffer 3, 151
Perlin noise 427
PNG 142
Polygon offset 11
Pop-up-меню 82
Profile 44
Program object 303
Programmable graphics pipeline 1
Projection 24
Public 162

R

RAR 142
RBO 167
Reference 98, 354
Render State 370
Renderable Texture 371
Renderbuffer object, RBO 167
Rendering pipeline 23
RenderMan Shading
Language 44, 69, 243
Render-To-Texture, RTT 149, 165
RGB 156
RGBA 156, 259, 358
Rim lighting 297
RTT 149, 165

S

Sample 15
Secondary 17
Secondary color 13
Separable 425
Sepia 413
SGI_color_matrix 14, 21
SGIS_generate_mipmap 17, 21
SGIS_texture_edge_clamp 13, 20
SGIS_texture_lod 13, 20
Shader object 68, 303

Slerp 138
 Solid 92
 Solid texturing 476
 Source 17
 Stencil shadow volumes 204
 Subchunk 235

T

T&L 386
 Tangent map 292
 Texture cube maps 15
 Texture environment mode 16
 Texture Object 356
 Texture proxies 12
 TGA 119, 142, 148
 Transform & Lighting 386
 Type-safe 243

A

Анизотропные поверхности 291
 Анимация объектов 386
 Атрибуты 246
 Аффинное преобразование 187, 229

Б

Базис пространства 206, 287, 380
 Библиотека DevIL 142
 Библиотека:
 GLAUX 142
 GLUT 4
 libExt 142
 libTexture 4, 142
 libTexture3D 142
 NV_MATH 114
 NV_UTIL 119
 Бинормаль 287, 292
 Бисектор направления 280
 Блики 13
 контрастность 295
 Блочный компонент 293

V

VBO 22
 Vertex arrays 11
 Vertex attributes 24
 Vertex Buffer Objects VBO 380
 Vertex program 2
 Vertex shader 2

W, Z

Ward 295
 Weighting 24
 WGL_ARB_pbuffer 151
 WGL_ARB_pixel_format 151
 WGL_ARB_render_texture 151, 155
 WGL-расширения 141
 Wire 92
 Workspace 348
 ZIP 119, 142

Блоки 235

Боковое направление 186
 Буфер глубины 88, 152, 161, 172, 173
 Буфер трафарета 17, 70, 88, 152, 161,
 168, 172
 Буфер кадра 149
 Буфер накопления 88, 161
 Буфера 78, 80
 вершинные 216
 цвета 259

В

Вектор 24
 Вектор нормали 13, 217, 287
 Вершинный буфер 216, 218
 Вершинное освещение 24
 Вершинные атрибуты 24
 Вершинные массивы 11, 380
 Вершинные программы
 (шейдеры) 2, 24, 243, 246, 269, 276
 Вершины 204
 Взвешенная сумма цветов 14

Видеорежим, переключение в 155, 162
Видимость окна 87
Виртуальный (пиксельный) буфер 151
Вложенные меню 83
Внутренний формат текстур 11
Всплывающие (popup) меню 82
Вторичный цвет 17
Вывод:
 в окно 154
 в цветовые буфера 70
Выделение контурных линий 297
Выходные регистры 25

Г

Гамма-коррекция 419
Гладкие объекты 287
Глобальные обработчики событий 87
Глубина фрагмента 259
Границы и ребра 204, 287
Графический процессор
 (ускоритель) 1, 9

Д

Данные 142
Двойная буферизация 80
Декодеры 146
Декодирование данных 142
Дисплейные списки 155
Диффузный и блоковый
 компоненты 292, 293
Дополнительный цвет 13
Дочернее окно, создание 79

З

Заголовок текущего окна 81
Запросы на определение видимости 19
Затуманивание вершин 261

И

Идентификатор меню 82
Идентификатор текущего окна 80
Иерархические меню 83

Изотропные поверхности 291
Имя, расширение 10
Индексация 253
Индексация массива 245
Интенсивность 422
Интеракторы 106
Интерполяция 138, 287
Источник 17
Источник данных 142
Итераторы 234

К

Каркасный режим 92
Карта:
 высот 147, 421
 глубин 244
 нормалей 147, 421, 287
 касательных 292
Касательное пространство 287
Касательный вектор 287, 292
Кватернион 101, 118, 180, 187
Кен Перлин 427
Клавиатурное событие 85
Коллинеарные векторы 135
Компиляция 256
Компланарные векторы 135
Конвейер 270
 OpenGL 269
 рендеринга 23, 67
Конструктор копирования 116
Конструкторы 250
Контекст 48
Контекст рендера 79, 153, 155, 165
Контрастность бликов 295
Контурные линии,
 выделение 297, 298
Координатная система,
 ориентация 186
Коэффициент 17, 203
 свертки 425
 Френеля 396
Кубическая карта 155, 459
 окружения 394
 текстурная 15, 142, 169, 244, 272

Л, М

Лог 305
 Массивы 245
 Матрица 117, 203
 поворота 138
 преобразования 193
 проектирования 24
 Меню 82, 83
 Метод теневых объемов 204
 Микрорельеф 287
 Модели освещения
 Моделлер 204
 Модель освещения 264, 276
 Блинна 280, 295, 440
 Уорда 295
 Фонга 295
 Эми Гуч 299
 Модельная матрица 24, 203, 276
 Мультисплайн 15
 Мультитекстурирование 15

Н

Нажатие клавиши 85
 Наложение текстур 15
 режимы 16
 Насыщенность 422
 Нескалярные типы 250
 Новое меню, создание 82
 Нормаль 92, 217, 287
 преобразования 24
 Нормальные окна 79

О

Область видимости 203, 245
 Образец 15
 Объект 12
 Объекты-шейдеры 68
 Ограничивающее тело 216
 Ограничивающие
плоскости 203
 Окно дочернее, создание 79
 Окно текущее,
идентификатор 80

Окно:

видимость 87
 отображение содержимого 84
 создание 79
 уничтожение 80
 Операции над регистрами 30
Ориентация:
 координатной системы 186
 поверхности 291
 Ортонормализация 193
 Ортонормированный базис 179, 288
 Освещение 278
 Освещенность 13
 Освобождение от текстуры 155
 Основной цвет 13
 Отображение содержимого окна 84
Отсечение:
 объектов 203
 текстурных координат 13, 16, 19

П

Параллельная обработка 2
 Параметры состояния 370
 Перегрузка процессора 203
 Переключение видеорежима 162
 Перерисовка 80
 Периодические текстуры 19, 431
 Пиксельный буфер 151
 Пирамида видимости 198
 усеченная 179
 Пирамидальное фильтрование 13, 155, 167, 170, 372
 Плоскости отсечения 198, 258
 Плоскость экрана 474
 Погрешности дискретизации 15
 Подсветка края 297
 Покомпонентные операции 253, 269
 Поле касательных векторов 291
 Полигональный объект 234
 Полноэкранный режим 80
 Пометка текущего окна 80
 Предопределенные переменные 365
 Преломленный вектор 269

Преобразования:

нормали 24
текстурных координат 207

Препроцессор 255

Префикс:

платформы 9
производителя (разработчика) 9

Приведение значения к заданному типу 250

Привязка к текстуре 155

Приемник 17

Программируемый графический конвейер 1

Проектирующая матрица 203, 276

Прокси-текстуры 12

Профили 44

Проход 353, 366

Р

Равномерно рассеивающийся свет 277

Разделяемые ядра свертки 425

Размер и формат пикселов 168

Размер массива 245

Размеры текстур 70

Размытие 371, 425

Расстояния отсечения 186

Растеризация 258

Расширения 9

поддержка 141

Регистры 24, 30

Редактор шейдеров 354

Режим 77, 92

RGBA 11

наложения текстуры 16

цветовых индексов 11

Рендербуфер, объект 167

Рендеринг 23, 149, 270

анизотропной поверхности 293, 294

в текстуру 149, 165

контекст 79, 153

с подсветкой края 298

систем частиц 17

С

Свертка 14

Свойства точек 17

Сжатые текстуры 15, 142

Система частиц 386

Скалярные типы 250

Смешение 11, 17

вершин 24

цветов 14

Создание:

дочернего окна 79

меню 82

Сплошной режим 92

Спрайты 70

Срез трехмерной текстуры 169

Ссылка 98, 354

Степень затуманивания 24

вершины 261

Структуры 244

Сферическая линейная

интерполяция 138

Т

Тексел 16

Текстура 169, 216, 244, 287

глубины 17

координаты 24, 261, 287, 288

параметры 271

периодические 19

размеры 70

сжатые 15, 142

трехмерные 142

Текстурный объект 356

Текущее меню 75

Текущее окно 75, 80

Теневые карты 19

Тени 17

Тип текстуры 169

Тон 422

Топологическая информация 2

Топология данных 24

Точка входа 47

Точка, свойства 17

Трехмерная текстура 142, 169, 272
 Туман 12, 17
 Турбулентность 443

У

Углы Эйлера 180
 Удельная плотность световой
 энергии 278

Уничтожение:

 меню 82
 окна 80

Упакованные форматы пикселов 13

Уровень 169

Уровень детализации 271

Усеченная пирамида
 видимости 179, 198

Ф

Фильтр выделения границ 420

Фильтры 412

Формат пикселов 165, 172
 упакованные 13

Форматы 12, 204

Фрагмент 259

Фрагментная программа (шейдер) 2,
 24, 243, 256, 276, 280, 293

Фрактальное броуновское
 движение 444
 Фрактальный 443
 фреймбуфер 1, 11, 12, 14, 15, 77,
 149, 419
 объект 166

Ц, Ч

Цвета 12
 смешение 14
 Цветные материалы 24
 Цветовой буфер 155
 Число компонент 11

Ш

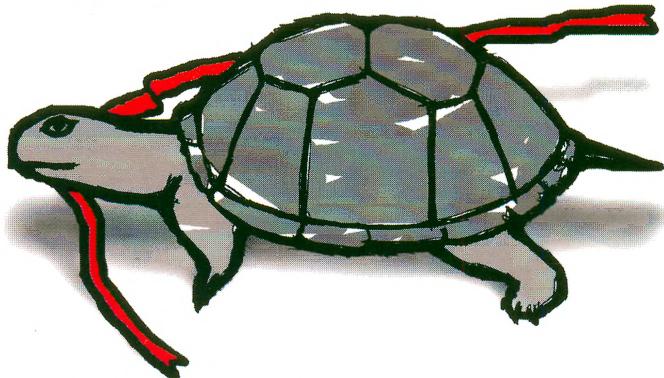
Шейдер (shader) 1
 Шейдерные языки 3
 Шрифты 91
 Шумовые функции 275

Э, Я

Эффекты 353
 Ядро 425

РАЗРАБОТКА И ОТЛАДКА ШЕЙДЕРОВ

Вы узнаете, как создаются сложные
и впечатляющие эффекты в
компьютерных играх
последнего поколения



Боресков Алексей Викторович, кандидат физико-математических наук, один из авторов курса компьютерной графики на факультете вычислительной математики и кибернетики Московского государственного университета им. М. В. Ломоносова. Автор семи книг по компьютерной графике, в том числе книги «Расширения OpenGL», выпущенной издательством «БХВ-Петербург» в 2005 году.

Книга является практическим пособием по разработке кроссплатформенных шейдеров, написанных на языке OpenGL Shader Language (GLSL) для использования в операционных системах Windows и Linux с различными версиями библиотеки OpenGL, ставшей стандартом в обучении компьютерной графике во всем мире. Именно применение шейдеров в ряде последних компьютерных игр позволило реализовать сложные и впечатляющие эффекты в реальном времени. Рассматривается использование пакета RenderMonkey для написания и отладки GLSL-шейдеров, позволяющих создать широкий класс эффектов.

Книга рассчитана как на специалистов в области компьютерной графики, так и на студентов и аспирантов, изучающих данную тему. Кроме того, она может применяться в качестве учебного пособия на курсах по программированию графики.

CD

Компакт-диск содержит кроссплатформенные коды для Microsoft Windows и Linux.

ISBN 5-94157-712-5



9 785941 577125 >



БХВ-ПЕТЕРБУРГ
194354,
ул. Есенина, 55
E-mail: mail@bhv.ru
internet: www.bhv.ru
тел./факс: (812) 591-6243