АЛЕКСЕЙ БОРЕСКОВ

bhv

РАСШИРЕНИЯ OpenGL

- КРОССПЛАТФОРМЕННЫЕ БИБЛИОТЕКИ
- ТРЕХМЕРНЫЕ ТЕКСТУРЫ
- СИСТЕМЫ ЧАСТИЦ
- КУБИЧЕСКИЕ ТЕКСТУРНЫЕ КАРТЫ
- ПОПИКСЕЛЬНОЕ, ДИФФУЗНОЕ И БЛИКОВОЕ ОСВЕЩЕНИЕ
- КАРТЫ ВЫСОТ И НОРМАЛЕЙ
- ШЕЙДЕРЫ, GLSL

PRO
ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

Алексей Боресков

РАСШИРЕНИЯ OpenGL

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.068+800.92
ББК 32.973.26-018.1
Б82

Боресков А. В.

Б82 Расширения OpenGL. — СПб.: БХВ-Петербург, 2005. — 688 с.: ил.

ISBN 5-94157-614-5

Описываются основные и наиболее популярные расширения библиотеки OpenGL, их использование на платформах Windows и Linux. Представлена реализация большого количества эффектов, созданных с помощью этих расширений. Показан механизм расширений и его использование для доступа к возможностям ускорителей с помощью языка шейдеров высокого уровня GLSL. Приведено много примеров реализации различных задач, решающихся с помощью расширений OpenGL. Изложенные в книге материалы помогут разработчикам при написании приложений, использующих трехмерную графику: игр, систем визуализации данных, систем проектирования.

На компакт-диске содержатся полные тексты примеров, приведенных в книге, исходные коды авторских библиотек, вспомогательные программы.

Для разработчиков графических приложений, студентов и аспирантов соответствующих специальностей

УДК 681.3.068+800.92
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Рыбинский*
Зав. редакцией	*Григорий Добин*
Редактор	*Леонид Кочин*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.04.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 55,47.
Тираж 3000 экз. Заказ №
"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 5-94157-614-5

Оглавление

Введение (о чем эта книга)

Одной из наиболее быстро развивающихся областей информатики (как у нас принято называть *computer science*) является трехмерная компьютерная графика. Прогресс в этой области легко можно увидеть, сравнивая компьютерные игры за последние годы. Чуть более десяти лет прошло с момента выхода первой версии *Doom'a* (10 декабря 1994 года) до появления *Doom III*, однако графическая наполненность игры и уровень реализма выросли на несколько порядков.

Для программистов, занимающихся трехмерной графикой, особенно важно появление и быстрое развитие различных графических библиотек при существенном росте возможностей видеоадаптеров. Современный видеоадаптер — это фактически мощный ускоритель трехмерной графики (далее мы будем применять термин графический ускоритель или GPU (*Graphics Processing Unit*), обладающий огромными возможностями и производительностью.

Эффективное использование всех ресурсов современных графических ускорителей возможно только с соответствующими интерфейсами API (*Application Program Interface*). Такой API выступает в качестве связующего звена между приложением и графическим ускорителем (рис. В1).

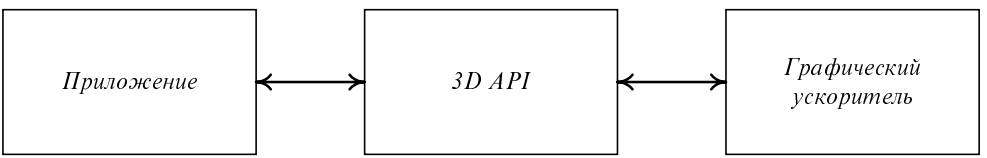


Рис. В1. Упрощенная схема взаимодействия приложения с графическим ускорителем

Одной из важных задач API является предоставление пользователю некоторого интерфейса, абстрагированного от конкретного графического ускорителя. Специфические особенности реального ускорителя обычно инкапсулируются в его драйвере (рис. В2).

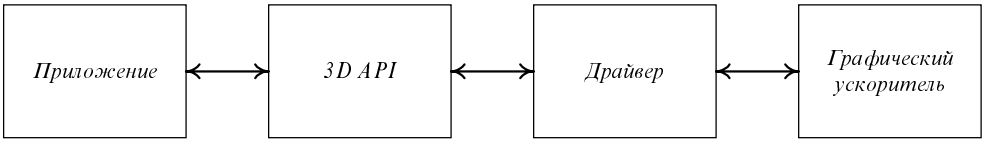


Рис. В2. Схема взаимодействия приложения с графическим ускорителем с учетом драйвера

На данный момент существуют всего два таких API, получившие достаточно широкое распространение для программирования трехмерной графики в реальном времени.

Это OpenGL, уже много лет являющийся стандартом как в области программирования трехмерной графики, так и обучения ей, и Direct3D, разработанный и усиленно проталкиваемый на рынок компанией Microsoft.

Библиотека OpenGL была изначально разработана компанией Silicon Graphics Inc., известной своими разработками в области трехмерной компьютерной графики, на основе собственной библиотеки IRIS GL.

Начиная с 1992 года, разработкой и поддержкой OpenGL занимается корпорация *OpenGL Architecture Review Board* (ARB). В состав OpenGL ARB входят такие компании, как Digital Equipment Corp., Evans & Sutherland, Hewlett-Packard Co., IBM Corp., Intel Corp., Silicon Graphics Inc., ATI, Compaq, nVidia, Intergraph, Sun Microsystems и Microsoft.

С самого начала OpenGL задумывался как открытый процедурный интерфейс к графическому ускорителю (в отличие от Direct3D, первые версии которого были ориентированы исключительно на программный (*software*) рендеринг, что сказалось на его общей архитектуре).

Важнейшими преимуществами (и характеристиками) OpenGL являются его простота, эффективность и независимость от конкретной аппаратной и программной платформы. Приложения на основе OpenGL хорошо работают как на персональных компьютерах под управлением Microsoft Windows, так и на компьютерах Silicon Graphics, Sun, Macintosh и др. Для переноса многих приложений (в частности, большинства примеров) достаточно просто перекомпилировать исходный текст на новой платформе.

При этом, поскольку сама библиотека OpenGL работает только с трехмерной графикой, то для использования ее на конкретной платформе обычно требуются дополнительные библиотеки, занимающиеся организацией взаимодействия с оконной системой, работой с текстурами и т. п.

В отличие от нее Direct3D, входящий в состав DirectX, предоставляет пользователю многие возможности (входящие зачастую не в Direct3D непосредственно, а в другие компоненты DirectX), не имеющие прямого отношения к программированию трехмерной графики, но довольно удобные для про-

граммистов (такие как взаимодействие с оконной системой, работа с текстурами и устройствами ввода).

Для OpenGL подобная функциональность предоставляется обычно целым рядом дополнительных библиотек, заметно облегчающих работу с ней. В этой книге мы будем использовать библиотеки GLU (*OpenGL Utilities*), GLUT (*OpenGL Utilities Toolkit*), а также некоторые библиотеки, разработанные автором. Все они содержатся на прилагаемом к книге компакт-диске. В конце книги вы найдете много полезных ссылок.

Большой ряд полезных возможностей (взаимодействие с оконной системой и устройствами ввода, работа с текстурами и звуком и т. п.) предоставляется библиотекой SDL (*Simple DirectMedia Layer*), также находящейся на компакт-диске, прилагаемом к книге.

Одной из самых удобных черт OpenGL является его спокойное эволюционное развитие, многие статьи и примеры, посвященные программированию с использованием OpenGL, по-прежнему сохраняют свою актуальность и полезность (чего никак нельзя сказать о Direct3D, который претерпевал сильные изменения фактически с каждой новой версией, причем большинство этих изменений значительно обесценивали опыт работы с предыдущей версией. Фактически если вы хотите найти пример реализации какой-то возможности на Direct3D, то вам нужно искать его именно на той версии Direct3D, с которой вы хотите работать в дальнейшем).

Еще одной неудобной чертой библиотеки Direct3D является то, что (как и весь интерфейс DirectX) она построена с использованием пресловутой модели COM. Для лиц, пишущих свои "программы" на Visual Basic, это, конечно, очень удобно. Однако для разработчиков на языке C++ модель COM представляет собой большое неудобство по сравнению с простым процедурным интерфейсом OpenGL.

Однако при таком эволюционном развитии OpenGL позволяет программисту легко и эффективно использовать в своих программах все возможности современных графических ускорителей (например, в игре *Doom III* для работы с графикой служит именно библиотека OpenGL; более того, в ее ресурсных файлах вы можете легко найти используемые вершинные и фрагментные программы).

Книга, которую вы сейчас держите в своих руках, посвящена именно тому, как при работе с OpenGL получить доступ ко всем возможностям современных графических ускорителей.

Основой этого являются так называемые расширения (*extensions*) OpenGL, позволяющие разработчикам графических ускорителей быстро добавлять новые функции, сразу делая их доступными разработчикам программного обеспечения.

Предполагается, что читатель уже знаком с библиотекой OpenGL (если нет, то [4—8] являются достаточно хорошими книгами, которые помогут вам изучить ее) и основой языка C++, поскольку все примеры к книге написаны именно на этом языке.

Дополнительную информацию, новые примеры и статьи вы можете найти на сайте автора по адресу: **www.steps3d.narod.ru**.

Практически все примеры к книге (которые вы можете найти на прилагаемом компакт-диске) компилируются и работают как под управлением операционной системы Microsoft Windows, так и под управлением операционной системы Linux. Для сборки примеров служат так называемые *make*-файлы (файл Makefile отвечает за сборку примеров для Linux, Makefile.nmake — для Windows). В некоторых случаях вам может понадобиться дополнительная настройка, которая выполняется путем изменения путей в *make*-файлах.

Обратите внимание, что в некоторых примерах задействованы новейшие возможности современных графических ускорителей, и это обусловливает довольно высокие требования, предъявляемые к графическому ускорителю и версии драйвера к нему. Поэтому некоторые примеры не будут выполняться на отдельных графических ускорителях.

Для компиляции примеров под Microsoft Windows использовался компилятор Visual C++ 6.0, под Linux — компилятор gcc.

Вся глава 1 книги посвящена расширениям OpenGL и основам работы с ними. В ней подробно разбирается, что такое расширения OpenGL, где можно получить полную документацию по всем существующим на данный момент времени расширениям, какие бывают расширения и каковы общие принципы работы с ними. Здесь также рассматриваются особенности работы с расширениями OpenGL под Windows и Linux, объяснения сопровождаются примерами, показывающими как получить полный список расширений, поддерживаемых вашим графическим ускорителем и драйвером к нему, как проверить поддержку того или иного расширения и начать его использовать. В главе 1 также описана разработанная автором библиотека libExt, предназначенная для кроссплатформенной работы с расширениями OpenGL.

Глава 2 книги посвящена простейшим расширениям OpenGL — мультитекстурированию, которое уже фактически стало частью стандарта OpenGL (и только под Windows, где все вынуждены использовать OpenGL версии 1.1, работа с мультитекстурированием идет через механизм расширений), texture_env_add (задающему еще один закон наложения текстуры), EXT_fog_coord (позволяющему создавать свои законы действия тумана, в частности, слои тумана с заданными характеристиками), EXT_secondary_color (позволяющему задавать для каждой вершины еще один цвет, что будет полезно при работе с другими расширениями) и ARB_texture_border_clamp. На

этих примерах иллюстрируются общие принципы работы с расширениями: проверка поддержки, инициализация, непосредственное использование. Показывается, как использование мультитекстурирования позволяет повысить быстродействие программы за счет комбинации сразу нескольких текстур с различными законами наложения на один проход. Приводится пример создания слоя тумана с помощью расширения EXT_fog_coord.

В главе 3 рассматривается расширение SGIS_generate_mipmap, позволяющее автоматически создавать все необходимые для пирамидального фильтрования (*mipmapping*) уровни текстуры непосредственно в GPU, а также расширения EXT_bgra и EXT_abgr, задающие новые внутренние форматы текстур, EXT_texture_filter_anisotropic, позволяющее в ряде случаев заметно улучшить качество изображения, и ARB_texture_non_power_of_two. Последнее позволяет свободно использовать текстуры, размеры которых не являются степенью двойки.

Глава 4 посвящена расширениям EXT_texture_env_combine и ARB_texture_env_combine, позволяющим задавать сложные законы наложения текстуры на объект. Ряд примеров показывают различные возможности, открываемые с помощью этих расширений.

Глава 5 посвящена новому типу текстур — кубическим текстурным картам (*cubic texture maps*). В ней подробно рассматривается отличие кубических карт от обычных (одно- и двумерных текстур), способы создания, загрузки и использования кубических карт. Рассматривается применение кубических карт для задания произвольной функции направления в трехмерном пространстве и для моделирования отражения окружающей среды поверхностью объекта (*environment mapping*).

Глава 6 вводит еще один тип текстур — трехмерные (3D) текстуры (сейчас они также вошли в стандарт OpenGL, но под Microsoft Windows работа с ними ведется через расширение EXT_texture3D). В этой главе рассматривается расширение EXT_texture3D, вводящее данный тип текстур, способы создания и использования трехмерных текстур.

В главе 7 описываются расширения ARB_point_parameters и ARB_point_sprite. На ряде примеров показывается, как можно с помощью этих расширений эффективно реализовывать различные типы так называемых систем частиц (*particle systems*), служащих для моделирования различных естественных явлений, таких как огонь, взрывы, дымовые следы и т. п.

В главе 8 вводится понятие попиксельного освещения (*per-pixel lighting*). Рассматривается простейшая модель диффузного освещения и приводятся примеры ее реализации через расширение ARB_texture_env_dot3. Здесь также вводится и подробно рассматривается понятие карт нормалей (*bump maps*), способы их задания и построения по картам высот (*height maps*). В этой главе вводится крайне важное для понимания всего попиксельного освещения

понятие касательного пространства (*tangent space*) и показывается его роль при реализации попиксельного диффузного освещения сложных объектов.

Глава 9 вводит понятие *register combiner* и рассматривает расширение NV_register_combiners, появление которого впервые позволило реализовать в реальном времени довольно сложные законы попиксельного освещения. В этой главе подробно рассматриваются организация конвейера рендеринга при использовании *register combiner*, все параметры и законы преобразования данных как в *general combiner*, так и в *final combiner*. Также показывается, как можно реализовать попиксельное диффузное и бликовое (*specular*) освещение, и приводятся многочисленные практические примеры.

Глава 10 рассматривает расширение ARB_vertex_buffer_object и на ряде примеров показывает использование вершинных и индексных буферов для повышения быстродействия программы при работе со сложными сценами.

Глава 11 посвящена *p*-буферам и рендерингу в текстуру. Использование рендеринга в текстуру позволяет легко реализовать ряд сложных визуальных эффектов, включая неровные зеркала, дрожание горячего воздуха, преломление (многочисленные примеры подобных эффектов можно увидеть в игре *Doom III*), постобработку результатов рендеринга. Здесь рассматриваются все необходимые для этого расширения и приводится программная реализация *p*-буферов в виде классов на языке C++. Рассматриваются особенности работы с *p*-буферами для платформ Windows и Linux.

В главе 12 описывается впервые появившееся для графических ускорителей GeForce 3 расширение NV_texture_shader. Вводится понятие шагов текстурирования и элементарных шейдеров. Рассматриваются все вводимые этим расширением шейдеры и примеры их практического использования. Также в этой главе рассматривается такой эффект, как EMBM (*Environment Mapped Bump Mapping*), и приводится его программная реализация с помощью расширения NV_texture_shader. Кроме этого, в главе 12 приводятся и другие примеры работы с расширением NV_texture_shader.

Вся глава 13 посвящена определению видимости в сложных сценах. В ней рассматриваются расширения HP_occlusion_test, NV_occlusion_query и ARB_occlusion_query, позволяющие динамически определять видимость объектов в сложных трехмерных сценах. В этой главе вводятся основные понятия, связанные с видимостью и отсечением объектов по видимости. Приведенные примеры показывают, как правильное использование отсечения (по пирамиде видимости и через расширение ARB_occlusion_query) позволяет заметно ускорить рендеринг сложной сцены.

Глава 14 посвящена сжатию текстур и работе со сжатыми текстурами в приложениях. В ней рассматриваются расширения ARB_texture_compression и EXT_texture_compression_s3tc и их использование. Подробно описывается алгоритм сжатия текстур S3TC и загрузка сжатых текстур из DDS-файлов.

Вся глава 15 посвящена вершинным программам (иногда называемым вершинными шейдерами) и работе с ними через расширение ARB_vertex_program. В этой главе вводится понятие вершинной программы, показывается ее место в конвейере рендеринга, рассматриваются типы регистров, с которыми работает вершинная программа, доступные ей параметры состояния OpenGL. Подробно рассматриваются способ загрузки вершинных программ и используемая система команд. Приводится инкапсуляция вершинной программы в класс языка C++, облегчающая работу с ней. Также даются многочисленные примеры вершинных программ, включая выполнение необходимых преобразований и подготовку вершинных данных для попиксельного освещения.

Глава 16 посвящена еще одному типу программ, поддерживаемых современными графическими ускорителями, — фрагментным программам. Рассматривается расширение ARB_fragment_program, место фрагментной программы в конвейере рендеринга, используемые фрагментной программой регистры, система команд. В главе приводятся многочисленные примеры фрагментных программ для реализации сложных моделей освещения и постобработки результатов рендеринга.

Если главы 15 и 16 были посвящены программам на языках низкого уровня (фактически на языке ассемблера), то в главе 17 рассматривается шейдерный язык высокого уровня — GLSL (*OpenGL Shading Language*). В этой главе излагаются основы языка GLSL, использование шейдеров, написанных на языке GLSL, в OpenGL-программах.

В главе 18 приводится инкапсуляция шейдера на языке GLSL в виде класса языка C++ и рассматривается применение шейдеров на языке GLSL для реализации сложных законов рендеринга и анимации.

В главах 17 и 18 приведено много различных шейдеров, написанных на GLSL, которые могут вам пригодиться.

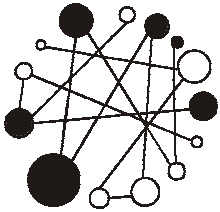
Приложение 1 посвящено основам линейной алгебры. В нем рассматриваются векторы, матрицы и их основные преобразования. Подробно описываются различные системы координат и переход из одной системы в другую. Это приложение поможет вам освежить свои знания в данной области, поскольку эти понятия очень важны при изложении ряда глав.

Приложение 2 включает основные модели освещения: диффузную и бликовую. В нем также рассматривается освещение Гуро и Фонга.

В приложении 3 приведено описание содержимого компакт-диска, прилагаемого к книге.

В списке литературы вы найдете ссылки на книги, изучение которых может оказаться вам весьма полезным, и много ссылок на различные ресурсы в Интернете.

Глава 1



Понятие расширений OpenGL, их основные типы и особенности работы с ними под Windows и Linux

Библиотека OpenGL является фактическим стандартом в мире профессиональной трехмерной графики. Она позволяет легко работать в реальном времени со сложными трехмерными объектами как на различных платформах (Windows, UNIX, Mac OS), так и на различных графических ускорителях (GeForce, Radeon, Matrox и др). Однако, как мы это сейчас наблюдаем, возможности современных графических ускорителей растут очень быстро, причем этот рост носит не только количественный (быстродействие), но и качественный (новые возможности) характер.

К тому же эти новые функции зачастую реализуются совершенно по-разному для различных моделей и производителей графических ускорителей, что создает серьезные проблемы разработчикам программного обеспечения, желающим быстро получить доступ ко всем этим возможностям.

Таким образом возникает ситуация, когда, с одной стороны, хочется, чтобы новые аппаратные возможности как можно быстрее стали доступны разработчикам ПО, а с другой — вносить серьезные изменения в сам OpenGL каждый раз, когда появляется что-то новое, было бы крайне нежелательным.

Компания Microsoft пошла именно путем внесения изменений в свою графическую библиотеку Direct3D. Периодически выпускается новая версия, измененная для поддержки новых возможностей. При этом эти изменения зачастую оказываются довольно радикальными и фактически обнуляют усилия, потраченные на изучение предыдущей версии этой библиотеки.

Иногда бывает, что некоторые возможности так и не вошли в очередную версию Direct3D в связи с какими-то соображениями компании Microsoft.

Разработчики OpenGL пошли другим путем — вместо постоянных серьезных изменений интерфейса библиотеки был введен механизм, позволяющий разработчикам графических ускорителей самим давать разработчикам про-

граммного обеспечения доступ к новым аппаратным возможностям. Этот механизм получил название расширений OpenGL (*OpenGL extensions*).

Фактически каждое расширение — это документированный набор новых функций и констант (близких к стилю, принятому в OpenGL) имеющий свое уникальное имя. За реализацию возможностей, предоставляемых тем или иным расширением, отвечает драйвер ICD (*Installable Client Driver*), который обычно пишется фирмой-разработчиком соответствующего графического ускорителя.

Это позволяет приложению получить полный список имен поддерживаемых расширений и адреса функций, вводимых тем или иным расширением (рис. 1.1).

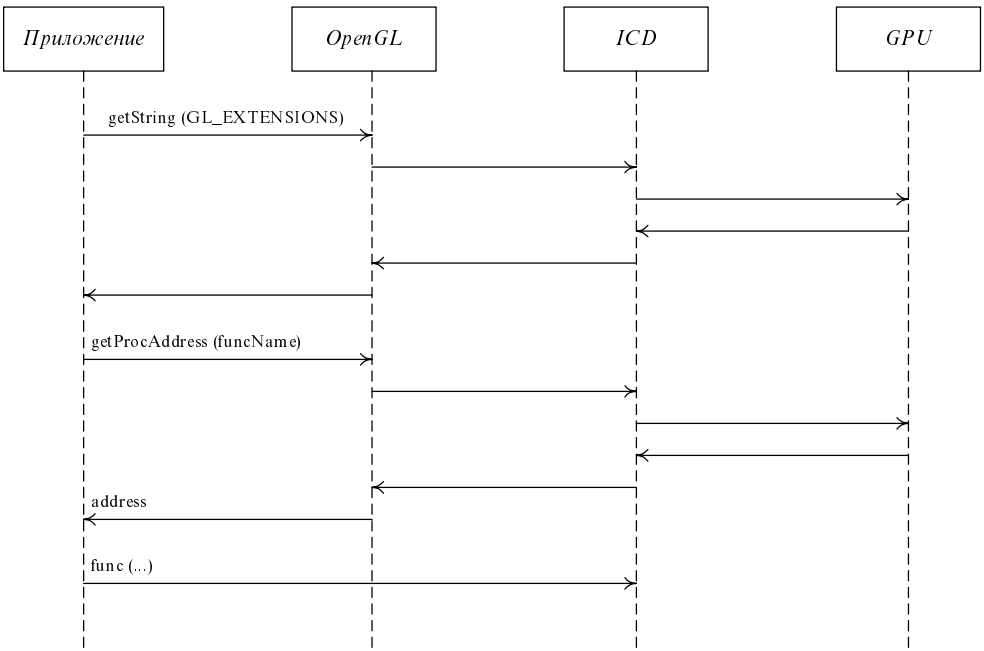


Рис. 1.1. Структура доступа к расширениям

При таком подходе с появлением какой-либо новой возможности достаточно просто документировать ее как новое расширение и добавить поддержку в драйвер. Разработчики ПО могут непосредственно во время выполнения программы проверить, поддерживается ли данное расширение, и в случае поддержки сразу же начать его использовать.

Для этого, если соответствующее расширение поддерживается, то через стандартный механизм получаются адреса новых функций. Способа получе-

ния новых констант непосредственно средствами OpenGL не предусмотрено, вместо этого они должны быть документированы в описании расширения. Таким образом, через описание соответствующего расширения разработчики получают доступ к нему (т. е. к вводимым этим расширением функциям и константам).

Во избежание возможного конфликта между расширениями от различных производителей принято довольно простое правило наименования расширений. Название каждого расширения начинается с префикса (GL, WGL, GLX), определяющего, является ли данное расширение общим (GL) или же оно соответствует какой-то определенной платформе (Windows — WGL, X Window System — GLX). Далее, после символа подчеркивания идет тип расширения, за ним — собственно название, например, GL_EXT_fog_coord. Основные типы расширений OpenGL приводятся в табл. 1.1.

Таблица 1.1. *Основные типы расширений OpenGL*

Тип	**Значение**
ARB	Расширения, введенные OpenGL Architecture Review Board
EXT	Расширения, введенные совместно различными производителями
3DFX	Расширения, введенные компанией 3DFX
APPLE	Расширения, введенные компанией Apple Inc.
ATI	Расширения, введенные компанией ATI Technoligies Inc.
HP	Расширения, введенные компанией Hewlett-Packard Co.
IBM	Расширения, введенные компанией Internation Buisness Machines Inc.
INTEL	Расширения, введенные компанией Intel Corp.
KTX	Расширения, введенные компанией Kinetix
NV	Расширения, введенные компанией NVIDIA
MESA	Расширения, введенные в реализации Mesa
SGI	Расширения, введенные компанией Silicon Graphics Inc.
SGIX	Расширения, введенные компанией Silicon Graphics Inc.
SGIS	Расширения, введенные компанией Silicon Graphics Inc.
SUN	Расширения, введенные компанией Sun Microsystems
WIN	Расширения, введенные компанией Microsoft Corp.

Подобная схема наименования расширений позволяет избежать возможных конфликтов имен между различными производителями.

Поскольку часто оказывалось, что одна и та же функциональность у разных производителей содержалась в различных расширениях, велась определенная работа по унификации расширений. Так, все расширения EXT являются результатом совместной работы различных производителей. Ряд расширений стандартизировались OpenGL Architecture Review Board и имеют тип ARB. Существует постоянно пополняемый список всех расширений и документации к ним, доступный в Интернете по адресу **http://oss.sgi.com/projects/ogl-sample/registry**.

Кроме списка расширений и их описаний поддерживаются также списки заголовочных файлов, содержащих описания расширений. Список общих (GL) расширений содержится в файле glext.h, полный список всех расширений для платформы Windows — в файле wglext.h, список расширений для X Window System — glxext.h. Все эти файлы вы можете найти на прилагаемом к книге компакт-диске, а самые последние версии — в Интернете по адресу: **http://oss.sgi.com/projects/ogl-sample/registry/**.

Для того чтобы во время выполнения программы получить список всех доступных расширений, используется функция `glGetString` с параметром `GL_EXTENSIONS`. Она возвращает список всех доступных программе расширений в виде строки имен, разделенных пробелами. Далее приводится пример подобной строки для графического ускорителя GeForce 2 MX.

```
GL_ARB_imaging GL_ARB_multitexture GL_ARB_point_parameters
GL_ARB_point_sprite GL_ARB_shader_objects GL_ARB_shading_language_100
GL_ARB_texture_compression GL_ARB_texture_cube_map GL_ARB_texture_env_add
GL_ARB_texture_env_combine GL_ARB_texture_env_dot3
GL_ARB_texture_mirrored_repeat GL_ARB_transpose_matrix
GL_ARB_vertex_buffer_object GL_ARB_vertex_program GL_ARB_vertex_shader
GL_ARB_window_pos GL_S3_s3tc GL_EXT_texture_env_add GL_EXT_abgr
GL_EXT_bgra GL_EXT_blend_color GL_EXT_blend_minmax GL_EXT_blend_subtract
GL_EXT_clip_volume_hint GL_EXT_compiled_vertex_array GL_EXT_Cg_shader
GL_EXT_draw_range_elements GL_EXT_fog_coord GL_EXT_multi_draw_arrays
GL_EXT_packed_pixels GL_EXT_paletted_texture GL_EXT_pixel_buffer_object
GL_EXT_point_parameters GL_EXT_rescale_normal GL_EXT_secondary_color
GL_EXT_separate_specular_color GL_EXT_shared_texture_palette
GL_EXT_stencil_wrap GL_EXT_texture_compression_s3tc
GL_EXT_texture_cube_map GL_EXT_texture_edge_clamp
GL_EXT_texture_env_combine GL_EXT_texture_env_dot3
GL_EXT_texture_filter_anisotropic GL_EXT_texture_lod
GL_EXT_texture_lod_bias GL_EXT_texture_object GL_EXT_vertex_array
GL_IBM_rasterpos_clip GL_IBM_texture_mirrored_repeat GL_KTX_buffer_region
GL_NV_blend_square GL_NV_fence GL_NV_fog_distance
GL_NV_light_max_exponent GL_NV_packed_depth_stencil
GL_NV_pixel_data_range GL_NV_point_sprite GL_NV_register_combiners
GL_NV_texgen_reflection GL_NV_texture_env_combine4
GL_NV_texture_rectangle GL_NV_vertex_array_range
GL_NV_vertex_array_range2 GL_NV_vertex_program GL_NV_vertex_program1_1
GL_SGIS_generate_mipmap GL_SGIS_multitexture GL_SGIS_texture_lod
GL_SUN_slice_accum GL_WIN_swap_hint WGL_EXT_swap_control
```

Обратите внимание на внушительный размер списка расширений для этого довольно старого графического ускорителя. Аналогичный список для ускорителя серии GeForceFX занял бы несколько страниц.

Заметьте также, что для получения этой строки необходимо сперва проинициализировать OpenGL, иначе функция `glGetString` возвращает значение `NULL`.

Листинг 1.1 содержит простейшую программу, печатающую полный список всех доступных расширений. При этом для простоты инициализации OpenGL используется библиотека GLUT, хотя никакого рисования здесь не производится.

Листинг 1.1. Печать списка всех поддерживаемых GL-расширений

```
#ifdef _WIN32
     #include     <windows.h>
#endif
#include     <GL/gl.h>
#include     "glut.h"
#include     "../glext.h"
#include     <stdio.h>
#include     <ctype.h>
void     printExtList ( const char * extension )
{
     char     name [1024];
     int      i, j;
     printf ( "Supported extensions:\n" );
     for ( i = 0, j = 0; extension [i] != '\0'; i++ )
          if ( !isspace ( extension [i] ) )
               name [j++] = extension [i];
          else
          {
               name [j] = '\0';e
               printf ( "\t%s\n", name );
               j = 0;
          }
     if ( j > 0 )
     {
          name [j] = '\0';
```

```
          printf ( "\t%s\n", name );
     }
}
int main ( int argc, char * argv [] )
{
     glutInit            ( &argc, argv );
     glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB |
                           GLUT_DEPTH );
     glutInitWindowSize  ( 400, 400 );
     int     win = glutCreateWindow ( "OpenGL example 1" );
     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 );
     printExtList ( extension );
     return 0;
}
```

Теперь для проверки того, поддерживается ли данное расширение или нет, достаточно просто получить строку со списком расширений и посмотреть, содержится ли в ней имя интересующего расширения.

Обратите внимание, что простой проверки только с помощью функции `strstr` недостаточно, поскольку надо убедиться, что было найдено имя именно данного расширения, а не начало названия какого-либо другого расширения. Для этого достаточно проверить возвращенное функцией `strstr` значение на наличие пробела или `\0` в том месте, где должно заканчиваться название интересующего нас расширения. Далее приводится исходный текст функции, выполняющей такую проверку.

```
bool     isExtensionSupported ( const char * ext )
{
    const char * extensions = (const char *)
                     glGetString ( GL_EXTENSIONS );
```

```
    const char * start      = extensions;
    const char * ptr;
    while ( ( ptr = strstr ( start, ext ) ) != NULL )
    {
                                        // we've found, ensure
                                      // name is exactly ext
        const char * end = ptr + strlen ( ext );
        if ( isspace ( *end ) || *end == '\0' )
            return true;
        start = end;
    }
    return false;
}
```

Вместо `isExtensionSupported` при работе с библиотеками GLUT можно воспользоваться функцией

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

Она возвращает ненулевое значение в случае, когда расширение с заданным именем поддерживается.

Обратите, однако, внимание, что таким образом проверяется поддержка только расширений типа GL, для проверки расширений WGL и GLX следует использовать другой способ, который будет рассмотрен несколько позже.

Следующим шагом, после проверки того, поддерживается ли данное расширение вашим графическим ускорителем и драйвером к нему (на самом деле поддержка расширений зависит не только от самого графического ускорителя, но и от установленного драйвера, поэтому всегда пользуйтесь последней версией драйвера для своего графического ускорителя), является получение нужных для работы адресов вводимых данным расширением функций.

Для каждой вводимой расширением функции в соответствующем заголовочном файле (glext.h, wglext.h, glxext.h) содержится определение типа указателя на данную функцию.

Так, для функции `glFogCoordEXT`, вводимой расширением GL_EXT_fog_coord, определяется следующий тип:

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

Таким образом тип `PFNGLFOGCOORDFEXTPROC` является адресом указателя на функцию `glFogCoordEXT`.

Поэтому если ввести переменную `glFogCoordEXT`, определив ее следующим образом

```
PFNGLFOGCOORDFEXTPROC            glFogCoordfEXT ;
```

и потом присвоить ей значение адреса этой функции, то обращение

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

будет корректным обращением к функции `glFogCoordfEXT`, вводимой данным расширением.

Таким образом, для получения из программы доступа к функциям, вводимым тем или иным расширением, достаточно ввести набор переменных — указателей на эти функции и проинициализировать их соответствующими адресами. После этого можно обращаться к ним как к обычным функциям.

Рассмотрим теперь, как по имени функции получить ее адрес.

К сожалению, это сильно зависит от используемой платформы. Поэтому мы сначала рассмотрим, как это делается для платформы Microsoft Windows, а потом перейдем к платформе Linux.

Для того чтобы под Windows получить адрес определенной, вводимой каким-либо расширением, функции служит функция `wglGetProcAddress`.

```
PROC wglGetProcAddress(LPCSTR lpszProc);
```

Она по имени требуемой функции возвращает указатель на нее.

Обратите внимание, что для корректного присвоения ее соответствующему указателю следует выполнить явное приведение типа возвращаемого этой функцией адреса.

С помощью функции `wglGetProcAddress` можно получать адреса не только GL-расширений, но и расширений платформы Windows — WGL-расширений.

Приводимый далее фрагмент кода, иллюстрирует (на примере функции `glFogCoordfEXT`) правильную работу с расширением (проверка поддержки, инициализация и использование).

```
#include          <glext.h>
PFNGLFOGCOORDFEXTPROC     glFogCoordfEXT = NULL;

if ( isExtensionSupported ( "GL_EXT_fog_coord" ) )
     glFogCoordfEXT = (PFNGLFOGCOORDFEXTPROC)
               wglGetProcAddress ( "glFogCoordfEXT" );
...
glFogCoordfEXT ( 0.5f );
...
```

Важной особенностью платформы Windows является то, что полученный адрес в общем случае пригоден только для конкретного контекста рендеринга, в котором он был получен, и в другом контексте может не работать.

Рассмотрим теперь, каким образом осуществляется работа с расширением для платформы Linux.

Работа с библиотекой OpenGL под управлением операционной системы Linux определяется OpenGL ABI (*OpenGL Application Binary Interface for Linux* (OpenGL ABI)). Этот документ можно найти в Интернете по адресу **http://oss.sgi.com/projects/ogl-sample/ABI/index.html**.

Поскольку в русскоязычной литературе информация по работе с OpenGL под Linux практически отсутствует, далее рассматриваются несколько важных моментов использования OpenGL под Linux.

Для работы с OpenGL под Linux вам понадобятся библиотеки libGL и libGLU, расположенные обычно в каталоге /usr/lib.

Если вы хотите использовать библиотеку GLUT, то вам придется собрать ее из исходных файлов (или скачать ее вариант в виде RPM-файла).

Необходимы следующие заголовочные файлы — GL/gl.h, GL/glx.h, GL/glu.h и GL/glext.h.

На данный момент уже существует версия 1.2 OpenGL, поставляемая с Linux (в отличие от Windows, где все до сих пор работают с версией 1.1).

Проверка поддержки расширения осуществляется под Linux точно так же, как и под Windows, для этого пригодна уже известная вам функция `isExtensionSupported`.

Функция для получения адреса вводимых расширением функций под Linux выглядит несколько иначе:

```
void * glXGetProcAddressARB ( const GLubyte * name );
```

Обратите внимание, что аргументом этой функции является не строка (`const char *`), а цепочка байтов, хотя и содержащая имя интересующей нас функции в виде ASCIIZ-строки. Поэтому необходимо приведение типа из строки в цепочку байтов.

В отличие от Microsoft Windows адреса, возвращаемые данной функцией, доступны в любом контексте рендеринга OpenGL. Эта функция может использоваться как для получения GL, так и GLX-расширений.

Вот вариант рассмотренной нами функции `glForCoordfEXT` для платформы Linux:

```
#include          <glext.h>
PFNGLFOGCOORDFEXTPROC     glFogCoordfEXT = NULL;
if ( isExtensionSupported ( "GL_EXT_fog_coord" ) )
```

```
    glFogCoordfEXT = (PFNGLFOGCOORDFEXTPROC)
      glXGetProcAddress ((const Glubyte *)"glFogCoordfEXT");
...
glFogCoordfEXT ( 0.5f );
...
```

Работа с WGL-расширениями

Поскольку эти расширения специфичны только для Windows, то проверка их поддержки непосредственно введенной ранее функцией `isExtensionSupported` невозможна, поскольку используемая там строка не содержит расширений платформы Windows.

Поэтому нужно получить строку со всеми поддерживаемыми расширениями, специфичными для Windows. Для этого сначала следует стандартным образом проверить поддержку расширения WGL_ARB_extensions, и в случае его поддержки получить указатель на функцию `wglGetExtensionsStringARB`.

```
if ( isExtensionSupported ( "WGL_ARB_extensions" ) )
    wglGetExtensionsStringARB = (PFNWGLGETEXTENSIONSSTRINGARBPROC)
                          getProcAddress ( "wglGetExtensionsStringARB" );
```

В результате мы получаем указатель на функцию, возвращающую список всех WGL-расширений для заданного контекста устройства (*device context*) в том же формате, в котором возвращается список всех платформонезависимых расширений:

```
const char * exts = wglGetExtensionsStringARB ( wglGetCurrentDC () );
```

Тогда для проверки какого-либо из WGL-расширений достаточно установить, встречается ли название расширения в полученной строке. Если да, то можно с помощью функции `wglGetProcAddress` получить адреса всех вводимых этим расширением функций.

Работа с GLX-расширениями

Под Linux также существует функция, возвращающая список всех поддерживаемых GLX-расширений. Однако она не является расширением и непосредственно доступна пользователям. Вот пример ее описания:

```
const char * glXQueryExtensionsString ( Display * dpy, int screen );
```

После того как при помощи этой функции был получен список всех GLX-расширений, достаточно рассмотренным ранее способом проверить наличие в этой строке названия расширения. В случае если соответствующее расши-

рение было найдено, то при помощи функции `glXGetProcAddress` можно получить адреса вводимых этим расширением функций.

Слегка модифицированный код функции `isExtensionSupported`, обеспечивающей проверку поддержки всех расширений, приводится в листинге 1.2.

Листинг 1.2. Модифицированный вариант проверки поддержки расширений

```
static     bool     isExtensionSupported ( const char * ext,
                                     const char * extList )
{
     const char * start = extList;
     const char * ptr;

     while ( ( ptr = strstr ( start, ext ) ) != NULL )
     {
                         // we've found, ensure name
                                  // is exactly ext
          const char * end = ptr + strlen ( ext );
          if ( isspace ( *end ) || *end == '\0' )
               return true;
          start = end;
     }
     return false;
}
bool     isExtensionSupported ( const char * ext )
{
    const char * extensions = (const char *) glGetString (
                                                  GL_EXTENSIONS );
    if ( isExtensionSupported ( ext, extensions ) )
     return true;
#ifdef     _WIN32                    // check Windoze extensions
    if ( wglGetExtensionsStringARB == NULL )
    return false;
    return isExtensionSupported ( ext,
             wglGetExtensionsStringARB ( wglGetCurrentDC () ) );
#else                          // check GLX extensions
    Display * display = glXGetCurrentDisplay ();
    int           screen  = DefaultScreen        ( display );
```

```
    return isExtensionSupported ( ext,
                        glXQueryExtensionsString ( display, screen ) );
#endif
}
```

Для удобства дальнейшей работы с расширениями все необходимые операции можно записать в небольшую библиотеку, причем желательно инкапсулировать всю платформозависимую часть.

Для формирования кросс-платформенного кода изменяющуюся часть (получение адреса функции) лучше вынести в отдельную функцию:

```
static void  * getProcAddress ( const char * name )
{
#ifdef     _WIN32
    return wglGetProcAddress ( name );
#else
    return (void *)glXGetProcAddressARB (
               (const GLubyte *)name );
#endif
}
```

Обратите внимание на использование условной компиляции — символ `_WIN32` служит для определения того, происходит ли компиляция под платформой Windows или нет (здесь мы будем считать, что если этот символ не определен, то компиляция происходит под Linux).

На прилагаемом к книге компакт-диске содержится написанная автором библиотека для работы с расширениями libExt. Она не претендует на полноту и поэтому поддерживает лишь часть из имеющихся расширений. Однако все рассматриваемые в этой книге расширения ею поддерживаются, и она является кроссплатформенной (успешно компилируется и под Windows, и под Linux, и способ работы с ней не зависит от платформы).

В листинге 1.3 приводится фрагмент заголовочного файла libExt.h этой библиотеки, содержащий описание вводимых этим расширением функций (функции, вводимые различными расширениями, здесь намеренно не приводятся из-за их большого количества).

Листинг 1.3. Файл libExt.h

```
#ifdef     _WIN32
    #include     <windows.h>
```

```
#else
    #define     GLX_GLXEXT_LEGACY
#endif
#include     <GL/gl.h>
#include     <GL/glu.h>
#include     "../glext.h"
#ifdef     _WIN32
     #include     "../wglext.h"
#else
    #include     <GL/glx.h>
    #include     <GL/glxext.h>
 #endif
bool  isExtensionSupported  ( const char * ext );
void  initExtensions        ();
void  printfInfo            ();
void  assertExtensionsSupported  ( const char * extList );
```

Для удобства работы этот файл включает необходимые заголовочные файлы и для платформы Microsoft Windows файл windows.h, без которого многие примеры под Windows просто не компилируются. Данный файл определяет также следующие функции — `isExtensionSupported`, `assertExtensionsSupported`, `initExtensions` и `printfInfo`.

Функция `isExtensionSupported` уже рассматривалась нами, она просто проверяет, поддерживается ли данное расширение (при этом проверяется поддержка как общих расширений, так и расширений, специфичных для конкретной платформы).

Функция `assertExtensionsSupported` позволяет проверить поддержку сразу нескольких расширений. На вход ей передается строка с именами расширений (разделенных пробелами или запятыми). Если хотя бы одно расширение из переданного списка не поддерживается, то выдается диагностическое сообщение с именем неподдерживаемого расширения, и выполнение программы прерывается.

Функция `initExtensions` служит для инициализации указателей на вводимые расширениями функции. Библиотека libExt определяет большое число указателей на функции, вводимые всеми рассматриваемыми в этой книге расширениями, и с помощью этой функции в указатели заносятся адреса вводимых расширениями функций.

Функция `printfInfo` печатает на `stdout` информацию об используемом графическом ускорителе и его производителе, а также о поддерживаемых расширениях.

Обратите внимание, что библиотека GLAUX, которую многие используют для загрузки текстур из BMP-файлов, под Linux обычно недоступна.

Поэтому на прилагаемом к книге компакт-диске вы найдете написанную автором небольшую кроссплатформенную библиотеку libTexture, позволяющую загружать текстуры из BMP-, TGA-, DDS-, JPG- и PNG-файлов. Обе эти библиотеки будут активно использоваться на протяжении всей книги.

Для сборки примеров мы будем активно использовать *make*-файлы. Листинги 1.4 и 1.5 иллюстрируют простейшие примеры для Windows и Linux, обеспечивающие сборку приложений с поддержкой библиотек libExt и libTexture.

Листинг 1.4. Пример nmake-файла для сборки примеров под Windows

```
#
# Makefile to build examples for texture_env_combine
#
!include <win32.mak>
!include <..\nmake.inc>
EXES = env-combine-1.exe env-combine-2.exe env-combine-3.exe
env-combine-4.exe env-combine-5.exe
all: $(EXES)
env-combine-1.exe: env-combine-1.obj Torus.obj version.res $(OBJS)
     $(link) env-combine-1.obj Torus.obj $(OBJS) version.res $(LLDLIBS)
env-combine-2.exe: env-combine-2.obj Torus.obj version.res $(OBJS)
     $(link) env-combine-2.obj Torus.obj $(OBJS) version.res $(LLDLIBS)
env-combine-3.exe: env-combine-3.obj Torus.obj version.res $(OBJS)
     $(link) env-combine-3.obj Torus.obj $(OBJS) version.res $(LLDLIBS)
env-combine-4.exe: env-combine-4.obj Torus.obj version.res $(OBJS)
     $(link) env-combine-4.obj Torus.obj $(OBJS) version.res $(LLDLIBS)
env-combine-5.exe: env-combine-5.obj Torus.obj version.res $(OBJS)
     $(link) env-combine-5.obj Torus.obj $(OBJS) version.res $(LLDLIBS)
clean:
     @del env-combine-1.obj env-combine-2.obj env-combine-3.obj
env-combine-4.obj env-combine-5.obj Torus.obj version.res $(EXES) $(OBJS)
> nul
!include <..\rules.win32>
```

Листинг 1.5. Пример *make*-файла для сборки примеров под Linux

```
#
# Makefile to build examples for chapter 4 for Linux
#
include ../make.inc
ALL = env-combine-1 env-combine-2 env-combine-3 env-combine-4
env-combine-5
all: $(ALL)
env-combine-1: env-combine-1.o Torus.o $(OBJS)
     cc $(LDFLAGS) env-combine-1.o Torus.o $(OBJS) -o env-combine-1
$(LIBS) -lc
env-combine-2: env-combine-2.o Torus.o $(OBJS)
     cc $(LDFLAGS) env-combine-2.o Torus.o $(OBJS) -o env-combine-2
$(LIBS) -lc
env-combine-3: env-combine-3.o Torus.o $(OBJS)
     cc $(LDFLAGS) env-combine-3.o Torus.o $(OBJS) -o env-combine-3
$(LIBS) -lc
env-combine-4: env-combine-4.o Torus.o $(OBJS)
     cc $(LDFLAGS) env-combine-4.o Torus.o $(OBJS) -o env-combine-4
$(LIBS) -lc
env-combine-5: env-combine-5.o Torus.o $(OBJS)
     cc $(LDFLAGS) env-combine-5.o Torus.o $(OBJS) -o env-combine-5
$(LIBS) -lc
clean:
     rm -r -f $(ALL) $(OBJS) env-combine-1.o env-combine-2.o
env-combine-3.o env-combine-4.o env-combine-5.o Torus.o 2> /dev/nul
include .../rules.linux
```

В файлах make.inc и nmake.inc находятся определения основных параметров компиляции, каталогов, библиотек и т. п. Пример такого файла для платформы Windows содержит листинг 1.6.

Листинг 1.6. Файл nmake.inc

```
#
# Include file with basic definitions for all examples for Win32
#
OPENGL         = glut32.lib glu32.lib opengl32.lib
GLUTPATH       = ../glut
LIBEXTPATH     = ../libExt/
```

```
LIBTEXTUREPATH = ../libTexture/
LIB3DPATH      = ../3D/
UTILSPATH      = ../utils
PBUFFERPATH    = ../PBuffer/Win32
PROGRAMPATH    = ../Program
PBUFFERINC      = -I../PBuffer -I../PBuffer/Win32
INC             = $(INC) -I$(GLUTPATH) -I$(LIBEXTPATH) -
I$(LIBTEXTUREPATH) -I$(UTILSPATH) -I$(LIB3DPATH) -I$(PROGRAMPATH) -
I$(LIBTEXTUREPATH)zlib -I$(LIBTEXTUREPATH)libpng -I$(LIBTEXTUREPATH)jpeg
LLDLIBS         = $(LFLAGS) $(OPENGL) $(guilibs)
$(LIBTEXTUREPATH)jpeg/libjpeg.lib $(LIBTEXTUREPATH)libpng/libpng.lib
$(LIBTEXTUREPATH)zlib/zlib.lib /LIBPATH:$(GLUTPATH)
LIBEXT_OBJS     = libExt.obj
LIBTEXTURE_OBJS = libTexture.obj Texture.obj CompressedTexture.obj
BmpLoader.obj TgaLoader.obj DdsLoader.obj JpegLoader.obj PngLoader.obj
Data.obj ZipFileSystem.obj
OBJS            = $(LIBEXT_OBJS) $(LIBTEXTURE_OBJS) Vector3D.obj
                  Vector2D.obj
CFLAGS          = $(CFLAGS) /GX
```

Файлы rules.win32 и rules.linux задают правила компиляции исходных файлов. В листинге 1.7 приводится пример такого файла для платформы Windows.

Листинг 1.7. Файл rules.win32

```
#
# Rules to compile programs under Win32
#
.cpp.obj:
     $(CC) $(CFLAGS) $(INC) /c $<
{$(LIBEXTPATH)}.cpp.obj::
     $(CC) $(CFLAGS) $(INC) /c $<
{$(LIBTEXTUREPATH)}.cpp.obj::
     $(CC) $(CFLAGS) $(INC) /c $<
{$(LIB3DPATH)}.cpp.obj::
     $(CC) $(CFLAGS) $(INC) /c $<
{$(UTILSPATH)}.cpp.obj::
     $(CC) $(CFLAGS) $(INC) /c $<
{$(PBUFFERPATH)}.cpp.obj::
     $(CC) $(CFLAGS) $(INC) /c $<
```

```
{$(PROGRAMPATH)}.cpp.obj::
     $(CC) $(CFLAGS) $(INC) /c $<
.c.obj:
     $(CC) $(CFLAGS) $(INC) /c $<
version.res : version.rc
     $(RC) $(RCFLAGS) /r /fo"version.res" version.rc
```

На компакт-диске, прилагаемом к книге, в каталоге с исходным кодом для *главы 1* вы найдете текст приложения, печатающего полный список поддерживаемых расширений или же проверяющего поддержку заданного (в командной строке) расширения. Листинг 1.8 иллюстрирует пример использования этой утилиты.

Листинг 1.8. Утилита для проверки поддержки расширений

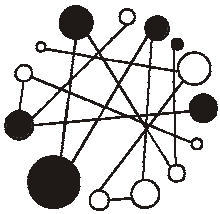
```
//
// Sample to print info about OpenGL card and driver
// Lists all supported extensions
//
#ifdef     _WIN32
     #include     <windows.h>
#endif
#include     <GL/gl.h>
#include     "glut.h"
#include     "../glext.h"
#include     <stdio.h>
#include     <stdlib.h>
#include     <string.h>
#include     <ctype.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 );
}
void     printExtList ( const char * extension )
{
     char     name [1024];
     int     i, j;
     printf ( "Supported extensions:\n" );
     for ( i = 0, j = 0; extension [i] != '\0'; i++ )
          if ( !isspace ( extension [i] ) )
               name [j++] = extension [i];
          else                                 // end of name
          {
               name [j] = '\0';                // terminate name
               printf ( "\t%s\n", name );
               j = 0;
          }
     if ( j > 0 )
     {
          name [j] = '\0';
          printf ( "\t%s\n", name );
     }
}
bool     isExtensionSupported ( const char * ext )
{
     const char * extensions = (const char *)
                                       glGetString ( GL_EXTENSIONS );
     const char * start      = extensions;
     const char * ptr;
     while ( ( ptr = strstr ( start, ext ) ) != NULL )
     {
```

```
                              // we've found, ensure name
                                        // is exactly ext
          const char * end = ptr + strlen ( ext );
          if ( isspace ( *end ) || *end == '\0' )
               return true;
          start = end;
     }
     return false;
}
int main ( int argc, char* argv [] )
{
                              // initialize glut
     glutInit            ( &argc, argv );
     glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
     glutInitWindowSize  ( 400, 400 );
                              // create window
     int     win = glutCreateWindow ( "OpenGL example 1" );
                              // register handlers
     glutDisplayFunc  ( display );
     glutReshapeFunc  ( reshape );
       glutKeyboardFunc ( key );
     init ();
     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 ( argc < 2 )                    // print a list of
                                                // extensions
          printExtList ( extension );
     else                               // check extensions
          for ( int i = 1; i < argc; i++ )
               if ( isExtensionSupported ( argv [i] ) )
                    printf("%-40s - supported\n", argv [i]);
               else
                    printf("%-40s - NOT supported\n", argv [i]);
     return 0;
}
```

Библиотека libTexture будет подробно рассмотрена в *главе 2*.

Глава 2



Простейшие расширения

Мультитекстурирование, расширение ARB_multitexture

Для получения ряда визуальных эффектов (карты освещенности, туман, микрофактурные текстуры и т. п.) часто возникает необходимость наложения на грань не одной, а сразу нескольких текстур, причем зачастую с разными законами наложения (рис. 2.1).

К сожалению, стандартный OpenGL позволяет накладывать только одну текстуру за раз (проход), что приводит к необходимости реализовывать для вывода граней несколько проходов. При этом временные затраты на рендеринг объектов возрастают в соответствующее число раз.

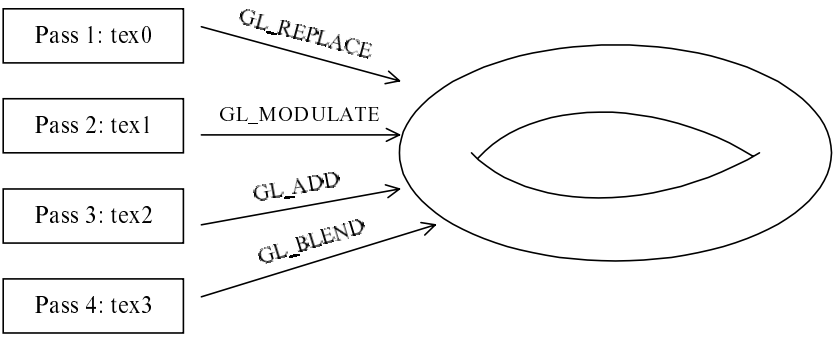


Рис. 2.1. Наложение нескольких текстур

Поэтому естественным является желание добавить возможность за один проход накладывать сразу несколько текстур. И не случайно, что первым официальным ARB-расширением, принятым в сентябре 1998 года, является именно ARB_multitexture.

Данное расширение позволяет за один проход накладывать сразу несколько текстур, при этом для каждой выводимой текстуры можно задать свои пара-

метры наложения, набор текстурных координат, матрицу преобразования текстурных координат и т. п. Общая схема мультитекстурирования представлена на рис. 2.2.

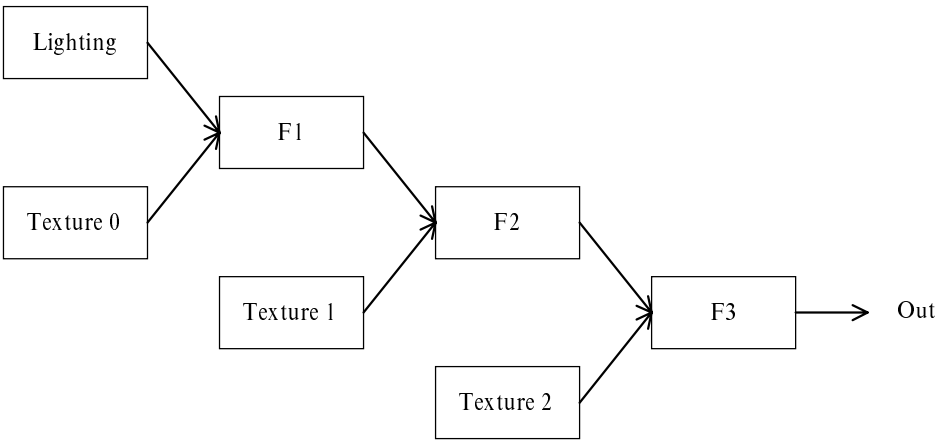


Рис. 2.2. Общая схема мультитекстурирования в OpenGL

Сейчас поддержка мультитекстурирования входит в стандарт OpenGL (начиная с версии 1.2.1), и только под Windows (так как Microsoft по-прежнему поддерживает лишь OpenGL версии 1.1) приходится работать с мультитекстурированием через соответствующее расширение.

Для проверки поддержки данного расширения можно использовать функцию `isExtensionSupported`, рассмотренную в *главе 1*.

Если

```
isExtensionSupported ( "GL_ARB_multitexture" )
```

возвращает значение `true`, то данное расширение поддерживается. Различные графические ускорители поддерживают разное число текстурных блоков (*texture units*, различных текстур, которые можно наложить за один проход). Их число для данного ускорителя (и драйвера) можно определить при помощи функции `glGetIntegerv`:

```
int     maxTextureUnits;
glGetIntegerv ( GL_MAX_TEXTURE_UNITS_ARB, &maxTextureUnits );
```

Максимальное число текстурных блоков, поддерживаемых расширением GL_ARB_multitexture, равно 32. Данное расширение вводит ряд функций и констант. Прототипы основных функций приводятся далее.

```
void glMultiTexCoord{1234}{sifd}     ( GLenum texture,T coords )
void glMultiTexCoord{1234}{sifd}v ( GLenum texture,T coords )
```

```
void glClientActiveTexture( GLenum texture );
void glActiveTexture( GLenum texture );
```

Для работы с этим расширением мы будем использовать библиотеку libExt, введенную в предыдущей главе. Для инициализации указателей на вводимые функции следует перед началом работы с этим (как и с любым другим) расширением вызвать функцию `initExtensions ()`.

После того как указатели на функции получены, можно (естественно после загрузки текстур) задать используемые текстуры.

Сначала следует при помощи функции `glActiveTextureARB` задать активный текстурный модуль:

```
glActiveTextureARB ( texture );
```

Параметр этой функции задает номер текстурного модуля и может принимать одно из следующих значений: `GL_TEXTURE0_ARB`, `GL_TEXTURE1_ARB`, `GL_TEXTURE2_ARB`, ..., `GL_TEXTURE31_ARB`.

По умолчанию (т. е. до первого вызова `glActiveTextureARB`) активным является нулевой текстурный блок (`GL_TEXTURE0_ARB`).

После задания текстурного модуля следует разрешить использование текстуры (например, при помощи команды `glEnable ( GL_TEXTURE_2D )`), задать конкретную текстуру и ее параметры (при помощи функций `glBindTexture` и `glTexParameter`), а также указать способ применения этой текстуры. В листинге 2.1 приводится простой пример задания двух текстурных модулей.

Листинг 2.1. Задание двух текстур с использованием мультитекстурирования

```
glBindTexture   ( GL_TEXTURE_2D, texture1 );
glTexParameterf ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
glTexParameterf ( GL_TEXTURE_2D, GL_TEXTURE_MAX_FILTER, GL_LINEAR  );
glTexImage2D    ( GL_TEXTURE_2D, GL_RGBA, width1, height1, 0, format1,
                  GL_UNSIGNED_BYTE, pixels1 );
glBindTexture   ( GL_TEXTURE_2D, texture2 );
glTexParameterf ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
glTexParameterf ( GL_TEXTURE_2D, GL_TEXTURE_MAX_FILTER, GL_LINEAR  );
glTexImage2D    ( GL_TEXTURE_2D, GL_RGBA, width2, height2, 0, format2,
                  GL_UNSIGNED_BYTE, pixels2 );
glActiveTextureARB ( GL_TEXTURE0_ARB );
glEnable           ( GL_TEXTURE_2D );
glBindTexture      ( GL_TEXTURE_2D, texture1 );
glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE );
glActiveTextureARB ( GL_TEXTURE1_ARB );
```

```
glEnable           ( GL_TEXTURE_2D );
glBindTexture      ( GL_TEXTURE_2D, texture2 );
glMatrixMode       ( GL_TEXTURE );
glScalef           ( 3, 3, 3 );
glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
```

В приведенном листинге первая группа команд определяет и загружает две текстуры, после чего они выбираются в соответствующих текстурных блоках. Обратите внимание, что для разных текстурных блоков задаются разные законы наложения текстуры (в данном примере `GL_REPLACE` и `GL_MODULATE`) и разные текстурные матрицы (в первом случае по умолчанию — единичная матрица, а во втором — единичная матрица, умноженная на три).

При задании вершин выводимого объекта для каждой вершины каждого текстурного блока следует определить свой набор текстурных координат при помощи функции `glMultiTexCoord`:

```
glMultiTexCoord2f  ( texture, u, v );
glMultiTexCoord2fv ( texture, ptr  );
```

Первый параметр задает один из текстурных блоков (т. е. принимает одно из значений `GL_TEXTUREn_ARB`, `n = 0,...,31`). Дальше задаются текстурные координаты либо как набор координат, либо как указатель на массив координат.

Заметьте, что по аналогии с функцией `glTexCoord` существуют варианты функции `glMultiTexCoord` для разных типов входных данных и разного числа текстурных координат (от 1 до 4).

Обратите внимание, что вызов `glTexCoord` задает текстурные координаты для нулевого текстурного блока, т. е. обращение к `glTexCoord` эквивалентно вызову `glMultiTexCoord` с параметром `texture`, равным `GL_TEXTURE0_ARB`.

В следующем примере (листинг 2.2) показывается задание текстурных координат с помощью двух текстурных блоков.

Листинг 2.2. Одновременное задание текстурных координат сразу для двух текстурных блоков

```
glBegin ( GL_TRIANGLES );
  glMultiTexCoord2fv ( GL_TEXTURE0_ARB, &t0 [0] );
  glMultiTexCoord2fv ( GL_TEXTURE1_ARB, &t1 [0] );
  glVertex3fv        ( &v [0]                    );
  glMultiTexCoord2fv ( GL_TEXTURE0_ARB, &t0 [1] );
  glMultiTexCoord2fv ( GL_TEXTURE1_ARB, &t1 [1] );
  glVertex3fv        ( &v [1]                    );
```

```
  glMultiTexCoord2fv ( GL_TEXTURE0_ARB, &t0 [2] );
  glMultiTexCoord2fv ( GL_TEXTURE1_ARB, &t1 [2] );
  glVertex3fv        ( &v [2]                    );
glEnd ();
```

В листинге 2.3 приводится исходный код для примера, выводящего грань с двумя наложенными на нее текстурами.

Листинг 2.3. Программа, выводящая грань с двумя наложенными на нее текстурами

```
//
// Sample to show multitexturing in OpenGL.
// In this example we show product of two textures -
//    texture1 and texture2
//
#include     "libExt.h"
#include     "libTexture.h"
#include     "Vector3D.h"
#include     "Vector2D.h"
#include     <glut.h>
#include     <stdio.h>
#include     <stdlib.h>
unsigned texture1, texture2;
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 );
     glActiveTextureARB ( GL_TEXTURE0_ARB );
     glEnable           ( GL_TEXTURE_2D );
     glBindTexture      ( GL_TEXTURE_2D, texture1 );
     glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                             GL_REPLACE );
     glActiveTextureARB ( GL_TEXTURE1_ARB );
     glEnable           ( GL_TEXTURE_2D );
```

```
    glBindTexture      ( GL_TEXTURE_2D, texture2 );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                         GL_MODULATE );
    glMatrixMode       ( GL_TEXTURE );
    glScalef           ( 3, 3, 3 );
    glBegin ( GL_QUADS );
        glMultiTexCoord2f ( GL_TEXTURE0_ARB, 0, 0 );
        glMultiTexCoord2f ( GL_TEXTURE1_ARB, 0, 0 );
        glVertex3f        ( -1, -1, 0 );
        glMultiTexCoord2f ( GL_TEXTURE0_ARB, 1, 0 );
        glMultiTexCoord2f ( GL_TEXTURE1_ARB, 1, 0 );
        glVertex3f        ( 1, -1, 0 );
        glMultiTexCoord2f ( GL_TEXTURE0_ARB, 1, 1 );
        glMultiTexCoord2f ( GL_TEXTURE1_ARB, 1, 1 );
        glVertex3f        ( 1, 1, 0 );
        glMultiTexCoord2f ( GL_TEXTURE0_ARB, 0, 1 );
        glMultiTexCoord2f ( GL_TEXTURE1_ARB, 0, 1 );
        glVertex3f        ( -1, 1, 0 );
    glEnd   ();
    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
    int     win = glutCreateWindow("OpenGL Multitexture example 2");
                                    // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key );
    initExtensions ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    texture1 = createTexture2D ( true, "../Textures/block.bmp" );
    texture2 = createTexture2D ( true, "../Textures/oak.bmp" );
    init ();
    glutMainLoop ();
    return 0;
}
```

Этот листинг также демонстрирует основы использования библиотек libExt и libTexture. Для инициализации расширений осуществляется вызов функции `initExtensions`, рассмотренной в *главе 1*. Функция `isExtensionSupported` служит для проверки поддержки расширений.

В своих программах всегда, прежде чем начать использовать расширение, проверьте его поддержку. Программа, выдающая сообщение о том, что требуемое расширение не поддерживается, выглядит гораздо лучше, чем программа, "вылетающая" из-за обращения по недопустимому адресу (из-за отсутствия необходимых функций) или просто работающая неправильно (из-за отсутствия параметров или режимов).

Загрузка обычной двумерной текстуры происходит с помощью функции `createTexture2D`, описание которой приводится далее.

```
unsigned    createTexture2D ( bool mipmap, const char * fileName );
```

Эта функция загружает текстуру из файла с именем `fileName`. При этом поддерживаются следующие форматы файлов с текстурами: BMP, TGA, JPG (JPEG), PNG и DDS. Для работы с форматами JPG и PNG предусмотрены библиотеки zlib, libjpeg и libpng. На компакт-диске имеются версии этих библиотек для Windows. При работе под Linux данные библиотеки устанавливаются вместе с Linux, и необходимо только задать их использование для *make*-файла.

Параметр `mipmap` отвечает за поддержку (значение `true`) пирамидального фильтрования при использовании данной текстуры.

Функция возвращает `0` при ошибке загрузки файла. В противном случае возвращается идентификатор текстуры, который может быть использован при вызовах OpenGL.

Для удобства, а также для единообразия при работе с разными платформами во всех вызовах библиотеки libTexture можно использовать символ `/` для разделения каталогов в пути к файлу (вместо принятого в Windows символа `\`). Это избавляет от дублирования символов внутри строк для языков C и C++ (например ..\\Textures\\Bumpmaps\\normal2.bmp).

Еще одной полезной функцией библиотеки libTexture является `saveScreenShot`, позволяющая сохранить текущее содержимое окна в файле типа TGA.

```
bool     saveScreenShot ( const char * fileName );
```

В листинге 2.4 приводится заголовочный файл библиотеки libTexture.

Листинг 2.4. Файл libTexture.h

```
//
// Simple library to load basic textures format
//
#ifndef     __LIB_TEXTURE__
#define     __LIB_TEXTURE__
#include     "TypeDefs.h"
class     FileSystem;
              // create corresponding textures
                                  // from file (or build them)
unsigned     createNormalizationCubemap ( int cubeSize,
                                          bool mipmap = false );
unsigned     createCubeMap   ( bool mipmap, const char * maps [] );
unsigned     createTexture2D ( bool mipmap, const char * fileName );
unsigned     createTexture1D ( bool mipmap, const char * fileName );
                                   // create normalmaps from heightmap
                                  // and from normal map textures
unsigned     createNormalMapFromHeightMap ( bool mipmap,
                                            const char * fileName,
                                            float scale );
unsigned     createNormalMap              ( bool mipmap,
                                            const char * fileName );
```

```
                                    // load texture from file into
                                    // given texture target
bool          loadTexture ( int target, bool mipmap,
                            const char * fileName );
                                    // simple check whether
                                    // the file exists
bool          fileExist ( const char * fileName );
                                    // save contents of the window
                                    // in a TGA file
bool          saveScreenShot ( const char * fileName );
                        // add file system
bool          addFileSystem ( FileSystem * fileSystem );
                        // add zip file source
bool          addZipFileSystem ( const char * fileName );
#endif
```

При помощи функций `addFileSystem` и `addZipFileSystem` можно задавать дополнительные пути для поиска данных (текстур). Так, например, можно задать в качестве параметра к функции `addZipFileSystem` имя ZIP-файла, содержащего текстуры (или часть их). Тогда запрос на чтение текстуры (в случае если обычный файл текстуры не найден) продолжит поиск текстуры уже внутри ZIP-файла.

Возможно задание нескольких таких "альтернативных путей поиска" файлов.

Наложение карт освещенности при помощи мультитекстурирования

Еще одним вариантом мультитекстурирования является вывод грани с картой освещенности. Карты освещенности (*lightmaps*) представляют собой простое и дешевое средство моделирования не изменяющейся со временем освещенности граней. Впервые карты освещенности появились в игре *Quake*.

В карте освещенности цвет точки задается с помощью формулы:

$$I = C_T C_L,$$

где C_T — цвет основной текстуры в соответствующей точке, а C_L — заранее рассчитанная освещенность точки.

При этом освещенность точки C_L также может быть представлена в виде текстуры. Возможность заранее, на этапе разработки сцены, рассчитать ее освещенность позволяет учитывать тени, зависимость освещения от расстояния до источника света и даже вторичную освещенность, когда свет,

падающий на одну грань, рассеивается, и его часть освещает соседние грани (учет вторичной освещенности был выполнен в игре *Quake II*).

Поскольку, в отличие от обычных текстур, карты освещенности уникальны для каждой грани, а освещенность вдоль грани обычно изменяется незначительно, то карты выполняются с меньшим разрешением по сравнению с обычными текстурами. Для растяжения карты освещенности служит механизм линейной интерполяции OpenGL.

Мультитекстурирование позволяет вывести грань вместе с текстурой и картой освещенности за один проход. Настройки OpenGL при этом аналогичны рассмотренному примеру (см. листинг 2.3).

Подробнее о картах освещенности можно прочитать в [1].

Поскольку текстурные координаты для каждого блока задаются независимо, то для него можно задать свою матрицу преобразования текстурных координат, а также способ автоматической генерации текстурных координат и свой закон наложения текстуры (рис. 2.3).

В следующем примере (листинг 2.5) выводится один из стандартных объектов GLUT с наложением как обычной текстуры, так и текстуры, имитирующей отражение (*environment mapping*). Этот пример демонстрирует также использование функции `saveScreenShot`.

Листинг 2.5. Вывод объекта с наложенной текстурой и картой отражения

```
#include     "libExt.h"
#include     "libTexture.h"
#include     "Vector3D.h"
#include     "Vector2D.h"
#include     <glut.h>
#include     <stdio.h>
#include     <stdlib.h>
unsigned texture1, texture2;
float      angle  = 0;
float      angle2 = 0;
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 );
```

```
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, texture1 );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                         GL_REPLACE );
      glActiveTextureARB ( GL_TEXTURE1_ARB );
      glEnable           ( GL_TEXTURE_2D );
      glEnable           ( GL_TEXTURE_GEN_S );
      glEnable           ( GL_TEXTURE_GEN_T );
      glTexGeni          ( GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP );
      glTexGeni          ( GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP );
      glBindTexture      ( GL_TEXTURE_2D, texture2 );
      glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                           GL_MODULATE );
      glPushMatrix    ();
      glTranslatef    ( 2, 2, 2 );
      glRotatef       ( angle,  1, 1, 0 );
      glRotatef       ( angle2, 0, 1, 1 );
      glutSolidTeapot ( 1 );
      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      ( 0.0, 0.0, 0.0,          // eye
                    5.0, 5.0, 5.0,          // center
                    0.0, 1.0, 0.0 );
}
void key ( unsigned char key, int x, int y )
{
   if ( key == 27 || key == 'q' || key == 'Q' )     // quit requested
       exit ( 0 );
```

```
   if ( key == 'p' || key == 'P' )                // saving screenshots
     saveScreenShot ( "multitex3.tga" );
}
void     animate ()
{
     angle  = 0.04f * glutGet ( GLUT_ELAPSED_TIME );
     angle2 = 0.01f * glutGet ( GLUT_ELAPSED_TIME );
     glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                                    // initialize glut
     glutInit            ( &argc, argv );
     glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
     glutInitWindowSize  ( 400, 400 );
                                    // create window
     int     win = glutCreateWindow ( "OpenGL Multitexture example 3" );
                                    // register handlers
     glutDisplayFunc  ( display );
     glutReshapeFunc  ( reshape );
     glutKeyboardFunc ( key );
     glutIdleFunc     ( animate );
     initExtensions ();
     if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
     {
          printf ( "ARB_multitexture NOT supported.\n" );
          return 1;
     }
     texture1 = createTexture2D ( true, "../Textures/block.bmp" );
     texture2 = createTexture2D ( true, "../Textures/stars.bmp" );
     init ();
     glutMainLoop ();
     return 0;
}
```

Обратите внимание, что в этом примере текстурные координаты для нулевого текстурного блока задаются явно (функция `glutSolidTorus` задает текстурные координаты для всех вершин тора), а для первого текстурного бло-

ка — режим автоматического вычисления текстурных координат, имитирующий отражение окружающей среды (*environment mapping*).

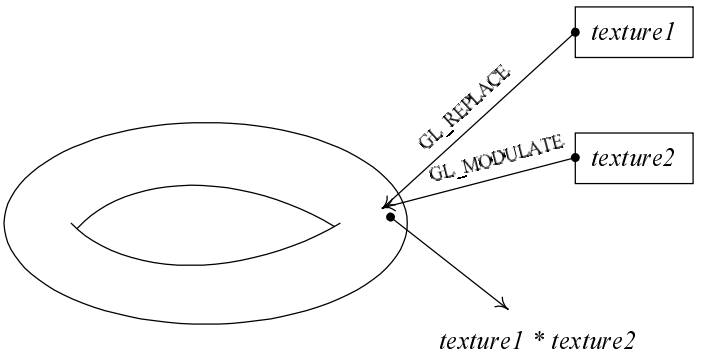


Рис. 2.3. Наложение двух текстур

Возможность мультитекстурирования существует также при работе с вершинными массивами (*vertex array*). Для этого служит функция `glClientActiveTextureARB`. Она берет в качестве входного параметра номер текстурного модуля (`GL_TEXTURE0_ARB`, ..., `GL_TEXTURE31_ARB`) и относит к нему следующие команды: `glTexCoordPointer`, `glEnableClientState` и `glDisableClientState`.

Так, задать массивы текстурных координат для первых двух текстурных модулей можно при помощи такого фрагмента кода:

```
                                // setup 1st texture unit
glClientActiveTextureARB ( GL_TEXTURE0_ARB );
glTexCoordPointer        ( 2, GL_FLOAT, 0, p0 );
glEnableClientState      ( GL_TEXTURE_COORD_ARRAY );
                                // setup 2nd texture unit
glClientActiveTextureARB ( GL_TEXTURE1_ARB );
glTexCoordPointer        ( 2, GL_FLOAT, 0, p1 );
glEnableClientState      ( GL_TEXTURE_COORD_ARRAY );
```

Расширения EXT_texture_env_add и ARB_texture_env_add

Стандартный OpenGL поддерживает следующие способы наложения текстуры: `GL_REPLACE`, `GL_BLEND`, `GL_DECAL` и `GL_MODULATE`.

Действие этих режимов поясняет табл. 2.1.

Таблица 2.1. *Стандартные режимы наложения текстуры в OpenGL*

Формат текстуры	**GL_REPLACE**	**GL_MODULATE**	**GL_DECAL**	**GL_BLEND**
GL_ALPHA	$C_v = C_f$ $A_v = A_t$	$C_v = C_f$ $A_v = A_f \cdot A_t$	Не определено	$C_v = C_f$ $A_v = A_f \cdot A_t$
GL_LUMINANCE	$C_v = C_t$ $A_v = A_f$	$C_v = C_f \cdot C_t$ $A_v = A_f$	Не определено	$C_v = C_f(1 - C_t) +$ $+ C_c \cdot C_t$ $A_v = A_f$
GL_LUMINANCE_ALPHA	$C_v = C_t$ $A_v = A_t$	$C_v = C_f \cdot C_t$ $A_v = A_f \cdot A_t$	Не определено	$C_v = C_f(1 - C_t) +$ $+ C_c \cdot C_t$ $A_v = A_f \cdot A_t$
GL_INTENSITY	$C_v = C_t$ $A_v = A_t$	$C_v = C_f \cdot C_t$ $A_v = A_f \cdot A_t$	Не определено	$C_v = C_f(1 - C_t) +$ $+ C_c \cdot C_t$ $A_v = A_f(1 - A_t) +$ $+ A_c \cdot A_t$
GL_RGB	$C_v = C_t$ $A_v = A_f$	$C_v = C_f \cdot C_t$ $A_v = A_f$	$C_v = C_t$ $A_v = A_f$	$C_v = C_f(1 - C_t) +$ $+ C_c \cdot C_t$ $A_v = A_f$
GL_RGBA	$C_v = C_t$ $A_v = A_t$	$C_v = C_f \cdot C_t$ $A_v = A_f \cdot A_t$	$C_v = (1 - A_t) \times$ $\times C_f + A_t \cdot C_t$ $A_v = A_f$	$C_v = C_f(1 - C_t) +$ $+ C_c \cdot C_t$ $A_v = A_f \cdot A_t$

Здесь через C обозначено RGB-значение цвета, через A — альфа-значение (альфа-компонента), индекс f обозначает RGB- или альфа-составляющую цвета, полученного из фрагмента, на который осуществляется наложение (значение, получаемое в результате вычисления освещенности или задаваемое командой `glColor`, в случае, когда расчет освещенности выключен), t обозначает, что соответствующее RGB- или альфа-значение получено из текстуры, c — RGB- или альфа-значение, заданное как `GL_TEXTURE_ENV_COLOR`, v — получившееся выходное значение.

Однако для получения реалистичных динамических изображений описанных способов наложения часто бывает недостаточно. Рассматриваемые далее расширения EXT_texture_env_add и ARB_texture_env_add предоставляют еще один режим наложения. Ему соответствует константа `GL_ADD` (одна и та же для обоих расширений). Действие этого режима описывается табл. 2.2.

Таблица 2.2. *Режим наложения текстуры* `GL_ADD`

Тип текстуры	GL_ADD
ALPHA	Rv = Rf Gv = Gf Bv = Bf Av = Af*At
LUMINANCE	Rv = min(1, Rf+Lt) Gv = min(1, Gf+Lt) Bv = min(1, Bf+Lt) Av = Af
LUMINANCE_ALPHA	Rv = min(1, Rf+Lt) Gv = min(1, Gf+Lt) Bv = min(1, Bf+Lt) Av = Af*At
INTENSITY	Rv = min(1, Rf+It) Gv = min(1, Gf+It) Bv = min(1, Bf+It) Av = min(1, Af+It)
RGB	Rv = min(1, Rf+Rt) Gv = min(1, Gf+Gt) Bv = min(1, Bf+Bt) Av = Af
RGBA	Rv = min(1, Rf+Rt) Gv = min(1, Gf+Gt) Bv = min(1,Bf+Bt) Av = Af*At

Никаких функций данное расширение не вводит.

Один из самых простых вариантов применения описанных расширений — это сложение двух текстур при операции мультитекстурирования, что необходимо при наложении бликов на поверхность объекта (рис. 2.4).

При этом на одном (например, нулевом) текстурном блоке задается первая текстура и операция наложения `GL_REPLACE`, а на следующем (втором в нашем случае) текстурном блоке — `GL_ADD`.

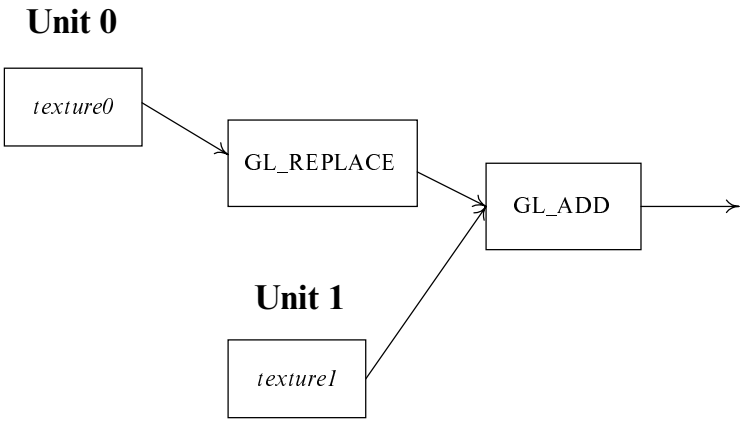


Рис. 2.4. Наложение бликов окружающей среды на главную текстуру при помощи `GL_ADD`

В листинге 2.6 приводится файл multitex4.cpp, демонстрирующий наложение отражения окружающей среды при помощи закона `GL_ADD`.

Листинг 2.6. Пример использования режима наложения текстуры `GL_ADD`

```
//
// Sample to show multitexturing in OpenGL.
//
#include     "libExt.h"
#include     "libTexture.h"
#include     "Vector3D.h"
#include     "Vector2D.h"
#include     <glut.h>
#include     <stdio.h>
#include     <stdlib.h>
unsigned texture1, texture2;
float      angle  = 0;
float      angle2 = 0;
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 );
     glActiveTextureARB ( GL_TEXTURE0_ARB );
     glEnable           ( GL_TEXTURE_2D );
     glBindTexture      ( GL_TEXTURE_2D, texture1 );
     glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                          GL_REPLACE );
       glActiveTextureARB ( GL_TEXTURE1_ARB );
       glEnable           ( GL_TEXTURE_2D );
       glEnable           ( GL_TEXTURE_GEN_S );
       glEnable           ( GL_TEXTURE_GEN_T );
       glTexGeni          ( GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP );
       glTexGeni          ( GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP );
       glBindTexture      ( GL_TEXTURE_2D, texture2 );
       glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                            GL_ADD );
       glPushMatrix    ();
       glTranslatef    ( 2, 2, 2 );
       glRotatef       ( angle,  1, 1, 0 );
       glRotatef       ( angle2, 0, 1, 1 );
       glutSolidTeapot ( 1 );
       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      ( 0.0, 0.0, 0.0,          // eye
                    5.0, 5.0, 5.0,          // center
                    0.0, 1.0, 0.0 );
}
void key ( unsigned char key, int x, int y )
```

```
{
  if ( key == 27 || key == 'q' || key == 'Q' )     // quit requested
       exit ( 0 );
   if ( key == 'p' || key == 'P' )          // saving screenshots
     saveScreenShot ( "multitex3.tga" );
}
void     animate ()
{
     angle  = 0.04f * glutGet ( GLUT_ELAPSED_TIME );
     angle2 = 0.01f * glutGet ( GLUT_ELAPSED_TIME );
     glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                              // initialize glut
     glutInit            ( &argc, argv );
     glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
     glutInitWindowSize  ( 400, 400 );
                              // create window
     int     win = glutCreateWindow ( "OpenGL Multitexture example 4" );
                              // register handlers
     glutDisplayFunc  ( display );
     glutReshapeFunc  ( reshape );
     glutKeyboardFunc ( key );
     glutIdleFunc     ( animate );
     initExtensions ();
     if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
     {
          printf ( "ARB_multitexture NOT supported.\n" );
          return 1;
     }
     texture1 = createTexture2D ( true, "../Textures/block.bmp" );
     texture2 = createTexture2D ( true, "../Textures/stars.bmp" );
     init ();
     glutMainLoop ();
     return 0;
}
```

Расширение EXT_fog_coord

Одним из самых простых объемных эффектов, существенно влияющих на восприятие сцены, является туман. При использовании тумана мы считаем, что какая-то область пространства заполнена некоторым веществом, частично поглощающим проходящий через него цвет.

При этом если через C_0 обозначить цвет объекта (который был бы виден при отсутствии тумана), а через C_f — цвет самого тумана, то интенсивность итогового цвета будет определяться следующей формулой:

$$C_r = C_0(1 - f) + C_f f, \tag{2.1}$$

где f — коэффициент (степень) "затуманивания" для соответствующей точки.

Стандартный OpenGL поддерживает задание тумана, но существующая для этого модель предполагает, что все пространство равномерно заполнено туманом с заданной неизменной плотностью, а сам процесс наложения тумана следующий:

1. Для каждой вершины вычисляется степень затуманивания в зависимости от ее расстояния d до наблюдателя (камеры).
2. После этого степень билинейно интерполируется на всю выводимую грань.

Степень затуманивания f вычисляется по одной из стандартных формул:

$$f = \exp(-d \cdot C),$$
$$f = \exp[(-d \cdot C)^2],$$
$$f = (d - start)/(end - start),$$

где *start* — начальное, а *end* — конечное значения расстояния наблюдения.

Реализовать другие варианты объемного тумана (например, слой тумана заданной толщины над поверхностью Земли, рис. 2.5) этим способом невозможно.

Для этого удобно воспользоваться расширением GL_EXT_fog_coord, поддерживаемым большинством современных графических ускорителей. Данное расширение позволяет пользователю самому задать величину затуманивания t для каждой вершины (т. е. фактически величина затуманивания t выступает в качестве еще одного атрибута вершины наравне с цветом, нормалью и текстурными координатами).

При выводе грани задаваемая в вершинах величина выступает в качестве параметра d при расчете затуманивания f по одной из трех стандартных моделей. При таком подходе степень "затуманивания" каждого пиксела грани

получается путем билинейной интерполяции значений, заданных в вершинах этой грани (рис. 2.6).

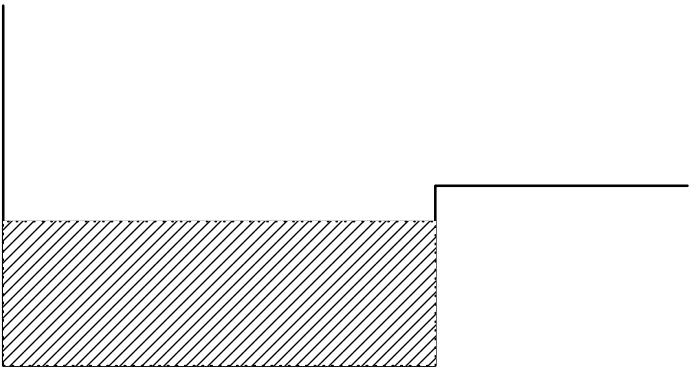


Рис. 2.5. Слой тумана

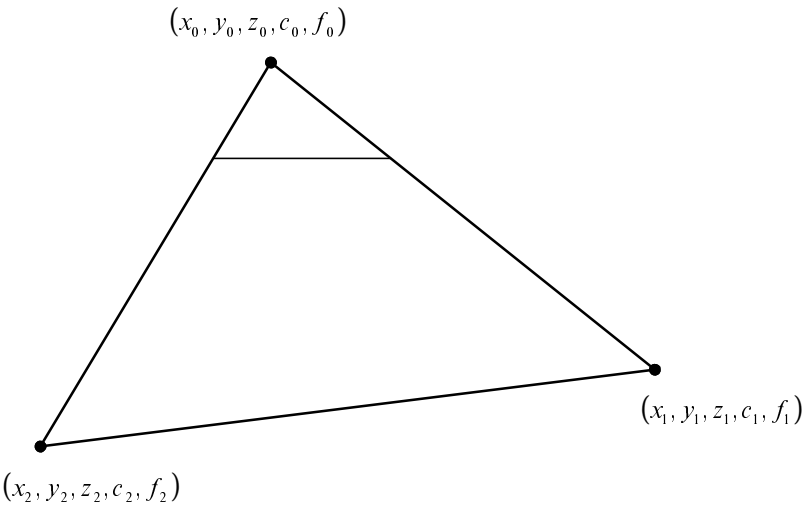


Рис. 2.6. Задание вершинного тумана

Данное расширение вводит три новые функции (`glFogCoordEXT`, `glFogCoordvEXT` и `glFogCoordPointerEXT`) и несколько констант. Их примеры приводятся далее.

```
void glFogCoord[fd]EXT    ( T coord )
void glFogCoord[fd]vEXT   ( T coord )
void glFogCoordPointerEXT ( GLenum type, GLsizei stride, GLvoid *pointer)
```

Для непосредственной работы с данным расширением необходимо настроить использование объемного тумана (листинг 2.7).

Листинг 2.7. Задание вершинного тумана

```
glEnable ( GL_FOG                );
glFogi   ( GL_FOG_MODE, GL_LINEAR );
glFogfv  ( GL_FOG_COLOR, fogColor );
glFogf   ( GL_FOG_START, fogStart );
glFogf   ( GL_FOG_END,   fogEnd   );
glFogi   ( GL_FOG_HINT, GL_NICEST );
glFogi   ( GL_FOG_COORDINATE_SOURCE_EXT, GL_FOG_COORDINATE_EXT );
```

Здесь командой `glEnable` включается режим тумана. После этого устанавливаются модель вычисления степени тумана, его цвет и параметры. Далее задается указание на использование более точного вычисления затуманивания (`GL_NICEST`).

При помощи следующей команды можно задать другое предпочтение — наиболее быстрый способ вычисления:

```
glFogi   ( GL_FOG_HINT, GL_FASTEST );
```

Последняя команда из листинга 2.7 определяет, что в качестве величины затуманивания d будут использованы явно заданные значения для каждой вершины.

Точное значение коэффициента f по заданному в вершине значению d вычисляется в данном примере по линейной модели:

$$f = \frac{d - fogStart}{fogEnd - fogStart}.$$

Значения величины затуманивания можно либо явно задавать в каждой вершине функцией `glFogCoordfEXT`, или же с помощью вершинных массивов:

```
glEnableClientState  ( GL_FOG_COORDINATE_ARRAY );
glFogCoordPointerEXT ( GL_FLOAT, 0, fogValues );
....
glDisableClientState ( GL_FOG_COORDINATE_ARRAY );
```

Рассмотрим теперь, каким образом можно явно вычислить степень затуманивания для заданной вершины некоторого объекта. Рассмотрим случай, когда у нас имеется слой тумана постоянной плотности, расположенный в диапазоне $y_0 \le y \le y_1$.

В этом случае степень затуманивания точки P фактически численно равна длине части отрезка, соединяющего положение камеры с этой точкой, лежащей внутри полосы тумана, умноженной на плотность тумана $\rho \cdot |AP|$ (рис. 2.7).

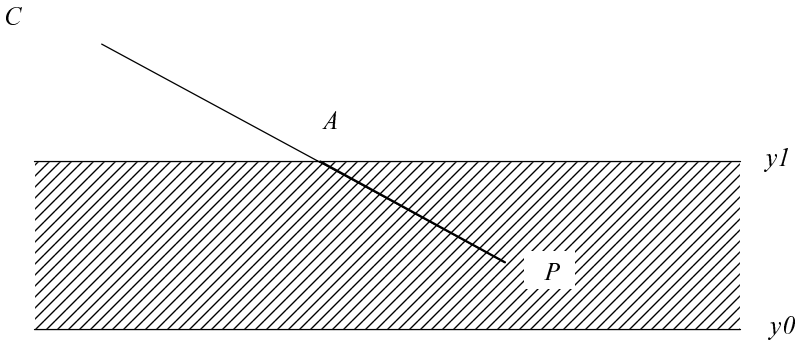


Рис. 2.7. Точное вычисление пути внутри слоя тумана

Подобные вычисления, особенно в случае, когда число вершин велико, могут сильно загрузить центральный процессор. Поэтому во многих случаях удобнее упрощенная модель (дающая в большинстве случаев вполне приемлемые результаты): вычисляется длина проекции отрезка на прямую, перпендикулярную плоскости тумана $\rho \cdot |A'P'|$. В нашем случае в качестве такой плоскости выступает ось Oy (рис. 2.8).

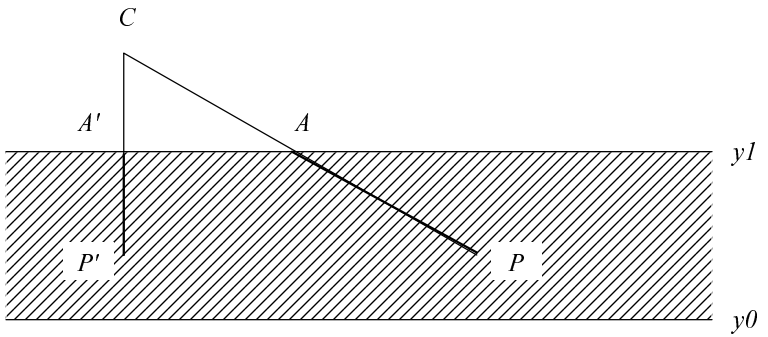


Рис. 2.8. Приближенное вычисление по сравнению с точным

Для вычисления этого достаточно (в случае, когда камеры расположена выше полосы тумана) просто взять разницу y-координаты самой точки P и $y1$, умножив ее на плотность тумана ρ.

Для закона `GL_LINEAR` вместо вычитания и задания плотности тумана можно просто задать соответствующим образом величины *start* и *end*. В этом случае в качестве величины *start* будет выступать $y1$, а величина *end* будет равна плотности тумана плюс $y1$:

$$start = y1;$$

$$end = \rho + y1.$$

Однако следует иметь в виду, что при преобразованиях объекта (при помощи матрицы *model-view*) в качестве степени затуманивания должна выступать *y*-координата уже преобразованной, а не исходной вершины. Все что, для этого следует сделать, — это получить текущую матрицу *model-view* при помощи `glGet` и для каждой вершины найти ее *y*-координату после преобразования этой матрицей. Реализация этого подхода приведена в листинге 2.8.

Листинг 2.8. Пример реализации слоя тумана при помощи расширения EXT_fog_coord

```
//
// Simple demo for GL_EXT_fog_coord OpenGL extension
//
#include     "libExt.h"
#include     "libTexture.h"
#include     "Vector3D.h"
#include     "Vector2D.h"
#include     <glut.h>
#include     <stdio.h>
#include     <stdlib.h>
#define     N1     90
#define     N2     20
struct     Facet
{
     int      index [3];  // indices into vertex, normal
                                  // and tex arrays
};
Vector3D     vertex [N1 * N2];
Vector3D     normal [N1 * N2];
Vector2D     tex    [N1 * N2];
Facet        facet  [N1 * N2 * 2];
```

```
float      angle  = 0;
float      angle2 = 0;
unsigned texture;                          // main texture
Vector3D     eye      ( 0, 0, 0  );
Vector3D     lookAt   ( 0, -5, 5 );
Vector3D     up       ( 0, 1, 0  );
Vector3D     fogColor ( 0, 0.7f, 0.8f );
inline     Vector3D     knot1D ( float t )
{
     float     r   = 1.8 + 0.8 * cos ( 3*t );
     float     phi = 0.2 * M_PI * sin ( 3*t );
     return r * Vector3D ( cos ( phi ) * sin ( 2*t ),
                           cos ( phi ) * cos ( 2*t ), sin ( phi ) );
}
inline     Vector3D     knot ( float u, float v, Vector3D& n )
{
     Vector3D     t, b;
     t = (knot1D (u + 0.01) - knot1D (u - 0.01)).normalize ();
     b = (t ^ Vector3D ( 0, 0, 1 )).normalize ();
     n =  t ^ b;
     return knot1D ( u ) + 0.55 * ( sin (v) * b + cos (v) * n );
}
void     initKnot ()
{
                          // 1. Create vertices
     for ( int i = 0, k = 0; i < N1; i++ )
     {
          float     phi = i * 2 * M_PI / N1;
          for ( int j = 0; j < N2; j++, k++ )
          {
               float     psi = j * 2 * M_PI / N2;
                         // compute vertex coordinates
               vertex [k] = knot ( phi, psi, normal [k] );
                         // compute texture coordinates
               tex [k].x = (float) i / (float) N1;
               tex [k].y = (float) j / (float) N2;
          }
     }
```

```
                        // 2. Create facets
     for ( i = k = 0; i < N1; i++ )
          for ( int j = 0; j < N2; j++, k += 2 )
          {
               facet [k].index [0] = i*N2 + j;
               facet [k].index [1] = ((i+1) % N1)*N2 + j;
               facet [k].index [2] = ((i+1) % N1)*N2 + (j+1)%N2;
               facet [k+1].index [0] = i*N2 + j;
               facet [k+1].index [1] = ((i+1) % N1)*N2 + (j+1)%N2;
               facet [k+1].index [2] = i*N2 + (j + 1) % N2;
          }
}
void     draw ()
{
     float     mvMatrix [16];
                                  // get modelview matrix to
                                  // transform back
     glGetFloatv ( GL_MODELVIEW_MATRIX, mvMatrix );
     glBegin ( GL_TRIANGLES );
     for ( int i = 0; i < N1 * N2 * 2; i++ )
          for ( int j = 0; j < 3; j++ )
          {
               int      k    = facet [i].index [j];
               Vector3D v   = vertex [k];
               float    fog = -v.x*mvMatrix[1] - v.y*mvMatrix[5] -
                                  v.z * mvMatrix [9] - mvMatrix [13];
               glFogCoordf   ( fog        );
               glNormal3fv   ( normal [k] );
               glTexCoord2fv ( tex    [k] );
               glVertex3fv   ( vertex [k] );
          }
     glEnd ();
}
void init ()
{
     glClearColor  ( 0, 0, 0, 1 );
     glEnable      ( GL_DEPTH_TEST );
     glDepthFunc   ( GL_LEQUAL );
```

```
    glEnable      ( GL_TEXTURE_2D );
    glBindTexture ( GL_TEXTURE_2D, texture );
                        // setup fog
    glEnable ( GL_FOG                );
    glFogi   ( GL_FOG_MODE, GL_LINEAR );
    glFogfv  ( GL_FOG_COLOR, fogColor );
    glFogf   ( GL_FOG_START, -4       );
    glFogf   ( GL_FOG_END,   3        );
    glFogi   ( GL_FOG_HINT, GL_NICEST );
    glFogi   ( GL_FOG_COORDINATE_SOURCE_EXT, GL_FOG_COORDINATE_EXT );
}
void display ()
{
    glClear         ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glPushMatrix    ();
    glTranslatef    ( 0, -7, 7 );
    glRotatef       ( angle,  1, 1, 0 );
    glRotatef       ( angle2, 0, 1, 1 );
    draw ();
    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
                    lookAt.x, lookAt.y, lookAt.z,      // center
                    up.x, up.y, up.z );
}
void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )     // quit requested
        exit ( 0 );
```

```
    if ( key == 'p' || key == 'P' )               // saving screenshots
          saveScreenShot ( "fog.tga" );
}
void     animate ()
{
     angle  = 0.04f * glutGet ( GLUT_ELAPSED_TIME );
     angle2 = 0.02f * 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
     int     win = glutCreateWindow ( "OpenGL fog_coord example" );
                         // register handlers
     glutDisplayFunc  ( display );
     glutReshapeFunc  ( reshape );
     glutKeyboardFunc ( key     );
     glutIdleFunc     ( animate );
     initExtensions ();
if ( !isExtensionSupported ( "GL_EXT_fog_coord" ) )
     {
          printf ( "GL_EXT_fog coord NOT supported.\n" );
          return 1;
     }
     texture = createTexture2D ( true, "../Textures/concgrey.bmp" );
     init     ();
     initKnot ();
     glutMainLoop ();
     return 0;
}
```

На рис. 2.9 приводится изображение, построенное данной программой.

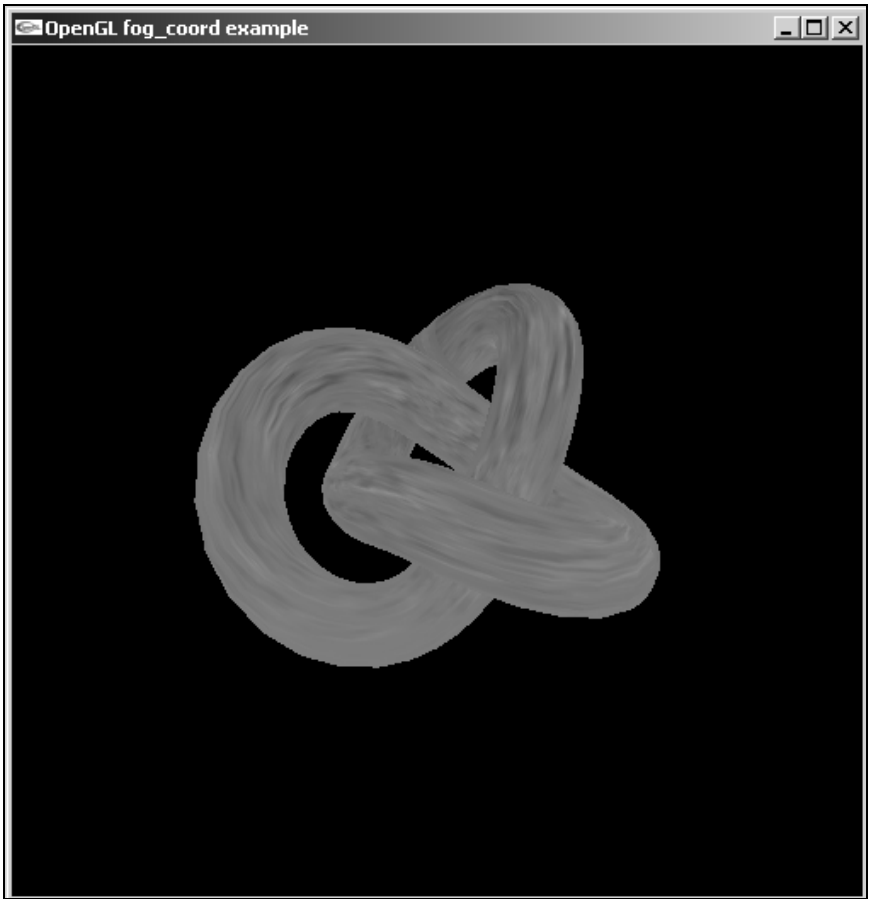


Рис. 2.9. Результат выполнения листинга 2.8 ("узел в тумане")

Расширения EXT_secondary_color и EXT_separate_specular_color

Стандартная модель освещения, используемая OpenGL, обладает определенными недостатками, одним из которых является следующий: цвет блика на поверхности объекта всегда совпадает с цветом самой поверхности. Это связано с тем, что освещение может лишь изменять яркость цвета на поверхности объекта, но не сам цвет (что не соответствует действительности). Многие объекты реального мира демонстрируют совершенно иное поведение, например, у зеленого пластмассового шара блик будет белого цвета (при освещении его белым светом).

Для преодоления подобного недостатка можно осуществить вывод объектов в два прохода: на первом проходе выводится фоновое и диффузное освещение, а на втором — только бликовое. За счет того, что бликовое освещение выводится отдельным проходом, появляется возможность управлять цветом блика, однако расплачиваться за это приходится уменьшением быстродействия программы.

Более простой способ реализуется расширениями EXT_secondary_color и EXT_separate_specular_color. Первое из них позволяет задавать для каждой вершины второй цвет. Второе — позволяет использовать этот второй цвет в качестве цвета блика на поверхности.

Для задания второго цвета в вершине служат функции

```
void glSecondaryColor3[bsifd ubusui]EXT  ( T components );
void glSecondaryColor3[bsifd ubusui]vEXT ( T components );
void glSecondaryColorPointerEXT ( GLint size, GLenum type,
                                  GLsizei stride,
                                  GLvoid *pointer );
```

Первые две позволяют задавать цвет для каждой вершины и аналогичны команде `glColor`. Функция `glSecondaryColorPointerEXT` служит для задания массива цветов для вершин при использовании вершинных массивов.

Обратите внимание, что второй (*secondary*) цвет всегда задается своими первыми тремя компонентами, альфа-компонента второго цвета не задается и всегда равна нулю.

Второе расширение — EXT_separate_specular_color — позволяет задать новую модель освещения для второго цвета в вершинах (цвета блика). Эта модель включается при помощи следующей команды:

```
glLightModeli  ( GL_LIGHT_MODEL_COLOR_CONTROL_EXT,
                 GL_SEPARATE_SPECULAR_COLOR_EXT );
```

В результате второй цвет интерполируется вдоль грани, умножается на бликовую составляющую освещенности и добавляется к цвету фрагмента уже после применения текстуры. При этом фактически вводится новый шаг в конвейере рендеринга OpenGL (рис. 2.10), называемый сложением цветов (*color sum*).

Важно, что второй цвет оказывает влияние на результирующий цвет фрагмента только через механизм вершинного освещения OpenGL.

Нормальный режим освещения можно включить командой

```
glLightModeli  ( GL_LIGHT_MODEL_COLOR_CONTROL_EXT,
                 GL_SINGLE_COLOR_EXT );
```

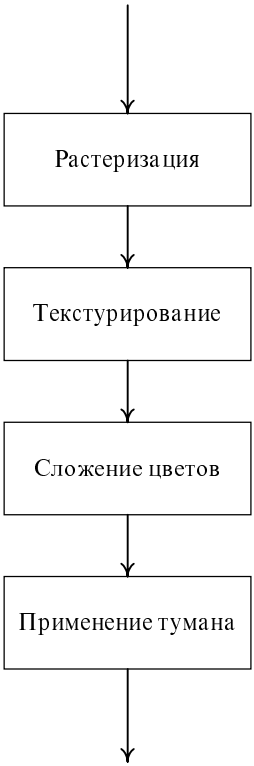


Рис. 2.10. Конвейер рендеринга

На самом деле этой моделью освещения мы в дальнейшем пользоваться не будем (поскольку мы будем применять попиксельное освещение), однако возможность задания второго цвета в вершинах нам очень пригодится.

Расширения ARB_texture_border_clamp и EXT_texture_edge_clamp

При текстурировании OpenGL приводит каждую компоненту текстурных координат вершины в диапазон [0, 1]. Способ исполнения задается функцией `glTexParameteri`, например, следующий фрагмент кода задает в качестве режима приведения повторение.

```
    // set texture to repeat mode
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
```

Однако когда такое усечение происходит одновременно с пирамидальным фильтрованием (*mipmapping*), то может возникнуть следующая проблема. Если значение текстурных координат лежит вне единичного квадрата (единичного отрезка в случае одномерной текстуры), то на полученное значение текстуры окажут влияние цвета текселов не только с границы текстуры, но и из ее внутренних точек (рис. 2.11).

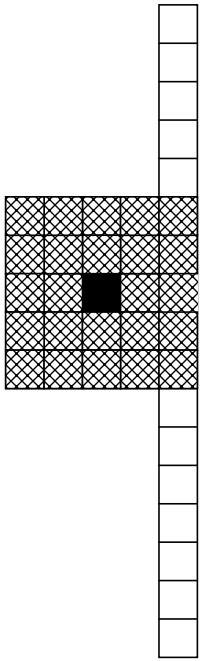


Рис. 2.11. Часть текстуры, используемая для получения значения при пирамидальном фильтровании

Если значение текстурных координат намеренно задано вне единичного квадрата, то в ряде случаев оказывается крайне нежелательным, чтобы на полученное при текстурировании значение оказывали влияние внутренние точки текстуры.

Возможность избежать этого предоставляет расширение ARB_texture_border_clamp. Если в качестве режима отсечения текстурных координат задать вводимый этим расширением режим `GL_CLAMP_TO_BORDER_ARB`, то для текстурных координат, сильно выходящих за единичный квадрат, при пирамидальном фильтровании будут использоваться только граничные текселы (или цвет границы, если он задан, рис. 2.12).

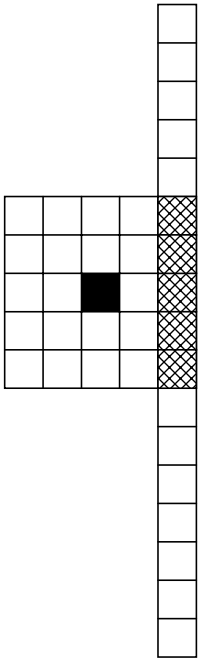


Рис. 2.12. Часть текстуры, используемая для получения значения при пирамидальном фильтровании для значений, сильно выходящих за единичный квадрат, в режиме `GL_CLAMP_TO_BORDER_ARB`

Вот несколько примеров работы с ARB_texture_border_clamp:

```
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                  GL_CLAMP_TO_BORDER_ARB );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                  GL_CLAMP_TO_BORDER_ARB );
```

С другой стороны, в ряде случаев желательно, чтобы цвет границы не оказывал бы никакого влияния на цвет получаемого пиксела при пирамидальном фильтровании.

Именно такую возможность предоставляет расширение EXT_texture_edge_clamp. При задании вводимого этим расширением режима `GL_CLAMP_TO_EDGE_EXT` гарантируется, что ни для какого уровня в пирамидальном фильтровании фильтр не затронет граничных пикселов (рис. 2.13).

Начиная с версии 1.2, режим `GL_CLAMP_TO_EDGE` вошел в стандарт OpenGL. Далее приведен формат команд для данного режима.

```
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                  GL_CLAMP_TO_EDGE );
```

```
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                  GL_CLAMP_TO_EDGE );
```

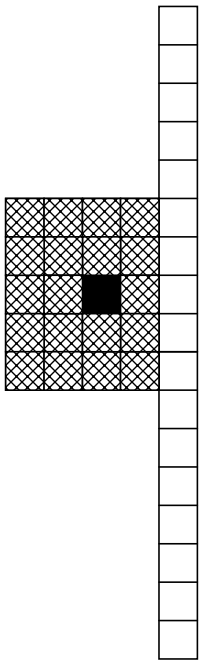
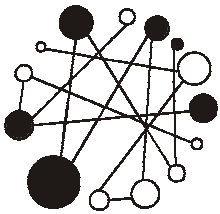


Рис. 2.13. Часть текстуры, используемая для получения значения при пирамидальном фильтровании для значений, сильно выходящих за единичный квадрат, в режиме `GL_CLAMP_TO_EDGE_EXT`

Глава 3



Расширения SGIS_generate_mipmap, EXT_bgra, EXT_abgr, XT_texture_filter_anisotropic и ARB_texture_non_power_of_two

Расширение SGIS_generate_mipmap

Любое изображение на экране по своей природе дискретно, поскольку представляется в виде регулярной прямоугольной матрицы пикселов. Следовательно, правильное пирамидальное фильтрование (*mipmapping*) играет большую роль в получении качественных изображений с малыми ошибками дискретизации (*aliasing artifacts*).

Пирамидальное фильтрование позволяет заметно уменьшить такие погрешности за счет использования вместо одной исходной текстуры некоторого набора текстур. Текстуры этого набора строятся рекурсивно, начиная с исходной и заканчивая текстурой размером 1×1 (рис. 3.1).

Для построения очередной текстуры по предыдущей ее текселы разбиваются в группы размером 2×2. Каждой такой группе соответствует всего один тексел в строящейся текстуре, значение для него получается путем усреднения значений по четырем текселам группы (рис. 3.2).

Сама библиотека OpenGL не содержит в себе средства для корректного построения всех этих промежуточных текстур. При явной загрузке текстуры из файла можно воспользоваться предоставляемой библиотекой GLU функцией `gluBuild2DMipmap` (для случая двумерных текстур). Также можно явно загрузить заранее построенные промежуточные уровни вместе с самой текстурой (например, DDS-файлы позволяют хранить вместе с основной весь необходимый набор промежуточных текстур).

Однако применение этой функции не всегда возможно и удобно, например, если в качестве текстуры мы берем изображение не из файла, а построенное средствами самого OpenGL (т. е. скорее всего находящееся в памяти графи-

ческого ускорителя), то работа с этой функцией оказывается довольно неудобной и очень медленной.

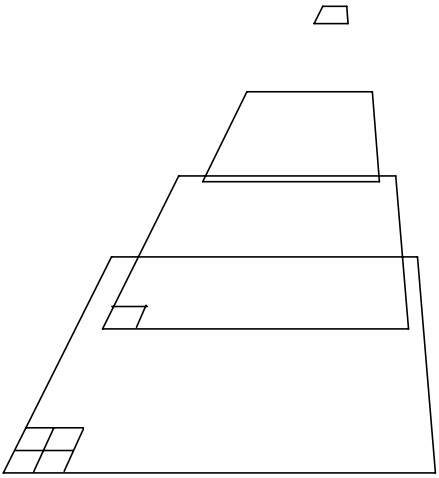


Рис. 3.1. Пирамида текстур

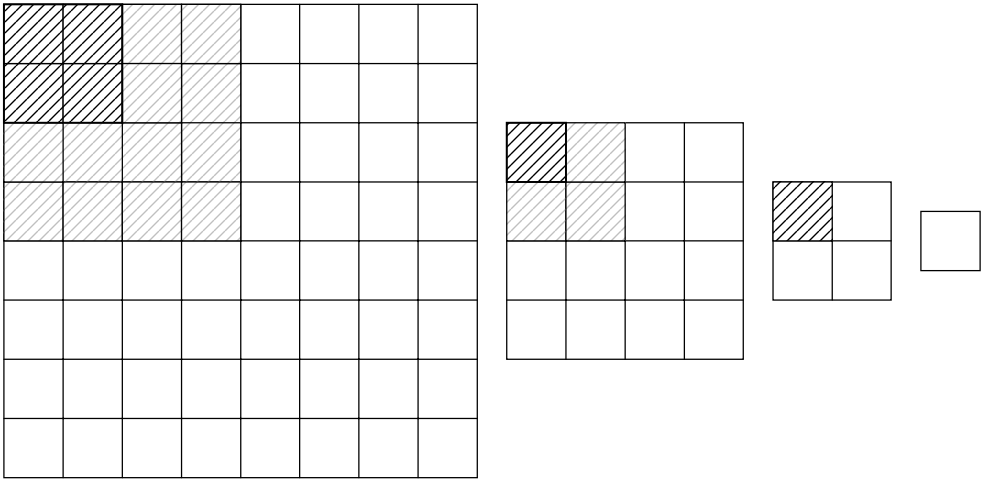


Рис. 3.2. Построение уровней пирамиды

Кроме того, построение всех промежуточных уровней непосредственно процессором часто бывает еще и неэффективным, поскольку на это тратится процессорное время и увеличивается объем передаваемых графическому ус-

корителю данных, что также отрицательно сказывается на быстродействии приложения.

Расширение SGIS_generate_mipmap определяет механизм, позволяющий OpenGL самому строить все необходимые для применения пирамидального фильтрования промежуточные текстуры по заданному начальному изображению (текстуре). Для этого достаточно у соответствующей текстуры задать свойство `GL_GENERATE_MIPMAP_SGIS`, равное `GL_TRUE`. Тогда любое изменение содержимого текстуры приводит к автоматическому пересчету всех промежуточных уровней (текстур).

Таким образом, для того чтобы для выбранной (командой `glBindTexture`) текстуры включить режим автоматического построения всех необходимых карт пирамидального фильтрования (этот режим задается отдельно для каждой текстуры), достаточно выполнить следующую команду:

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

Выключение этого режима для выбранной текстуры осуществляется командой

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

Параметр `target` определяет тип текстуры, для которой будет включен (или выключен) этот режим. Для одно- и двумерных текстур параметр принимает значения `GL_TEXTURE_1D` и `GL_TEXTURE_2D` соответственно.

Расширение SGIS_generate_mipmap позволяет также задать предпочтения (*hints*) для способа, используемого при построении промежуточных текстур.

- `glHint ( GL_GENERATE_MIPMAP_HINT_SGIS, GL_FASTEST );`

  сообщает OpenGL, что желательно выбрать наиболее быстрый способ построения промежуточных текстур.

- `glHint ( GL_GENERATE_MIPMAP_HINT_SGIS, GL_NICEST );`

  устанавливает, что предпочтение следует отдать методу, обеспечивающему наибольшее качество.

- `glHint ( GL_GENERATE_MIPMAP_HINT_SGIS, GL_DONT_CARE );`

  сообщает, что приложению все равно, какой способ построения промежуточных текстур будет использован.

Расширения EXT_bgra и EXT_abgr

В стандартном OpenGL для RGB- и RGBA-текстур жестко задан порядок следования цветовых компонентов для каждого пиксела (рис. 3.3).

Однако в так называемых аппаратно-независимых картах (*Device Independent Bitmap*, *DIB*), широко используемых в Microsoft Windows, принят по умолчанию совсем другой порядок байтов (рис. 3.4).

Рис. 3.3. Порядок байтов для RGB- и RGBA-текстур

Рис. 3.4. Порядок байтов для DIB

Это обстоятельство делает невозможным непосредственное использование информации из DIB в качестве данных текстуры для OpenGL, поскольку сначала необходимо осуществить перестановку байтов согласно порядку, принятому в OpenGL. Однако с другой стороны было бы очень удобно, если бы данные из DIB можно было напрямую брать при задании текстур (поскольку в Windows есть много функций и библиотек для работы с DIB).

Именно это позволяет расширение EXT_bgra. Оно вводит два новых формата, идентифицируемых константами `GL_BGR_EXT` и `GL_BGRA_EXT`, которые могут служить в качестве параметра `format` при задании текстур. Порядок следования байтов в этих форматах соответствует DIB, что значительно облегчает задание текстур. Еще одним преимуществом указанных форматов является ускорение работы с текстурами (по крайней мере, для графических ускорителей GeForce).

Аналогично, расширение EXT_abgr вводит еще один формат специально для RGBA-текстур, более удобных при другом порядке байтов в слове (так называемые *big endian*).

В табл. 3.1 приводятся доступные форматы текстур с учетом рассмотренных расширений.

Таблица 3.1. *Допустимые форматы текстур*

Константа	Формат	Буфер
`GL_DEPTH_COMPONENT`	Значение глубины	Буфер глубины
`GL_RED`	R	Буфер цвета
`GL_GREEN`	G	Буфер цвета
`GL_BLUE`	B	Буфер цвета
`GL_ALPHA`	A	Буфер цвета
`GL_RGB`	R, G, B	Буфер цвета

Таблица 3.1 *(окончание)*

Константа	Формат	Буфер
GL_RGBA	R, G, B, A	Буфер цвета
GL_LUMINANCE	L	Буфер цвета
GL_LUMINANCE_ALPHA	L, A	Буфер цвета
GL_BGR_EXT	B, G, R	Буфер цвета
GL_BGRA_EXT	B, G, R, A	Буфер цвета
GL_ABGR_EXT	A, B, G, R	Буфер цвета

Расширение EXT_texture_filter_anisotropic

При работе с текстурами для получения цвета пиксела служит так называемое текстурное фильтрование (*texture filtering*) — некий механизм, выдающий значение цвета исходя из положения и ориентации исходного фрагмента в пространстве.

В самом простейшем случае текстурное фильтрование заключается просто в выборке одного тексела из текстуры (GL_NEAREST). Однако при столь простом фильтровании в получающихся изображениях часто появляются заметные погрешности, ошибки дискретизации (*aliasing artifacts*).

Их уменьшение обычно осуществляется путем пирамидального фильтрования (*mipmapping*), когда вместо одного тексела берется квадратный блок и возвращается усредненное значение всех его текселов. Обычно чтобы не проводить эту операцию на этапе рендеринга, заранее строится набор промежуточных текстур, где каждому текселу промежуточной соответствует блок текселов исходной текстуры, и при выводе просто выбирается значение с одной из таких промежуточных текстур.

Фактически любое текстурное фильтрование заключается в выборе блока текселов на исходной текстуре и усреднении их значений с некоторыми весовыми коэффициентами.

Однако данная операция хорошо работает, когда конфигурация исходного и преобразованного блоков текселов на экране совпадает. Подобный случай называется изотропным.

Часто это не так; бывает, что блок текселов неравномерно масштабируется при проектировании (рис. 3.5). Такой случай получил название анизотропного.

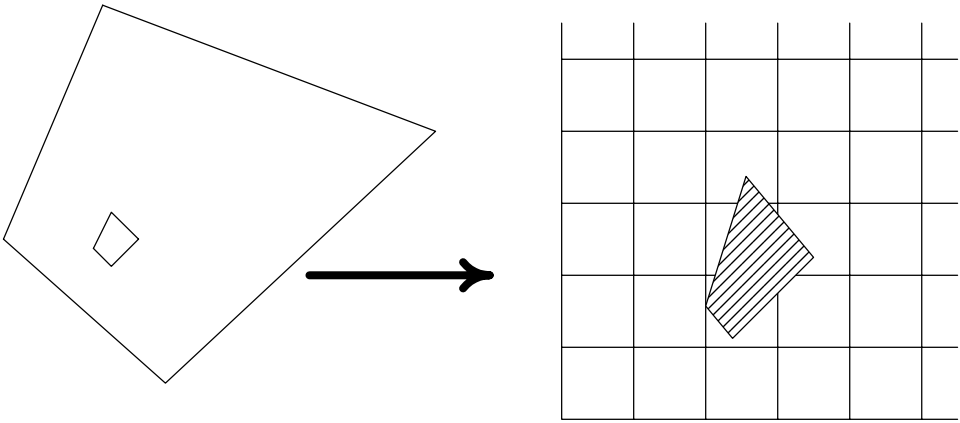


Рис. 3.5. Неравномерное преобразование блока текселов при проектировании

Если в этом случае используется обычное (изотропное) фильтрование, то на изображении могут появиться различные искажения. Для борьбы с ними существуют специальные методы, получившие название анизотропной фильтрации.

Расширение EXT_texture_filter_anisotropic предоставляет программисту механизм, позволяющий задавать желаемую степень анизотропии (чем она выше, тем фильтрация будет качественнее). Сам механизм осуществления анизотропной фильтрации скрыт от пользователя и возлагается на драйвер.

Данное расширение позволяет индивидуально для каждой текстуры задать свое значение анизотропии. Чем больше это значение, тем выше качество. Но за повышение качества приходится расплачиваться некоторым снижением быстродействия. Стандартной фильтрации в OpenGL соответствует значение анизотропии, равное единице. Расширение EXT_texture_filter_anisotropic поддерживает значение анизотропии не менее двух.

Максимально возможное значение анизотропии, поддерживаемое графическим ускорителем и его драйвером, можно узнать при помощи следующего фрагмента кода:

```
int         maxAniso;
glGetIntegerv (GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &maxAniso );
```

Задание уровня анизотропии для текущей текстуры осуществляется командой

```
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, aniso );
```

В листинге 3.1 приводится исходный код примера, демонстрирующего анизотропную фильтрацию для управления качеством получаемого изображения.

При помощи мыши можно поворачивать камеру, правда, в этом примере возможны лишь повороты налево/направо. Нажатие левой кнопки мыши

перемещает камеру вперед. Клавишами <+> и <–> можно изменять текущий уровень анизотропной фильтрации, при этом его значение отображается в заголовке окна.

Листинг 3.1. Пример использования анизотропной фильтрации

```
//
// Sample to show anisotropic texture filtering in OpenGL
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <list>
using namespace std;
#include    "libTexture.h"
#include    "TypeDefs.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "boxes.h"
#define        CAPTION        "OpenGL anisotropic filtering example"
Vector3D    eye   ( 0, 0, -2 );         // camera position
Vector3D    viewDir ( 1, 0, 0 );        // viewer direction
unsigned    stoneMap;
int         mouseOldX = 0;
int         mouseOldY = 0;
Vector3D    rot ( 0, 0, 0 );
float       step = 0.003;            // movement rate per second
bool        mousePressed = false;
float       roomSize     = 50;        // room half size
int         maxAniso     = 1;
int        curAniso     = 1;
void    setupCamera ();
void    setAnisotropy ( int aniso )
{
    char    str [256];
    if ( aniso < 1 || aniso > maxAniso )
        return;
    curAniso = aniso;
    glBindTexture   ( GL_TEXTURE_2D, stoneMap );
```

```
    glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT,
                      curAniso );
       sprintf         ( str, "%s - anisotropy = %2d", CAPTION,
                      curAniso );
       glutSetWindowTitle ( str );
   }
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 main hall
    drawBox  ( Vector3D ( -roomSize, -roomSize, -3 ),
               Vector3D ( 2*roomSize, 2*roomSize, 7 ),
               stoneMap, false );
    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, 0.1, 1000 );
    setupCamera    ();
}
void    setupCamera ()
{
    Vector3D    to = eye + viewDir;
       glMatrixMode   ( GL_MODELVIEW );
       glLoadIdentity ();
       gluLookAt      ( eye.x, eye.y, eye.z,    // eye
                        to.x,  to.y,  to.z,     // center
                        0.0, 0.0, 1.0 );        // up
```

```
}
void motion ( int x, int y )
{
    static    bool    mouseInited = false;
    if ( !mouseInited )
    {
        mouseInited = true;
        mouseOldX   = x;
        mouseOldY   = y;
        return;
    }
    float    scale = -M_PI / 100;
    viewDir.x = cos ( scale * x );
    viewDir.y = sin ( scale * x );
    viewDir.z = 0;
    setupCamera ();
    glutPostRedisplay ();
}
void mouse ( int button, int state, int x, int y )
{
    if ( button == GLUT_LEFT_BUTTON )
        mousePressed = (state == GLUT_DOWN);
    motion ( x, y );
}
void key ( unsigned char key, int x, int y )
{
    char    str [256];
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
               exit ( 0 );
    if ( key == '+' )
            setAnisotropy ( curAniso + 1 );
       else
       if ( key == '-' )
            setAnisotropy ( curAniso - 1 );
}
void    animate ()
{
    static    bool    timeInited = false;
    static    int    lastTime;
```

```
    if ( !timeInited )
    {
        lastTime    = glutGet ( GLUT_ELAPSED_TIME );
        timeInited = true;
    }
    if ( mousePressed )
    {
        Vector3D    delta = viewDir * (step *
                     (glutGet ( GLUT_ELAPSED_TIME ) - lastTime)*0.001f);
        Vector3D    e     = eye + delta;
        if ( e.x < roomSize && e.x > -roomSize )
            eye.x = e.x;
        if ( e.y < roomSize && e.y > -roomSize )
            eye.y = e.y;
        setupCamera ();
    }
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                            // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                            // create window
    int    win = glutCreateWindow ( CAPTION );
                            // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutMouseFunc    ( mouse   );
    glutMotionFunc   ( motion  );
    glutPassiveMotionFunc   ( motion  );
    glutIdleFunc     ( animate );
    init           ();
    initExtensions ();
    printfInfo     ();
    if ( !isExtensionSupported ("GL_EXT_texture_filter_anisotropic") )
```

```
    {
        printf ("EXT_texture_filter_anisotropic NOT supported.\n");
        return 1;
    }
    stoneMap = createTexture2D ( true, "../Textures/block.bmp" );
    glGetIntegerv (GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &maxAniso );
    printf ( "Max supported anisotry %d\n", maxAniso );
    setAnisotropy ( 1 );
    glutMainLoop ();
    return 0;
}
```

Расширение ARB_texture_non_power_of_two

Как известно, стандартный OpenGL требует, чтобы размеры всех текстур (одно-, дву- и трехмерных, а также кубических текстурных карт) являлись степенями двойки.

Однако на практике это не всегда удобно, поэтому в ряде случаев приходится прибегать к масштабированию текстуры для изменения ее размера. Существует даже несколько расширений (например, NV_texture_rectangle), позволяющих работать с текстурами, размеры которых не являются степенью двойки, однако вводимые ими текстуры имеют свои особенности, затрудняющие работу.

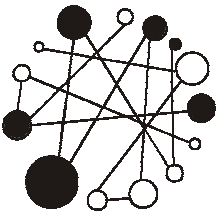
Расширение ARB_texture_non_power_of_two дает возможность приложению свободно оперировать произвольными размерами для одно-, дву- и трехмерных текстур, а также кубических текстурных карт.

Обратите внимание, что данное расширение не вводит никаких новых функций или констант. Просто сам факт его поддержки снимает ограничение на размеры всех стандартных текстур. Текстуры, размеры которых не являются степенями двойки, ведут себя точно так же, как и текстуры стандартного размера. Для них поддерживается пирамидальное фильтрование (*mipmapping*), могут задаваться различные режимы наложения и отсечения текстурных координат и т. п.

Важно помнить, что каждая компонента текстурных координат для таких текстур принимает значения из отрезка [0, 1], как и для стандартных текстур (в отличие, например, от текстур, вводимых расширением NV_texture_rectangle).

Иными словами, это обычные текстуры, ничем не отличающиеся от текстур с размерами, являющимися степенями двойки. Поэтому весь код для загрузки и работы с подобными текстурами ничем не отличается от обычного.

Глава 4



Расширения EXT_texture_env_combine и ARB_texture_env_combine. Их применение

Как уже отмечалось в *главе 2*, стандартный OpenGL поддерживает 4 способа наложения текстуры: `GL_REPLACE`, `GL_BLEND`, `GL_DECAL` и `GL_MODULATE`. Пояснения к этим режимам были приведены в табл. 2.1.

Задать значения для `GL_TEXTURE_ENV_COLOR` можно, выполнив фрагмент кода

```
float    color [] = { 1, 0.5, 0.3, 1 };
glTexEnvfv ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, color );
```

Однако для получения изображений в реальном времени этого, как уже отмечалось, часто бывает недостаточно. Расширение Texture_env_add, введенное в *главе 2*, добавляет еще один режим. Но во многих случаях желательно иметь гораздо бóльшую гибкость при задании законов наложения текстуры. Именно такую возможность и дают расширения ARB_texture_env_combine и EXT_texture_env_combine. Они позволяют задавать режимы наложения текстуры довольно общего вида, причем раздельно для цветовых компонент (RGB) и для альфа-канала (*alpha*), т. е. для цветовых каналов можно задать один режим наложения, а для альфа-канала — совершенно другой.

Проверку поддержки этих расширений можно выполнить командами

```
isExtensionSupported ( "GL_ARB_texture_env_combine" )
isExtensionSupported ( "GL_EXT_texture_env_combine" )
```

Данные расширения не вводят дополнительных функций, но добавляют целый набор новых констант. Поскольку оба эти расширения практически одинаковы, мы далее будем рассматривать только ARB_texture_env_combine, для EXT_texture_env_combine, как правило, достаточно заменить в константах окончание ARB на EXT. Более подробно различие этих расширений будет рассмотрено в конце главы.

В табл. 4.1 приведены режимы наложения, обеспечиваемые этими расширениями (отдельно для цветовых компонент и альфа-компоненты).

Таблица 4.1. *Новые режимы наложения*

Тип	Формула наложения
GL_REPLACE	Arg0
GL_MODULATE	Arg0 · Arg1
GL_ADD	Arg0 + Arg1
GL_ ADD _SIGNED_ARB	Arg0 + Arg1 - 0.5
GL_SUBTRACT_ARB	Arg0 - Arg1
GL_INTERPOLATE_ARB	Arg0 · Arg2 + Arg1 · (1 - Arg2)

Значения операндов Arg0, Arg1 и Arg2 не являются жестко заданными, программист сам определяет, откуда следует взять данные для каждого из операндов (как цветовой, так и альфа-частей), и какому преобразованию их следует подвергнуть.

В табл. 4.2 приводятся возможные источники данных (как для цветовой, так и для альфа-частей) для операндов Arg0, Arg1 и Arg2.

Таблица 4.2. *Источники для операндов Arg0, Arg1 и Arg2*

Константа	Соответствующий источник
GL_PRIMARY_COLOR_ARB	Основной цвет фрагмента (C_f, A_f)
GL_TEXTURE	Цвет текстуры (C_t, A_t)
GL_CONSTANT_ARB	Цвет, заданный при помощи glTexEnfv
GL_PREVIOUS_ARB	Результат предыдущей операции, для первого текстурного блока эквивалентен GL_PRIMARY_COLOR_ARB

Поскольку RGB- и альфа-компоненты для операндов Arg0, Arg1 и Arg2 задаются независимо, то цвет и альфа-значение могут браться из разных источников.

Кроме того, для каждого из операндов (точнее, RGB- и альфа-частей этих операндов) можно указать, используется ли в качестве соответствующего операнда само исходное значение (в соответствии с табл. 4.2 C или alpha) или же его дополнение до единицы (т. е. 1 - C или 1 - alpha соответственно).

Также можно умножить результат операции (отдельно для цветовых и альфа-компоненты) на множитель 2 или 4.

Рассмотрим теперь подробнее, каким образом осуществляется задание всех этих параметров.

Сначала при помощи команды

```
glTexEnvi ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_ARB );
```

задается режим наложения `GL_COMBINE_ARB`. После этого следует задать закон наложения отдельно для RGB- и альфа-части.

Задание закона наложения осуществляется командой

```
glTexEnvi ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB, expr );
```

для цветовой части и

```
glTexEnvi ( GL_TEXTURE_ENV, GL_COMBINE_ALPHA_ARB, expr );
```

для альфа-части.

Параметр `expr` задает используемый закон наложения и принимает одно из значений, перечисленных в табл. 4.1.

Далее следует для каждого из операндов (`Arg0`, `Arg1` и `Arg2`) и каждой его части (цветовой и альфа) задать, откуда берется значение, и какому преобразованию оно подвергается.

Для задания того, откуда следует взять соответствующую часть операнда, служит вызов:

```
glTexEnvi ( GL_TEXTURE_ENV, operand, expr );
```

Здесь параметр `operand` задает как сам операнд (его номер — 0, 1, 2), так и его часть (цветовую или альфа) и принимает одно из значений согласно табл. 4.3.

Таблица 4.3. *Возможные источники данных для операндов*

Операнд	**Смысл**
`GL_SOURCE0_RGB_ARB`	Цветовая часть для операнда `Arg0`
`GL_SOURCE0_ALPHA_ARB`	Альфа-часть для операнда `Arg0`
`GL_SOURCE1_RGB_ARB`	Цветовая часть для операнда `Arg1`
`GL_SOURCE1_ALPHA_ARB`	Альфа-часть для операнда `Arg1`
`GL_SOURCE2_RGB_ARB`	Цветовая часть для операнда `Arg2`
`GL_SOURCE2_ALPHA_ARB`	Альфа-часть для операнда `Arg2`

Параметр `expr` задает источник для получения соответствующего операнда и принимает одно из значений согласно табл. 4.2.

После этого следует задать способ получения операнда из данного источника при помощи команды

```
glTexEnvi ( GL_TEXTURE_ENV, operand, method );
```

Параметр `operand` задается табл. 4.4. Способ получения соответствующего операнда из источника определяется параметром `method`, возможные значения для которого перечислены в табл. 4.5.

Таблица 4.4. *Операнды*

Значение	Компонент	Операнд
GL_OPERAND0_RGB_ARB	RGB	Arg0
GL_OPERAND0_ALPHA_ARB	Alpha	Arg0
GL_OPERAND1_RGB_ARB	RGB	Arg1
GL_OPERAND1_ALPHA_ARB	Alpha	Arg1
GL_OPERAND2_RGB_ARB	RGB	Arg2
GL_OPERAND2_ALPHA_ARB	Alpha	Arg2

Таблица 4.5. *Преобразования для операндов*

Значение	Смысл
GL_SRC_COLOR	C
GL_ONE_MINUS_SRC_COLOR	1 - C
GL_SRC_ALPHA	alpha
GL_ONE_MINUS_SRC_ALPHA	1 - alpha

Обратите внимание, что для RGB-канала в качестве источника можно взять как RGB-, так и альфа-канал. В последнем случае все цветовые компоненты будут равны значению `alpha`.

Рассмотрим задание более подробно. Далее приведены примеры.

Пример 1

Команды

```
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB, GL_TEXTURE );
glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND0_RGB_ARB, GL_SRC_COLOR );
```

задают, что в качестве источника данных для Arg0 следует взять текстуру (первая команда), а в качестве операции преобразования выступает операция получения цветовой части.

Таким образом, эти команды задают следующий закон задания операнда Arg0:

```
Arg0.rgb = texture.rgb
```

Пример 2

Следующие две команды

```
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB, GL_TEXTURE );
glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND1_RGB_ARB,
            GL_ONE_MINUS_SRC_ALPHA );
```

задают более сложный закон операнда Arg1. Источником по-прежнему является текстура, однако в качестве преобразования выступает операция 1 - alpha, т. е. все три цветовые значения будут равны 1 - alpha.

В этом случае закон задания операнда Arg1 выглядит следующим образом:

```
Arg1.rgb = rgb ( 1 - texture.alpha )
```

После того как все необходимые части всех используемых операндов будут заданы, для цветовой и альфа-частей можно задать числовые множители, на которые следует умножить получаемый результат. Для этого служит вызов

```
glTexEnvi ( GL_TEXTURE_ENV, component, constant );
```

Здесь параметр component задает, какую именно часть результата следует умножить, а constant — коэффициент умножения.

Параметр component принимает одно из двух значений: GL_RGB_SCALE_ARB и GL_ALPHA_SCALE_ARB, а constant — 1, 2 или 4.

Выбор в качестве источника данных GL_PREVIOUS_ARB позволяет эффективно использовать данное расширение совместно с мультитекстурированием: операнд, для которого источником данных является GL_PREVIOUS_ARB, берет свои значения из результата операции наложения для предыдущего текстурного блока.

А поскольку режим наложения задается индивидуально для каждого текстурного блока, то можно строить сложные законы наложения нескольких текстур за одну операцию вывода примитива (проход).

На рис. 4.1 изображена общая схема работы с данным расширением: выбирается режим наложения, далее для каждого из используемых операндов

выбирается источник данных и их преобразование. Также задается масштабирующий множитель для результата операции наложения.

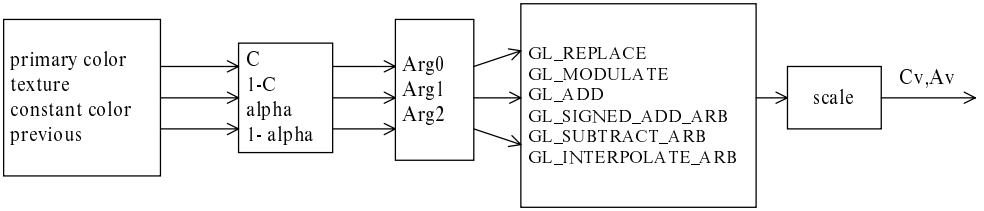


Рис. 4.1. Общая схема работы ARB_texture_env_combine

Рассмотрим несколько простых примеров.

Управление силой отражения при помощи текстуры

Пусть мы хотим смешать две текстуры `tex0` и `tex1`, используя альфа-канал первой текстуры (`tex0`) в качестве веса, т. е. результирующий цвет будет задаваться следующей формулой:

$$C = C_{tex0} \cdot A_{tex0} + C_{tex1} \cdot (1 - A_{tex0}).$$

Примером реализации такого закона может быть наложение карты отражения окружающей среды (*environment mapping*) на поверхность с текстурой, при этом альфа-канал текстуры определяет силу отражения на поверхности в точке.

В данном случае естественным режимом для наложения цветовых компонент является `GL_INTERPOLATE_ARB`, где `Arg0` и `Arg1` выступают RGB-части текстур `tex0` и `tex1`, а в качестве `Arg2` выступает альфа-канал текстуры `tex0`.

Для размещения исходных текстур нам понадобятся два блока (что позволит осуществить вывод всего за один проход). В нулевом текстурном блоке мы разместим текстуру `tex0` и зададим закон ее вывода как `GL_REPLACE`. В результате после ее вывода мы получим саму текстуру как результат предыдущей операции (`GL_PREVIOUS_ARB`).

```
glActiveTextureARB ( GL_TEXTURE0_ARB     );
glBindTexture      ( GL_TEXTURE_2D, tex0 );
```

В первом текстурном блоке мы разместим текстуру `tex1`. Для ее вывода мы используем закон наложения `GL_INTERPOLATE_ARB` для цветовых компонент и `GL_REPLACE` для альфа-канала.

При этом для вывода цветовых компонент в качестве `Arg0` будет выступать результат предыдущей операции (`GL_PREVIOUS_ARB`), в качестве `Arg1` выступает сама текстура (`GL_TEXTURE`), а в качестве `Arg2` следует взять альфа-канал из текстуры `tex0`, т. е. `GL_PREVIOUS_ARB`.

Задание операндов в этом случае выглядит следующим образом:

```
Arg0.rgb = previous.rgb
Arg1.rgb = tex.rgb
Arg2.rgb = rgb ( previous.alpha )
```

Данная схема наложения изображена на рис. 4.2.

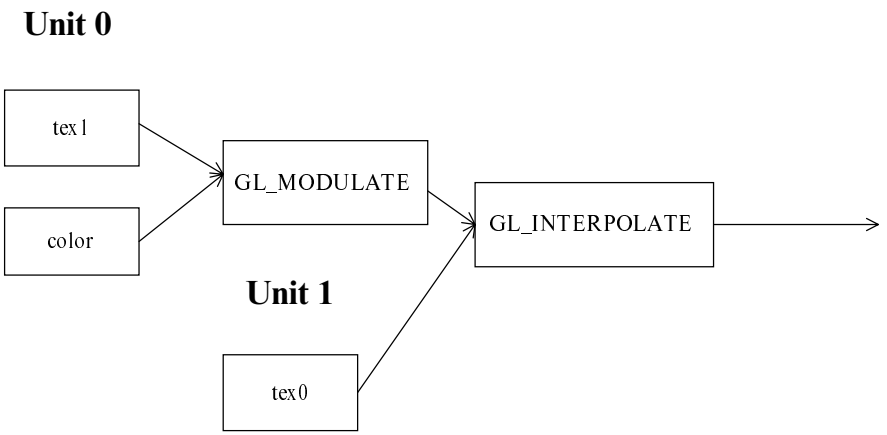


Рис. 4.2. Настройка наложения для управления силой отражения

Таким образом, мы приходим к следующему фрагменту кода:

```
glActiveTextureARB ( GL_TEXTURE1_ARB   );
glBindTexture      ( GL_TEXTURE_2D, t2 );
glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                                     GL_COMBINE_ARB );
glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,
                                     GL_INTERPOLATE_ARB );
glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_ALPHA_ARB, GL_REPLACE );
                                          // now setup Arg0
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,  GL_PREVIOUS_ARB );
glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND0_RGB_ARB, GL_SRC_COLOR    );
                                          // now setup Arg1
```

```
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,  GL_TEXTURE   );
glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND1_RGB_ARB, GL_SRC_COLOR );
                                        // now setup Arg2
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE2_RGB_ARB,  GL_PREVIOUS_ARB );
glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND2_RGB_ARB, GL_SRC_ALPHA    );
```

При этом мультитекстурирование позволяет за один проход осуществить вывод объекта с заданным законом наложения.

Листинг 4.1 содержит полный код для этого примера.

Листинг 4.1. Наложение отражения на главную текстуру с использованием ее альфа-канала

```
//
// Simple demo for GL_texture_env_combine
//
#include    "libExt.h"
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Torus.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
float        angle  = 0;
float        angle2 = 0;
unsigned     mainTex;               // main texture
unsigned     envMap;                // env. map
Torus        torus ( 1, 3, 30, 30 );
void init ()
{
    glClearColor ( 0, 0, 0, 1 );
    glEnable     ( GL_DEPTH_TEST );
}
void display ()
{
    glClear         ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glPushMatrix    ();
```

```
    glTranslatef    ( 7, 7, 7 );
    glRotatef       ( angle,  1, 1, 0 );
    glRotatef       ( angle2, 0, 1, 1 );
                                                    // main texture
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, mainTex );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                             GL_REPLACE );
                // env. map
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glEnable           ( GL_TEXTURE_GEN_S );
    glEnable           ( GL_TEXTURE_GEN_T );
    glTexGeni          ( GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP );
    glTexGeni          ( GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP );
    glBindTexture      ( GL_TEXTURE_2D, envMap );
                // setup combine mode
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                             GL_COMBINE_ARB );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,
                             GL_INTERPOLATE_ARB );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_ALPHA_ARB,
                             GL_REPLACE );
                // setup Arg0
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,
                   GL_PREVIOUS_ARB );
    glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND0_RGB_ARB,
                   GL_SRC_COLOR );
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_ALPHA_ARB,
                   GL_PREVIOUS_ARB );
    glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND0_ALPHA_ARB, GL_SRC_ALPHA );
                             // now setup Arg1
     glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,    GL_TEXTURE   );
     glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND1_RGB_ARB,   GL_SRC_COLOR );
     glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_ALPHA_ARB,  GL_TEXTURE   );
     glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND1_ALPHA_ARB, GL_SRC_ALPHA );
```

```
                            // now setup Arg2 (weight factor)     glTex-
Envi ( GL_TEXTURE_ENV, GL_SOURCE2_RGB_ARB,
                    GL_PREVIOUS_ARB );
       glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND2_RGB_ARB,
                    GL_SRC_ALPHA    );
       glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE2_ALPHA_ARB,
                    GL_PREVIOUS_ARB );
       glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND2_ALPHA_ARB,
                    GL_SRC_ALPHA    );
    torus.draw ();
    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      ( 0.0, 0.0, 0.0,        // eye
                    5.0, 5.0, 5.0,        // center
                    0.0, 1.0, 0.0 );
}
void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
        exit ( 0 );
}
void    animate ()
{
    angle  = 0.04f * glutGet ( GLUT_ELAPSED_TIME );
    angle2 = 0.02f * 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
    int    win = glutCreateWindow (
                          "OpenGL texure_env_combine example 1" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_EXT_texture_env_combine" ) )
    {
        printf ( "EXT_texture_env_combine NOT supported.\n" );
        return 1;
    }
    initExtensions ();
    mainTex = createTexture2D ( true,
                                   "../Textures/block-with-alpha.tga" );
    envMap  = createTexture2D ( true,
                           "../Textures/stars.bmp" );
    glutMainLoop ();
    return 0;
}
```

На рис. 4.3 приводится результат работы данной программы.

Рис. 4.3. Изображение тора с управляемым отражением

Сложение текстур с коэффициентом

Рассмотрим еще один пример. Пусть у нас есть две текстуры `tex0` и `tex1`, и мы хотим задать следующий режим их наложения: `tex0 + tex1 · C`, где `C` — некоторый произвольный коэффициент (например, изменяемый во время анимации).

Тогда мы можем разместить в нулевом текстурном блоке текстуру `tex1` и задать режим наложения как `GL_MODULATE` (т. е. `Arg0*Arg1`), выбрав в качестве операнда `Arg0` цветовую часть самой текстуры, а для `Arg1` — постоянный цвет:

```
Arg0.rgb = texture.rgb
Arg1.rgb = constant-color.rgb
```

В первом текстурном блоке мы разместим текстуру `tex0` и зададим закон наложения как `GL_ADD`, используя в качестве `Arg0` цветовую часть результата предыдущего текстурного блока, а для `Arg1` — цветовое значение текстуры `tex0`:

```
Arg0.rgb = previous.rgb
Arg1.rgb = tex.rgb
```

Этот алгоритм показан на рис. 4.4.

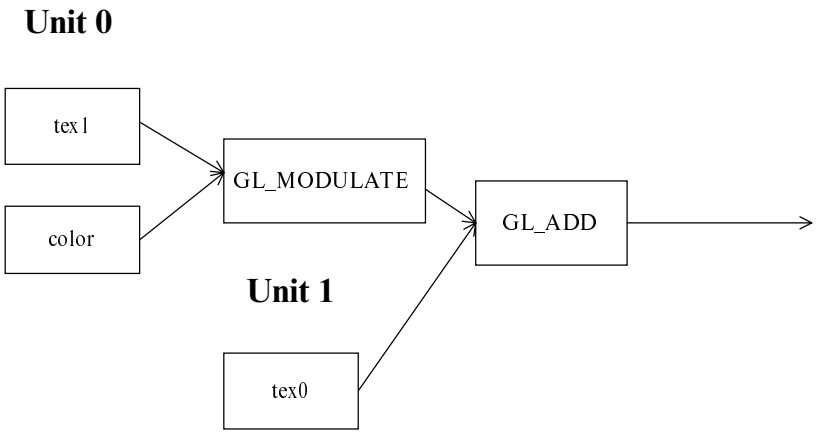


Рис. 4.4. Схема наложения для смешения текстур

Соответствующий способ наложения реализуется листингом 4.2.

Листинг 4.2. Наложение двух текстур по закону `tex0*C+tex1`

```
//
// Simple demo for GL_texture_env_combine
//
#include    "libExt.h"
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Vector4D.h"
#include    "Torus.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
```

```
float        angle  = 0;
float        angle2 = 0;
unsigned     tex0, tex1;
Vector4D     color ( 1, 1, 1, 1 );
Torus        torus ( 1, 3, 30, 30 );
void init ()
{
    glClearColor ( 0, 0, 0, 1 );
    glEnable     ( GL_DEPTH_TEST );
}
void display ()
{
    glClear         ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMatrixMode    ( GL_MODELVIEW );
    glPushMatrix    ();
    glTranslatef    ( 7, 7, 7 );
    glRotatef       ( angle,  1, 1, 0 );
    glRotatef       ( angle2, 0, 1, 1 );
                    // tex0
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, tex0 );
    glMatrixMode       ( GL_TEXTURE );
    glLoadIdentity     ();
    glScalef           ( 12, 12, 12 );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                             GL_COMBINE_ARB );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,
                             GL_MODULATE );
                    // setup Arg0
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,  GL_TEXTURE );
    glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND0_RGB_ARB, GL_SRC_COLOR );
                                    // now setup Arg1
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,
                   GL_CONSTANT_ARB );
    glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND1_RGB_ARB,  GL_SRC_COLOR );
    glTexEnvfv( GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, color );
                    // tex1
```

```
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, tex1 );
    glMatrixMode       ( GL_TEXTURE );
    glLoadIdentity     ();
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                             GL_COMBINE_ARB );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB, GL_ADD );
                    // setup Arg0
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,
                   GL_PREVIOUS_ARB );
    glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND0_RGB_ARB,   GL_SRC_COLOR );
                                    // now setup Arg1
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,    GL_TEXTURE   );
    glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND1_RGB_ARB,   GL_SRC_COLOR );
    torus.draw ();
    glMatrixMode    ( GL_MODELVIEW );
    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      ( 0.0, 0.0, 0.0,        // eye
                    5.0, 5.0, 5.0,        // center
                    0.0, 1.0, 0.0 );
}
void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
        exit ( 0 );
    if ( key == 'p' || key == 'P' )            // screenshot
        saveScreenShot ( "env-combine-2.tga" );
}
```

```
void    animate ()
{
    angle  = 0.04f * glutGet ( GLUT_ELAPSED_TIME );
    angle2 = 0.02f * glutGet ( GLUT_ELAPSED_TIME );

    color.x =
    color.y =
    color.z = 0.5 + 0.4*sin ( 0.005f*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
    int    win = glutCreateWindow (
                             "OpenGL texure_env_combine example 2" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_EXT_texture_env_combine" ) )
    {
        printf ( "EXT_texture_env_combine NOT supported.\n" );
        return 1;
    }
    initExtensions ();
    tex0 = createTexture2D ( true, "../Textures/stars.bmp" );
```

```
    tex1 = createTexture2D ( true, "../Textures/block.bmp" );
    glutMainLoop ();
    return 0;
}
```

Наложение текстуры детализации

Рассмотрим теперь, каким образом можно наложить текстуру детализации (текстуру микрорельефа, микрофасетную текстуру) вместе с основной текстурой на заданную грань за один проход.

Текстура микрорельефа служит для имитации на грани мелких деталей, видимых только с очень близкого расстояния. Непосредственное задание всех этих деталей прямо в основной текстуре привело бы к тому, что она имела бы очень большой размер, что крайне отрицательно сказывается на быстродействии приложения.

Вместо этого обычно используется совершенно другой подход: кроме основной текстуры (*mainTex*) задается дополнительная текстура микрорельефа (*detailTex*).

При этом, поскольку текстура микрорельефа служит для отображения мелких деталей на поверхности объекта видимых только вблизи (в отличие от основной текстуры, содержащей элементы, хорошо видимые на расстоянии), то в качестве текстурных координат для текстуры микрорельефа обычно берут координаты для основной текстуры, умножив их на некоторый масштабирующий коэффициент (обычно от 10 до 30).

На самом деле это умножение можно выполнять не в центральном процессоре, а в графическом ускорителе. Для этого его следует реализовать при помощи матрицы преобразования текстурных координат.

На рис. 4.5 приведен пример типичной текстуры детализации.

Обратите внимание, что среднее значение трех цветовых компонент для всей этой текстуры одинаково и равно 0,5.

Возможны следующие два способа вычисления значений при наложении текстуры микрорельефа на поверхность объекта:

1. $C_{main} \cdot C_{detail}$;
2. $C_{main} + C_{detail} - 0{,}5$.

И первый и второй обеспечивают заметность результатов наложения текстуры микрорельефа только вблизи.

Рассмотрим теперь, каким образом можно реализовать оба этих подхода с помощью расширения ARB_texture_env_combine и мультитекстурирования.

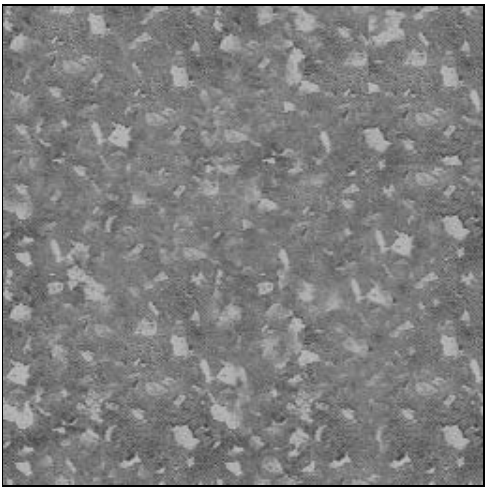


Рис. 4.5. Текстура детализации

Для реализации наложения текстуры микрорельефа по закону $2 \cdot C_{\text{main}} \cdot C_{\text{detail}}$ удобно использовать режим `GL_MODULATE` и умножение результата на 2.

В нулевом текстурном блоке мы разместим главную текстуру и зададим закон наложения `GL_REPLACE`. В первом — текстуру микрорельефа и закон `GL_MODULATE` со следующими назначениями операндов:

```
Arg0.rgb = tex.rgb
Arg1.rgb = previous.rgb
```

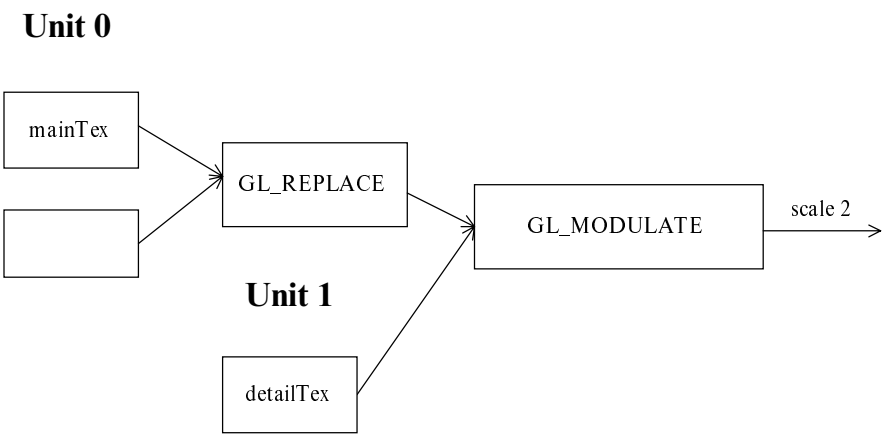


Рис. 4.6. Первый способ наложения текстуры детализации

В этом текстурном блоке мы зададим также матрицу преобразования текстурных координат, обеспечивающую их умножение на масштабирующий коэффициент. Результат операции умножается на 2.

На рис. 4.6 показана реализация этого алгоритма.

В листинге 4.3 приведена процедура настройки OpenGL для вывода грани с этим способом наложения текстур.

Листинг 4.3. Наложение текстуры детализации по первому закону

```
//
// Simple demo for GL_texture_env_combine
//
#include    "libExt.h"
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Vector4D.h"
#include    "Torus.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
float        angle     = 0;
float        angle2    = 0;
bool         useDetail = true;
unsigned     mainTex, detailTex;
Torus        torus ( 1, 3, 30, 30 );
void    setupDetailTexture ()
{
    if ( !useDetail )
    {
        glActiveTextureARB ( GL_TEXTURE1_ARB );
        glDisable          ( GL_TEXTURE_2D );
        return;
    }
                    // detailTex
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, detailTex );
    glMatrixMode       ( GL_TEXTURE );
```

```
    glLoadIdentity     ();
    glScalef           ( 20, 20, 20 );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                         GL_COMBINE_ARB );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,
                         GL_MODULATE );
                    // setup Arg0
    glTexEnvi( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,  GL_PREVIOUS_ARB );
    glTexEnvi( GL_TEXTURE_ENV, GL_OPERAND0_RGB_ARB, GL_SRC_COLOR    );
                                    // now setup Arg1
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,    GL_TEXTURE   );
    glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND1_RGB_ARB,   GL_SRC_COLOR );
                    // set scaling
    glTexEnvf ( GL_TEXTURE_ENV, GL_RGB_SCALE_ARB, 2 );
}
void init ()
{
    glClearColor ( 0, 0, 0, 1 );
    glEnable     ( GL_DEPTH_TEST );
}
void display ()
{
    glClear         ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMatrixMode    ( GL_MODELVIEW );
    glPushMatrix    ();
    glTranslatef    ( 5, 5, 5 );
    glRotatef       ( angle,  1, 1, 0 );
    glRotatef       ( angle2, 0, 1, 1 );
                    // mainTex
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, mainTex );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                         GL_COMBINE_ARB );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,
                         GL_REPLACE );
                    // setup Arg0
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,    GL_TEXTURE );
    glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND0_RGB_ARB,   GL_SRC_COLOR );
```

```
    setupDetailTexture ();
    torus.draw ();
    glMatrixMode    ( GL_MODELVIEW );
    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      ( 0.0, 0.0, 0.0,         // eye
                    5.0, 5.0, 5.0,         // center
                    0.0, 1.0, 0.0 );
}
void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
        exit ( 0 );
    if ( key == 'p' || key == 'P' )            // screenshot
        saveScreenShot ( "env-combine-2.tga" );
    if ( key == '1' )
        useDetail = false;
     if ( key == '2' )
        useDetail = true;
}
void    animate ()
{
    angle  = 0.04f * glutGet ( GLUT_ELAPSED_TIME );
    angle2 = 0.02f * 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
    int    win = glutCreateWindow (
                         "OpenGL texure_env_combine example 4" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_EXT_texture_env_combine" ) )
    {
        printf ( "EXT_texture_env_combine NOT supported.\n" );
        return 1;
    }
    initExtensions ();
    mainTex   = createTexture2D ( true, "../Textures/oak.bmp" );
    detailTex = createTexture2D ( true, "../Textures/detail.tga" );
    glutMainLoop ();
    return 0;
}
```

Для лучшей заметности получаемого эффекта в этой программе при помощи клавиш <1> и <2> можно выключать и включать наложение текстуры детализации.

Настройка текстуры детализации производится в функции `setupDetailTexture`. Обратите внимание, как при помощи матрицы преобразования текстурных координат задается масштаб для наложения текстуры детализации на объект.

```
glMatrixMode       ( GL_TEXTURE );
glLoadIdentity     ();
glScalef           ( 20, 20, 20 );
```

Рассмотрим теперь реализацию второго закона наложения.

Для этого, как и ранее, в нулевом текстурном блоке мы разместим главную текстуру с законом наложения `GL_REPLACE`. В первом — текстуру детализации, задав закон наложения как `GL_SIGNED_ADD_ARB` и назначив операнды следующим образом:

```
Arg0.rgb = tex.rgb
Arg1.rgb = previous.rgb
```

На рис. 4.7 показан алгоритм вычислений, а в листинге 4.4 — код, реализующий наложение по этому закону.

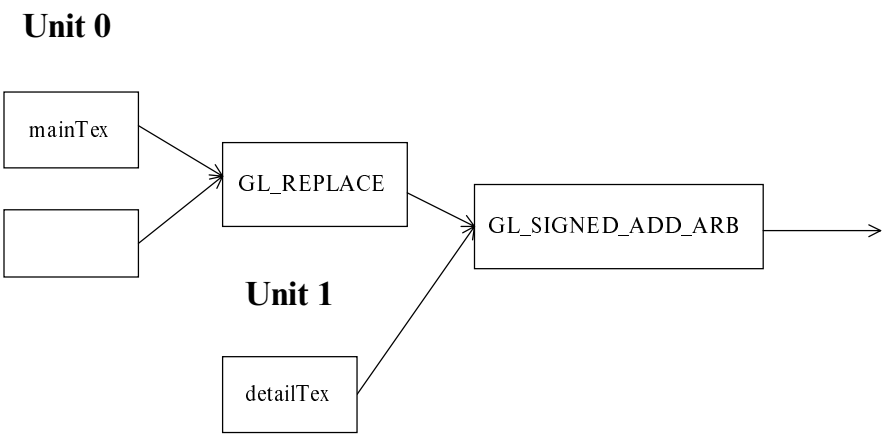


Рис. 4.7. Второй способ наложения текстуры детализации

Листинг 4.4. Наложение текстуры детализации по второму способу

```
//
// Simple demo for GL_texture_env_combine
//
#include    "libExt.h"
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Vector4D.h"
#include    "Torus.h"
#include    <glut.h>
#include    <stdio.h>
```

```
#include    <stdlib.h>
float         angle     = 0;
float         angle2    = 0;
bool        useDetail = true;
unsigned     mainTex, detailTex;
Torus        torus ( 1, 3, 30, 30 );
void    setupDetailTexture ()
{
    if ( !useDetail )
    {
        glActiveTextureARB ( GL_TEXTURE1_ARB );
        glDisable          ( GL_TEXTURE_2D );
        return;
    }
                        // detailTex
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, detailTex );
    glMatrixMode       ( GL_TEXTURE );
    glLoadIdentity     ();
    glScalef           ( 20, 20, 20 );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                             GL_COMBINE_ARB );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,
                             GL_ADD_SIGNED_ARB );
                                          // setup Arg0
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,
                   GL_PREVIOUS_ARB );
    glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND0_RGB_ARB,   GL_SRC_COLOR );

                                        // now setup Arg1
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,    GL_TEXTURE   );
    glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND1_RGB_ARB,   GL_SRC_COLOR );
}
void init ()
{
    glClearColor ( 0, 0, 0, 1 );
    glEnable     ( GL_DEPTH_TEST );
}
```

```
void display ()
{
    glClear         ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMatrixMode    ( GL_MODELVIEW );
    glPushMatrix    ();
    glTranslatef    ( 5, 5, 5 );
    glRotatef       ( angle,  1, 1, 0 );
    glRotatef       ( angle2, 0, 1, 1 );
                       // mainTex
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, mainTex );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                            GL_COMBINE_ARB );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,
                            GL_REPLACE );
                        // setup Arg0
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,    GL_TEXTURE );
    glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND0_RGB_ARB,   GL_SRC_COLOR );
    setupDetailTexture ();
    torus.draw ();
    glMatrixMode    ( GL_MODELVIEW );
    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      ( 0.0, 0.0, 0.0,        // eye
                    5.0, 5.0, 5.0,        // center
                    0.0, 1.0, 0.0 );
}
void key ( unsigned char key, int x, int y )
{
```

```
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
        exit ( 0 );
    if ( key == 'p' || key == 'P' )            // screenshot
        saveScreenShot ( "env-combine-2.tga" );
    if ( key == '1' )
        useDetail = false;
    if ( key == '2' )
        useDetail = true;
}
void    animate ()
{
    angle  = 0.04f * glutGet ( GLUT_ELAPSED_TIME );
    angle2 = 0.02f * 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
    int    win = glutCreateWindow (
                         "OpenGL texure_env_combine example 4" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_EXT_texture_env_combine" ) )
    {
        printf ( "EXT_texture_env_combine NOT supported.\n" );
        return 1;
```

```
    }
    initExtensions ();
    mainTex   = createTexture2D ( true, "../Textures/oak.bmp" );
    detailTex = createTexture2D ( true, "../Textures/detail.tga" );
    glutMainLoop ();
    return 0;
}
```

Если у нас есть более двух текстурных блоков, то можно за один проход вывести основную текстуру (*mainTex*), наложить на нее карту освещенности (*lightTex*) и текстуру микрорельефа (*detailTex*).

Общая схема подобного наложения приведена на рис. 4.8.

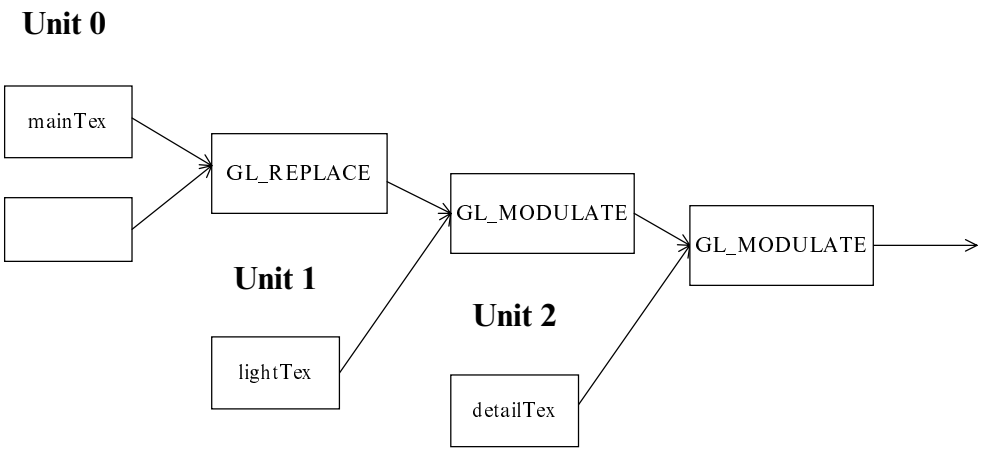


Рис. 4.8. Наложение основной текстуры, карты освещенности и текстуры микрорельефа за один проход

Как видно из рисунка, в нулевом текстурном блоке мы, как и ранее, размещаем главную текстуру и задаем закон наложения `GL_REPLACE`. В первом — карту освещенности, задаем закон наложения `GL_MODULATE` и следующие значения операндов:

```
Arg0.rgb = tex.rgb
Arg1.rgb = previous.rgb
```

Во втором текстурном блоке мы размещаем текстуру микрорельефа, задаем закон наложения `GL_MODULATE` и следующие значения операндов:

```
Arg0.rgb = tex.rgb
Arg1.rgb = previous.rgb
```

Результат операции во втором текстурном блоке мы умножаем на 2.

Для вывода небольших объектов можно по аналогии с только что рассмотренным примером вместо карты освещенности (которая для игроков и объектов в игре обычно не применяется) выбрать карту отражения окружающей среды. При этом альфа-канал основной текстуры, как и ранее, управляет наложением карты отражения, а на получившийся результат накладывается карта детализации.

В листинге 4.5 приводится такая процедура наложения.

Листинг 4.5. Одновременное наложение основной текстуры, карты отражения и текстуры детализации

```
void display ()
{
    glClear         ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMatrixMode    ( GL_MODELVIEW );
    glPushMatrix    ();
    glTranslatef    ( 4.5, 4.5, 4.5 );
    glRotatef       ( angle,  1, 1, 0 );
    glRotatef       ( angle2, 0, 1, 1 );
                    // main texture
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, mainTex );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                             GL_REPLACE );
                    // env. map
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glEnable           ( GL_TEXTURE_GEN_S );
    glEnable           ( GL_TEXTURE_GEN_T );
    glTexGeni          ( GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP );
    glTexGeni          ( GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP );
    glBindTexture      ( GL_TEXTURE_2D, envMap );
                    // setup combine mode
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                             GL_COMBINE_ARB );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,
                             GL_INTERPOLATE_ARB );
```

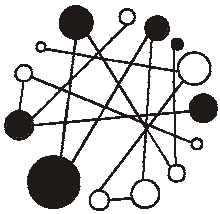
```
glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_ALPHA_ARB,
                       GL_REPLACE );
                // setup Arg0
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,
              GL_PREVIOUS_ARB );
glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND0_RGB_ARB,
              GL_SRC_COLOR    );
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_ALPHA_ARB,
              GL_PREVIOUS_ARB );
glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND0_ALPHA_ARB,
              GL_SRC_ALPHA    );
                                // now setup Arg1
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,    GL_TEXTURE   );
glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND1_RGB_ARB,   GL_SRC_COLOR );
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_ALPHA_ARB,  GL_TEXTURE   );
glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND1_ALPHA_ARB, GL_SRC_ALPHA );
                                // now setup Arg2 (weight factor)
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE2_RGB_ARB,
              GL_PREVIOUS_ARB );
glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND2_RGB_ARB,   GL_SRC_ALPHA );
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE2_ALPHA_ARB,
              GL_PREVIOUS_ARB );
glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND2_ALPHA_ARB, GL_SRC_ALPHA );
glActiveTextureARB ( GL_TEXTURE2_ARB );
glEnable           ( GL_TEXTURE_2D );
glBindTexture      ( GL_TEXTURE_2D, detailMap );
glMatrixMode       ( GL_TEXTURE );
glLoadIdentity     ();
glScalef           ( 20, 20, 20 );
glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                       GL_COMBINE_ARB );
glTexEnvi          ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,
                       GL_MODULATE );
                // setup Arg0
glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,
              GL_PREVIOUS_ARB );
glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND0_RGB_ARB,
              GL_SRC_COLOR    );
                                   // now setup Arg1
```

```
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,    GL_TEXTURE   );
    glTexEnvi ( GL_TEXTURE_ENV, GL_OPERAND1_RGB_ARB,   GL_SRC_COLOR );
                    // set scaling
    glTexEnvf ( GL_TEXTURE_ENV, GL_RGB_SCALE_ARB, 2 );
    torus.draw ();
    glMatrixMode    ( GL_MODELVIEW );
    glPopMatrix     ();
    glutSwapBuffers ();
}
```

Отличие расширения EXT_texture_env_combine от ARB_texture_env_combine заключается в отсутствии в первом режима `GL_SUBSTRACT_ARB`.

Для операнда 2 возможен только режим `SRC_ALPHA`, а для операнда 1 — `SRC_ALPHA` и `ONE_MINUS_SRC_ALHA`.

Глава 5



Кубические текстурные карты и расширение ARB_texture_cube_map

Стандартный OpenGL поддерживает всего два типа текстур: одномерные (1D) и двумерные (2D). Начиная с версии 1.2, в OpenGL добавлена поддержка трехмерных (3D) текстур, рассматриваемых в *главе 6*.

Расширение GL_ARB_texture_cube_map (так же как и GL_EXT_texture_cube_map) добавляет еще один тип текстур — кубические (*cube maps*). Таким образом, становятся доступными 4 типа текстур (*texture targets*): 1D, 2D, 3D и *cube maps*.

С каждым из этих типов связана своя константа, передаваемая в такие функции, как `glBindTexture`, `glTexParameter` и `glEnable`. Для кубических текстур такой константой является `GL_TEXTURE_CUBE_MAP_ARB` (или равная ей по значению `GL_TEXTURE_CUBE_MAP_EXT`).

Для проверки поддержки работы с кубическими текстурами следует использовать команду

```
isExtensionSupported ( "GL_ARB_texture_cube_map" )
```

Но в отличие от традиционных (одно-, дву- и трехмерных) текстур, каждая кубическая текстура представляет собой набор из шести текстур одинаково размера, которые как бы наложены на грани куба (рис. 5.1).

Когда возникает необходимость работы с одной из этих шести текстур (например, для загрузки в память графического ускорителя), следует указывать константы, однозначно идентифицирующие соответствующую текстуру (грань куба, на которую она должна быть наклеена). Для этого существует шесть констант:

```
GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB
GL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB
GL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB
GL_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB
```

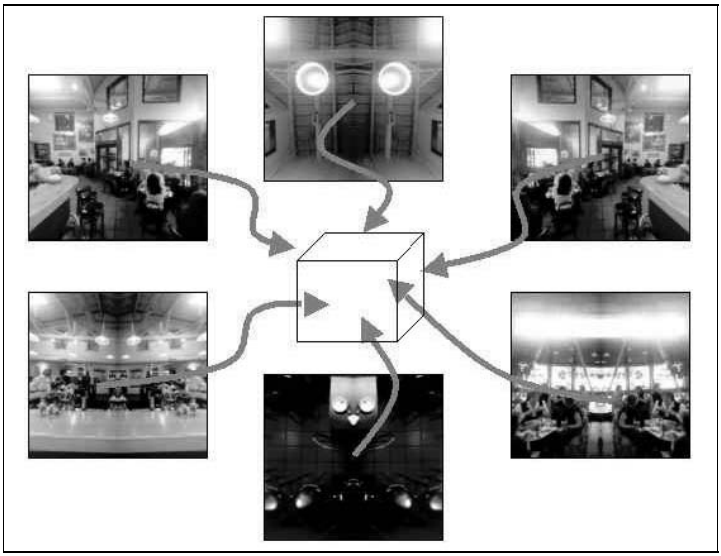


Рис. 5.1. Наложение кубической текстуры на грани куба

Каждая из них соответствует одной из граней куба, расположенной в положительном или отрицательном направлении соответствующей координатной оси (перпендикулярной данной грани). Впервые подобная схема появилась в системе *RenderMan*.

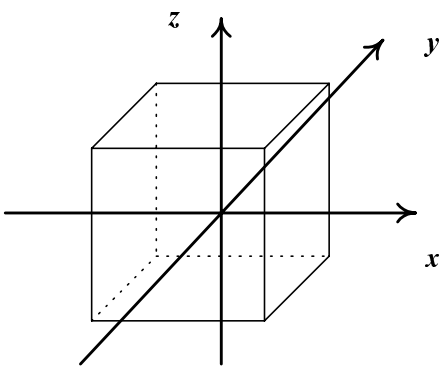


Рис. 5.2. Ориентация граней куба относительно координатных осей

Значения всех этих констант идут последовательно одно за другим, поэтому справедливы следующие соотношения:

```
GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB + 1 =
GL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB
```

```
GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB + 2 =
GL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB
```

Для функций, работающих с конкретными текстурами (например, `glTexImage2D`), следует использовать именно эти константы.

В листинге 5.1 приводится функция из библиотеки libTexture, осуществляющая загрузку кубической текстуры в память графического ускорителя. Обратите внимание, что вместо одного имени файла в эту функцию передается список имен (по одному для каждой грани).

Листинг 5.1. Загрузка кубической карты

```
unsigned    createCubeMap ( bool mipmap, const char * maps [] )
{
    unsigned    textureId;
    glGenTextures   ( 1, &textureId );
    glBindTexture   ( GL_TEXTURE_CUBE_MAP_ARB, textureId );
    glPixelStorei   ( GL_UNPACK_ALIGNMENT, 1 );    // set 1-byte
                                                    //alignment
    glTexParameterf ( GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_S,
                      GL_CLAMP_TO_EDGE );
    glTexParameterf ( GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_T,
                      GL_CLAMP_TO_EDGE );
    loadTexture ( GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB, mipmap, maps [0] );
    loadTexture ( GL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, mipmap, maps [1] );
    loadTexture ( GL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, mipmap, maps [2] );
    loadTexture ( GL_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, mipmap, maps [3] );
    loadTexture ( GL_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, mipmap, maps [4] );
    loadTexture ( GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB, mipmap, maps [5] );
    glEnable ( GL_TEXTURE_CUBE_MAP_ARB );
    if ( mipmap )
    {
    glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB,
                            GL_TEXTURE_MIN_FILTER,
                            GL_LINEAR_MIPMAP_LINEAR );
    glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB,
                            GL_TEXTURE_MAG_FILTER,
                            GL_LINEAR );
    }
    else
```

```
    {
    glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB,
                            GL_TEXTURE_MIN_FILTER,
                            GL_LINEAR );
    glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB,
                            GL_TEXTURE_MAG_FILTER,
                            GL_LINEAR );
    }
    return textureId;
}
```

В данном примере входными параметрами функции является список из шести имен текстур и признак пирамидального фильтрования (*mipmapping*).

Важно отметить, что в качестве режима отсечения текстурных координат выбирается `GL_GLAMP_TO_EDGE`, что в ряде случаев необходимо для корректной работы программ.

Для работы с кубическими текстурами служат три текстурные координаты (*s*, *t*, *r*). Фактически они задают направление прямой, выходящей из начала координат. Тогда каждая точка на текстуре определяется как точка пересечения этой прямой с единичным кубом, описанным вокруг начала координат (с ребрами, параллельными координатным осям).

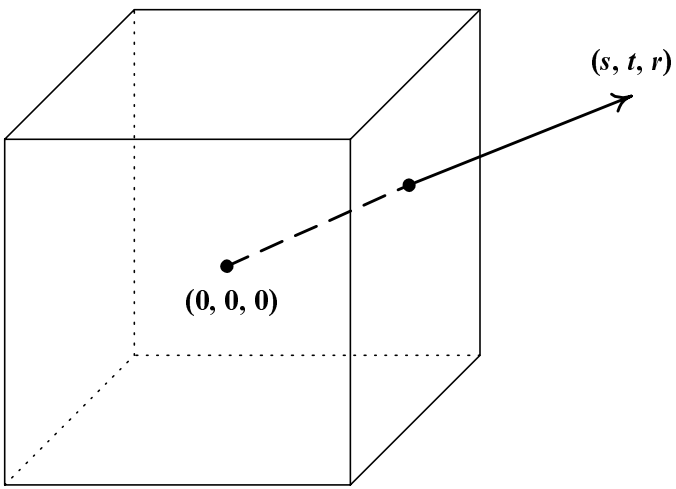


Рис. 5.3. Определение точки на кубе, соответствующей текстурным координатам (*s*, *t*, *r*)

Легко видно, что пересекаемая грань будет определяться текстурной координатой, имеющей наибольшее по модулю значение. Так, если это значение имеет координата *r*, то, в зависимости от знака *r* будет выбрана либо `GL_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB`, либо `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB`.

При этом текстурные координаты (*u*, *v*) для обращения к текстуре соответствующей грани, вычисляются по следующим формулам:

$$u = (s/|r| + 1)/2;$$
$$v = (t/|r| + 1)/2.$$

Это означает, что текстурные координаты для кубической карты определены с точностью до положительного множителя. Так, координатам (2*s*, 2*t*, 2*r*) и (10*s*, 10*t*, 10*r*) соответствует точно та же точка на кубической карте, что и координатам (*s*, *t*, *r*).

Таким образом, кубическая карта задает функцию от направления в трехмерном пространстве.

Текстурные координаты можно задать явно:

```
glTexCoord3f  ( s, t, r );
glTexCoord3fv ( tex );
```

Автоматическую генерацию текстурных координат можно задать с помощью следующего фрагмента кода:

```
glEnable  ( GL_TEXTURE_GEN_S );
glEnable  ( GL_TEXTURE_GEN_T );
glEnable  ( GL_TEXTURE_GEN_R );
glTexGeni ( GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_ARB );
glTexGeni ( GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_ARB );
glTexGeni ( GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_ARB );
```

Кроме режима `GL_REFLECTION_MAP_ARB` существует еще один — `GL_NORMAL_MAP_ARB`, при котором текстурные координаты соответствуют интерполяции вектора нормали вдоль граней.

Задать текстурные координаты, соответствующие нормали, можно с помощью следующего фрагмента кода:

```
glEnable  ( GL_TEXTURE_GEN_S );
glEnable  ( GL_TEXTURE_GEN_T );
glEnable  ( GL_TEXTURE_GEN_R );
glTexGeni ( GL_S, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP_ARB );
glTexGeni ( GL_T, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP_ARB );
glTexGeni ( GL_R, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP_ARB );
```

Обратите внимание, что и в том и в другом случаях необходимо задание нормалей для вершин.

Полезным может оказаться также режим GL_EYE_LINEAR, который позволяет получать ненормированный вектор от наблюдателя.

Для предотвращения появления артефактов в случаях, когда точка на поверхности куба, соответствующая текстурным координатам, подходит слишком близко к какому-либо из его ребер, вместо GL_REPEAT (или GL_CLAMP) следует применить режим GL_CLAMP_TO_EDGE () при задании отсечения текстурных координат (*texture wrapping*).

Задание этого режима осуществляется следующими командами:

```
glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_S,
                  GL_CLAMP_TO_EDGE );
glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_T,
                  GL_CLAMP_TO_EDGE );
```

Обратите внимание, что данный режим устанавливается только для *s* и *t*, поскольку отсечение координат происходит в пределах одной из граней куба.

Так как в OpenGL по умолчанию устанавливается режим GL_REPEAT, то при работе с кубическими картами для всех карт следует явно выставить GL_CLAMP_TO_EDGE.

Для использования кубических текстур их (подобно всем остальным) следует сначала разрешить при помощи команды

```
glEnable  ( GL_TEXTURE_CUBE_MAP_ARB );
```

Важно помнить, что в OpenGL кубические текстуры имеют более высокий приоритет по сравнению с обычными, поэтому если одновременно в одном и том же текстурном блоке разрешить и кубическую, и обычную (например, двумерную) текстуры, то будет выбрана именно кубическая.

В листинге 5.2 приводится исходный код примера, демонстрирующего кубические текстуры с автоматической генерацией текстурных координат. В данном примере при помощи клавиш <r> и <n> можно переключаться между режимами автоматической генерации текстурных координат, а цифровые клавиши (так же как и клавиши <+> и <–>) управляют фигурой.

Листинг 5.2. Пример использования кубической текстурной карты

```
//
// Sample to OpenGL cubic maps
//
#include    "libExt.h"
#include    "libTexture.h"
```

```
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
float     angle    = 0;
float     angle2   = 0;
int     reflMode = GL_REFLECTION_MAP_ARB;
int     object   = 0;
unsigned    texture;
void init ()
{
    glClearColor ( 0, 0, 0, 1 );
    glEnable     ( GL_DEPTH_TEST );

    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 );
       glTexGeni ( GL_S, GL_TEXTURE_GEN_MODE, reflMode );
       glTexGeni ( GL_T, GL_TEXTURE_GEN_MODE, reflMode );
glTexGeni ( GL_R, GL_TEXTURE_GEN_MODE, reflMode );
glEnable  ( GL_TEXTURE_CUBE_MAP_ARB );
glBindTexture ( GL_TEXTURE_CUBE_MAP_ARB, texture );
glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_S,
                        GL_CLAMP );
glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_T,
                        GL_CLAMP );
       glEnable  ( GL_TEXTURE_GEN_S );
       glEnable  ( GL_TEXTURE_GEN_T );
       glEnable  ( GL_TEXTURE_GEN_R );
glPushMatrix    ();
glTranslatef    ( 2, 2, 2 );
glRotatef       ( angle,  1, 1, 0 );
glRotatef       ( angle2, 0, 1, 1 );
if ( object == 0 )
    glutSolidTeapot ( 1 );
```

```
else
if ( object == 1 )
    glutSolidSphere ( 1, 30, 30 );
else
if ( object == 2 )
    glutSolidCube ( 1 );
else
if ( object == 3 )
    glutSolidCone ( 0.7, 1.2, 30, 30 );
else
if ( object == 4 )
    glutSolidTorus ( 0.5, 1, 40, 40 );
else
if ( object == 5 )
    glutSolidDodecahedron ();
else
if ( object == 6 )
    glutSolidOctahedron ();
else
if ( object == 7 )
    glutSolidTetrahedron ();
else
if ( object == 8 )
    glutSolidIcosahedron ();
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      ( 0.0, 0.0, 0.0,        // eye
                    5.0, 5.0, 5.0,        // center
                    0.0, 1.0, 0.0 );
}
```

```
void key ( unsigned char key, int x, int y )
{
if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
    exit ( 0 );
if ( key == 'n' || key == 'N' )
    reflMode = GL_NORMAL_MAP_ARB;
else
if ( key == 'r' || key == 'R' )
    reflMode = GL_REFLECTION_MAP_ARB;
else
if ( key == '0' )
    object = 0;
else
if ( key == '1' )
    object = 1;
else
if ( key == '2' )
    object = 2;
else
if ( key == '3' )
    object = 3;
else
if ( key == '4' )
    object = 4;
else
if ( key == '5' )
    object = 5;
else
if ( key == '6' )
    object = 6;
else
if ( key == '7' )
    object = 7;
else
if ( key == '8' )
    object = 8;
else
if ( key == '+' )
```

```
{
    if ( ++object > 8 )
        object = 0;
}
else
if ( key == '-' )
{
    if ( --object < 0 )
        object = 8;
}
}
void    animate ()
{
angle  = 0.04f * glutGet ( GLUT_ELAPSED_TIME );
angle2 = 0.01f * 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
    int    win = glutCreateWindow ( "OpenGL cubemap example" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    if ( !isExtensionSupported ( "GL_ARB_texture_cube_map" ) )
    {
        printf ( "ARB_texture_cube_map NOT supported.\n" );
        return 1;
    }
    init           ();
    initExtensions ();
    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",
    };
    texture = createCubeMap ( true, faces );
    glutMainLoop ();
    return 0;
}
```

Кубическая текстура позволяет в простой и удобной форме задавать функции от направления. Каждому направлению естественным образом соответствует точка на кубе, поэтому достаточно просто задать значения функции в виде текстур на гранях данного куба.

Еще одним возможным применением кубической текстуры (мы с ним столкнемся позже, когда будем рассматривать попиксельное освещение) является нормализация векторов.

Единичному трехмерному вектору можно легко сопоставить значение компонент RGB. Но при интерполяции и других простых операциях с подобными значениями (векторами) у нас могут получиться ненормированные результаты (непригодные для уравнения освещенности). Кубическая текстура предоставляет простой способ нормализации.

Другим примером является задание источника света в виде проектора, когда сила света (и его цвет), падающего в точку, зависит от направления из этой точки на источник (рис. 5.4). Кубическая текстура в этом случае позволяет в легкой и удобной форме задать подобный проектор.

Рассмотрим, каким образом можно реализовать источник света (проектор) при помощи кубической карты.

Для проектора освещенность зависит от направления на источник, т. е. является функцией направления и поэтому может быть представлена при помощи кубической текстурной карты.

Пусть источник света расположен в точке *light*. Тогда для произвольной вершины v_i в качестве текстурных координат для кубической карты, задающей распределение освещенности, возьмем значения согласно формуле

$$(s, t, r) = v_i - light. \tag{5.1}$$

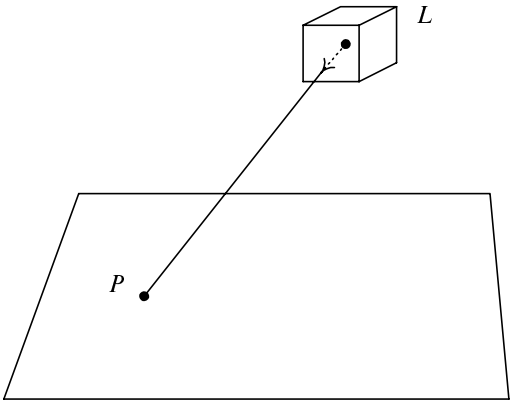


Рис. 5.4. Задание проектора при помощи кубической текстуры

Затем каждой точке произвольной грани сопоставим текстурные координаты на карте освещенности, т. е. зададим падающий на нее свет.

Исходя из этого, мы можем реализовать освещение таким проектором следующим образом.

1. В нулевом текстурном блоке мы разместим обычную двумерную текстуру объекта и зададим закон наложения `GL_REPLACE`.
2. В первом текстурном блоке мы разместим кубическую карту, задающую распределение освещенности. В качестве закона наложения выберем `GL_MODULATE`, текстурные координаты будем задавать по формуле (5.1).

Если мы при этом хотим задавать не только положение источника света в пространстве, но и его ориентацию, то это легко можно сделать при помощи матрицы преобразования текстурных координат.

Ориентация объекта в пространстве может быть задана при помощи ортогональной матрицы (или трех углов Эйлера, или кватерниона). Тогда, если задать в качестве матрицы преобразования текстурных координат транспонированную матрицу ориентации (для ортогональных матриц транспонированная совпадает с обратной), то этого будет достаточно для моделирования ориентации источника света.

В листинге 5.3 приводится соответствующий исходный код.

Листинг 5.3. Реализация прожектора при помощи кубической текстурной карты

```
//
// Sample to OpenGL cubic maps - using a lamp with color plate (mask)
// attached to it.
```

```
//
#include    "libExt.h"
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
Vector3D    eye   ( 2, 2, 2 );        // camera position
Vector3D    light ( 0.5, 0, 1 );      // light position
float         angle    = 0;
float         angle2   = 0;
int        reflMode = GL_REFLECTION_MAP_ARB;
int        object   = 0;
unsigned     texture;
unsigned     lampMap;
Vector3D    v  [12];
Vector2D    t0 [12];
Vector3D    t1 [12];

void init ()
{
    glClearColor ( 0, 0, 0, 1 );
    glEnable     ( GL_DEPTH_TEST );
    glHint ( GL_POLYGON_SMOOTH_HINT,         GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}
void display ()
{
    int    i;

    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMatrixMode       ( GL_MODELVIEW );
                       // setup normal texture
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glDisable          ( GL_TEXTURE_CUBE_MAP_ARB );
    glBindTexture      ( GL_TEXTURE_2D, texture );
```

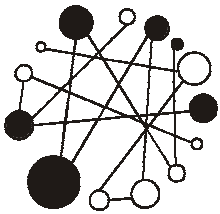
```
    glTexEnvi           ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                              GL_REPLACE );
                      // setup lamp texture
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnable            ( GL_TEXTURE_CUBE_MAP_ARB );
    glBindTexture       ( GL_TEXTURE_CUBE_MAP_ARB, lampMap );
    glTexEnvi           ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                              GL_MODULATE );
                      // apply rotation to lamp
    glMatrixMode        ( GL_TEXTURE );
    glLoadIdentity      ();
    glRotatef           ( angle,  1, 1, 0 );
    glRotatef           ( angle2, 0, 1, 1 );
                      // compute texture coords into lamp
    for ( i = 0; i < 12; i++ )
        t1 [i] = v [i] - light;
                      // now draw
    glBegin ( GL_QUADS );
    for ( i = 0; i < 12; i++ )
    {
        glMultiTexCoord2fv ( GL_TEXTURE0_ARB, t0 [i] );
        glMultiTexCoord3fv ( GL_TEXTURE1_ARB, t1 [i] );
          glVertex3fv        ( v [i]                   );
    }
    glEnd   ();
    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 key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
        exit ( 0 );
}
void    animate ()
{
    angle  = 0.04f * glutGet ( GLUT_ELAPSED_TIME );
    angle2 = 0.01f * 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
    int    win = glutCreateWindow("OpenGL cubemap projector example");
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key );
    glutIdleFunc     ( animate );
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_ARB_texture_cube_map" ) )
    {
        printf ( "ARB_texture_cube_map NOT supported.\n" );
        return 1;
    }
    init           ();
    initExtensions ();
    const char * faces [6] =
```

```
    {
        "../Textures/floor.bmp",
        "../Textures/floor.bmp",
        "../Textures/floor.bmp",
        "../Textures/floor.bmp",
        "../Textures/floor.bmp",
        "../Textures/floor.bmp"
    };
    lampMap = createCubeMap   ( true, faces );
    texture = createTexture2D ( true, "../Textures/block.bmp" );
       v [0 ] = Vector3D ( -1, -1, 0 );
        v [1 ] = Vector3D (  1, -1, 0 );
        v [2 ] = Vector3D (  1,  1, 0 );
        v [3 ] = Vector3D ( -1,  1, 0 );
        v [4 ] = Vector3D ( -1, -1, 0 );
        v [5 ] = Vector3D (  1, -1, 0 );
        v [6 ] = Vector3D (  1, -1, 2 );
        v [7 ] = Vector3D ( -1, -1, 2 );
        v [8 ] = Vector3D ( -1, -1, 0 );
        v [9 ] = Vector3D ( -1,  1, 0 );
        v [10] = Vector3D ( -1,  1, 2 );
        v [11] = Vector3D ( -1, -1, 2 );
                        // setup texture coordinates for bump map
        t0 [0 ] = Vector2D ( 0, 0 );
        t0 [1 ] = Vector2D ( 1, 0 );
        t0 [2 ] = Vector2D ( 1, 1 );
        t0 [3 ] = Vector2D ( 0, 1 );
        t0 [4 ] = Vector2D ( 0, 0 );
        t0 [5 ] = Vector2D ( 1, 0 );
        t0 [6 ] = Vector2D ( 1, 1 );
        t0 [7 ] = Vector2D ( 0, 1 );
        t0 [8 ] = Vector2D ( 0, 0 );
        t0 [9 ] = Vector2D ( 1, 0 );
        t0 [10] = Vector2D ( 1, 1 );
        t0 [11] = Vector2D ( 0, 1 );
    glutMainLoop ();
    return 0;
}
```

Глава 6



Трехмерные (3D) текстуры и расширение EXT_texture3D

Текстура представляет собой удобный способ задания функции нескольких переменных. Функция одной переменной легко задается при помощи одномерной (1D) текстуры, двух — двумерной, функция направления в трехмерном пространстве — при помощи кубической текстурной карты.

При этом сложность задаваемой функции не играет практически никакой роли. После того как функция была задана при помощи текстуры, работа с ней происходит одинаково легко вне зависимости от сложности исходной функции.

Механизм линейного фильтрования OpenGL фактически является способом полилинейного (в зависимости от типа текстуры) интерполирования значений текселов на все точки единичного отрезка (квадрата, сферы).

Данное обстоятельство активно используется при написании сложных шейдеров (*shader*) — процедур закрашивания граней. За счет этого удается в ряде случаев обеспечить довольно сложные законы рендеринга, реализация которых путем прямого вычисления соответствующих функций либо вообще невозможна, либо слишком громоздка с вычислительной точки зрения.

Однако в ряде случаев (с некоторыми из них мы столкнемся позже) возникает необходимость определения функции трех независимых переменных. Это естественным образом приводит к понятию трехмерной (3D) текстуры.

Изначально поддержка трехмерных текстур в OpenGL была реализована при помощи расширения EXT_texture3D, потом, начиная с версии 1.2, она вошла в стандарт OpenGL.

Расширение EXT_texture3D позволяет приложению создавать и использовать в OpenGL трехмерные текстуры, которые представляются трехмерными матрицами текселов размером *width*×*height*×*depth* (рис. 6.1).

Расширение EXT_texture3D вводит новую функцию `glTexImage3DEXT`, позволяющую задавать трехмерные текстуры, а также ряд констант, необходимых для работы с ними.

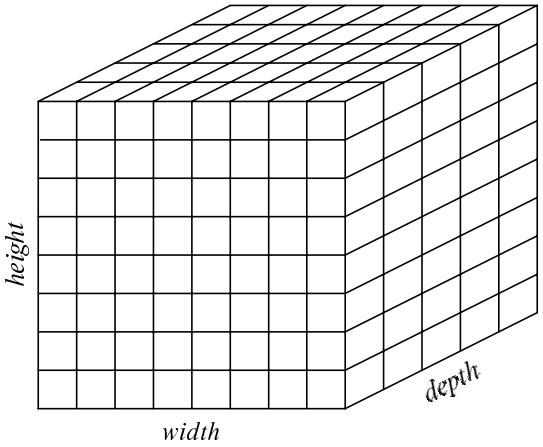


Рис. 6.1. Трехмерная текстура

Самой важной из вводимых констант является `GL_TEXTURE_3D_EXT`, выступающая в качестве типа текстуры в обращении к функциям `glEnable`, `glDisable`, `glIsEnabled`, `glGetTexImage` и `glGetTexParameter`.

Также вводится константа `GL_MAX_3D_TEXTURE_SIZE_EXT`, позволяющая получить максимальный размер трехмерной текстуры. Для этого служит следующий фрагмент кода:

```
int    maxSize;
glGetIntegerv ( GL_MAX_3D_TEXTURE_SIZE_EXT, &maxSize );
```

Поскольку при обращении к трехмерной текстуре необходимы все три текстурные координаты (s, t, r), то вводится константа `GL_TEXTURE_WRAP_R_EXT`, позволяющая задать закон приведения координаты r в отрезок [0, 1].

Непосредственное задание трехмерной текстуры осуществляется при помощи функции `glTexImage3DEXT`, пример вызова которой выглядит следующим образом:

```
void glTexImage3DEXT ( GLenum target, GLint level, GLenum internalformat,
                       GLsizei width, GLsizei height, GLsizei depth,
                       GLint border, GLenum format, GLenum type,
                       const GLvoid * pixels );
```

Здесь в качестве параметра `target` выступает константа `GL_TEXTURE_3D_EXT`, параметры `width`, `height` и `depth` задают размер задаваемой текстуры в текселях по каждому измерению. Параметры `level`, `internalformat`, `border`,

`format` и `type` полностью аналогичны соответствующим параметрам функций `glTexImage1D` и `glTexImage2D`.

Для правильного задания трехмерной текстуры при помощи функции `glTexImage3DEXT` текселы, расположенные в буфере по адресу `pixels`, должны быть организованы как набор из `depth` прямоугольных текстур размером `width` на `height` текселов (рис. 6.2).

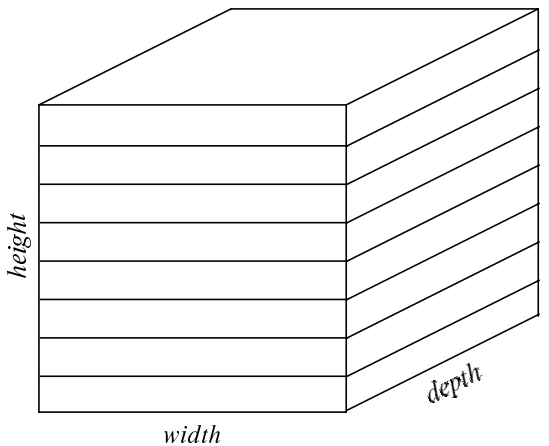


Рис. 6.2. Организация трехмерной текстуры в памяти

При помощи параметров `GL_UNPACK_ROW_LENGTH` и `GL_UNPACK_ALIGNMENT` можно задать расположение текселов для каждой из этих прямоугольных текстур.

Для трехмерной текстуры допустимо также пирамидальное фильтрование (*mipmapping*), однако для этого потребуется явно задать все необходимые промежуточные уровни аналогично тому, как это происходит для одно- и двумерных текстур.

Обратите внимание на то, что если в одном и том же текстурном блоке одновременно разрешены текстуры разных типов, то работает механизм приоритетов: наиболее приоритетной текстурой является кубическая текстурная карта, затем в порядке убывания приоритета идут трехмерная, двумерная и одномерная текстуры.

В листинге 6.1 приводится функция, создающая и загружающая в память графического ускорителя трехмерную текстуру. В качестве значений цветов для текселов текстуры используются координаты соответствующих точек единичного трехмерного куба (т. е. текселу с координатами (*s*, *t*, *r*) сопоставляется RGB-значение, численно равное (*s*, *t*, *r*)).

Листинг 6.1. Создание трехмерной текстуры

```
unsigned    createSimpleTexture ( int size )
{
    printf ( "Creating 3D texture.\n" );
    unsigned    texture;
    byte      * data    = new byte [3*size*size*size];
    int         offs    = 0;
    float       step    = 1.0f / (float) size;
    for ( int i = 0; i < size; i++ )
        for ( int j = 0; j < size; j++ )
            for ( int k = 0; k < size; k++ )
            {
                data [offs++] = (byte)(255*step*i);
                data [offs++] = (byte)(255*step*j);
                data [offs++] = (byte)(255*step*k);
            }
    glGenTextures ( 1, &texture);
    glBindTexture ( GL_TEXTURE_3D_EXT, texture );
    glTexImage3DEXT ( GL_TEXTURE_3D_EXT, 0, GL_RGB, size, size, size,
                      0, GL_RGB, GL_UNSIGNED_BYTE, data );
    glTexParameteri ( GL_TEXTURE_3D_EXT, GL_TEXTURE_MAG_FILTER,
                      GL_LINEAR );
    glTexParameteri ( GL_TEXTURE_3D_EXT, GL_TEXTURE_MIN_FILTER,
                      GL_LINEAR );
    glTexParameteri ( GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_S,
                      GL_REPEAT );
    glTexParameteri ( GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_T,
                      GL_REPEAT );
    glTexParameteri ( GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_R,
                      GL_REPEAT );
    delete data;
    printf ( "Done.\n" );
    return texture;
}
```

Следующий далее фрагмент кода (листинг 6.2) демонстрирует применение трехмерной текстуры для текстурирования сложного объекта, при этом вводится матрица преобразования текстурных координат, чтобы сохранить

неизменными координаты соответствующих точек при преобразованиях объекта.

Листинг 6.2. Пример использования трехмерной текстуры

```
//
// Sample to show creating and using 3D textures in OpenGL
//
#include    "libExt.h"
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Torus.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
Vector3D    eye   ( 2, 2, 2 );            // camera position
unsigned    tex3D;
float       angle  = 0;
float       angle2 = 0;
Torus       torus ( 1, 3, 30, 30 );
unsigned    createSimpleTexture ( int size )
{
    printf ( "Creating 3D texture.\n" );
    unsigned    texture;
    byte      * data    = new byte [3*size*size*size];
    int         offs    = 0;
    float       step    = 1.0f / (float) size;
for ( int i = 0; i < size; i++ )
        for ( int j = 0; j < size; j++ )
            for ( int k = 0; k < size; k++ )
            {
                data [offs++] = (byte)(255*step*i);
                data [offs++] = (byte)(255*step*j);
                data [offs++] = (byte)(255*step*k);
            }
    glGenTextures ( 1, &texture);
    glBindTexture ( GL_TEXTURE_3D_EXT, texture );
```

```
    glTexImage3DEXT ( GL_TEXTURE_3D_EXT, 0, GL_RGB, size, size, size,
                      0, GL_RGB, GL_UNSIGNED_BYTE, data );

    glTexParameteri ( GL_TEXTURE_3D_EXT, GL_TEXTURE_MAG_FILTER,
                      GL_LINEAR );
    glTexParameteri ( GL_TEXTURE_3D_EXT, GL_TEXTURE_MIN_FILTER,
                      GL_LINEAR );
    glTexParameteri ( GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_S,
                      GL_REPEAT );
    glTexParameteri ( GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_T,
                      GL_REPEAT );
    glTexParameteri ( GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_R,
                      GL_REPEAT );
    delete data;
    printf ( "Done.\n" );
    return texture;
}
void init ()
{
    glClearColor ( 0, 0, 0, 1 );
    glEnable     ( GL_DEPTH_TEST );
    glHint ( GL_POLYGON_SMOOTH_HINT,         GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}
void display ()
{
    int    i;
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glEnable      ( GL_TEXTURE_3D_EXT );
    glBindTexture ( GL_TEXTURE_3D_EXT, tex3D );
    glMatrixMode  ( GL_MODELVIEW );
    glPushMatrix  ();
    glTranslatef  ( -4, -4, -4 );
    glRotatef     ( angle,  1, 1, 0 );
    glRotatef     ( angle2, 0, 1, 1 );
    torus.draw ();
    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 key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
        exit ( 0 );
    if ( key == 'p' || key == 'P' )                 // screenshot
        saveScreenShot ( "texture-3D.tga" );
}
void    animate ()
{
    angle  = 0.04f * glutGet ( GLUT_ELAPSED_TIME );
    angle2 = 0.02f * 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
    int    win = glutCreateWindow ( "OpenGL EXT_texture3D example " );
                            // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
```

```
    if ( !isExtensionSupported ( "GL_EXT_texture3D" ) )
    {
        printf ( "EXT_texture3D NOT supported.\n" );
        return 1;
    }
    initExtensions ();
    tex3D  = createSimpleTexture ( 32 );
    init ();
    glutMainLoop ();
    return 0;
}
```

Не забывайте, что трехмерные текстуры (в отличие от одномерных и двумерных) занимают гораздо больше памяти (так трехмерная RGBA-текстура размером 256×256×256 занимает 64 Мбайт видеопамяти), поэтому их использование может отрицательно сказаться на быстродействии программы.

Далеко не все форматы изображений поддерживают хранение трехмерных текстур. Наиболее подходящими для работы с ними являются DDS-файлы.

На прилагаемом к книге компакт-диске содержится библиотека libTexture3D, которая позволяет загружать в память графического ускорителя трехмерные текстуры из DDS-файлов и сохранять массивы текселов как трехмерную текстуру в файле.

Листинг 6.3 иллюстрирует заголовочный файл данной библиотеки.

Листинг 6.3. Файл libTexture3D.h

```
//
// load and save 3D images from/to DDS file's
#ifndef    __LIB_TEXTURE_3D__
#define    __LIB_TEXTURE_3D__
class    Data;
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 );
#endif
```

Функция `saveTexture3D` сохраняет трехмерную текстуру размером `width*height*depth` с числом байтов на тексел, заданных параметром `components`, в файл с заданным именем в формате DDS. Функции `createTexture3D` служат для загрузки в память графического ускорителя трехмерной текстуры из файла формата DDS. В качестве входного параметра может выступать как имя файла, так и объект класса `Data`.

Воспользуемся этой библиотекой для создания трехмерной текстуры, содержащей значения так называемой шумовой функции (*Perlin noise*).

Впервые введенная Кеном Перлином шумовая функция представляет собой непрерывную функцию, принимающую значения из отрезка [−1, 1]. Ее поведение очень напоминает так называемый белый шум, в котором практически незаметны какие-либо регулярности, что делает очень удобным применение этой функции для представления различных стохастических процессов или явлений.

В частности, эта функция очень широко применяется при написании различных шейдеров — программ, задающих закон закрашивания поверхности. С ее помощью можно легко придать поверхности предмета некоторую нерегулярность.

На рис. 6.3 представлен график двумерной шумовой функции (в виде текстуры).

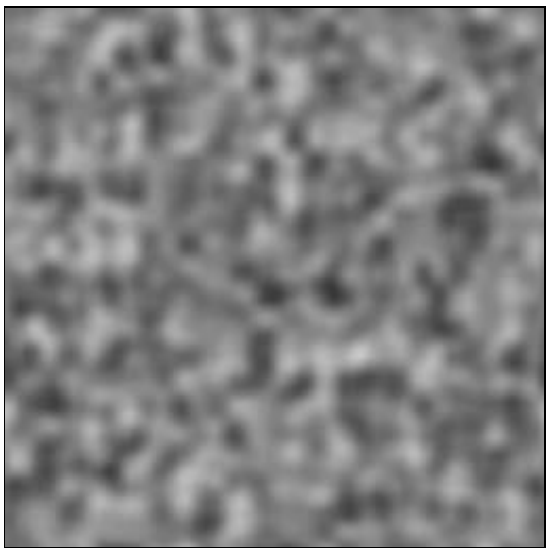


Рис. 6.3. График шумовой функции от двух переменных

Однако непосредственное вычисление такой функции (особенно от нескольких переменных) требует больших вычислительных затрат, поэтому во многих случаях она задается при помощи заранее рассчитанной текстуры. Но в этом случае ее значения на противоположных краях должны совпадать, иначе при выходе текстурных координат за единичный квадрат произойдет скачок, что крайне нежелательно.

Поэтому, как правило, сначала строится обычная шумовая текстура, а затем обеспечивается совпадение значений на ее краях.

Однако во многих случаях обычной двумерной функции бывает недостаточно, поэтому возникает потребность в трехмерной шумовой текстуре. В листинге 6.4 приводится полный исходный текст утилиты, строящей по заданным параметрам такую текстуру.

Листинг 6.4. Утилита для построения трехмерной шумовой текстуры

```
//
// Create a 3D periodic noise texture and save it in DDS file
//
#include    "libExt.h"
#include    "noise.h"
#include    "libTexture3D.h"

#include    <stdio.h>
class    VCube
{
    int            n;
    int            n2;
    Vector3D  * v;
public:
    VCube ( int size )
    {
        n  = size;
        n2 = n*n;
        v  = new Vector3D [n*n*n];
    }
    ~VCube ()
    {
        delete v;
    }
```

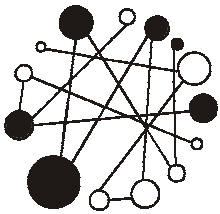
```
    Vector3D&    vertex ( int i, int j, int k )
    {
        return v [i*n2 + j*n + k];
    }
};
inline    float    drop ( float t )
{
    return 1.0f - pow ( t, 40 );
}
const Vector3D    offsGreen ( 0.12345, 0.23456, 0.34567 );
const Vector3D    offsBlue  ( 0.35792, 0.46803, 0.79486 );
static    byte * createNoiseTextureBytes ( int size, int octaves )
{
                            // build cube lattice
                            // with noise
    VCube        vc ( size );
    int          i, j, k;
    float        inv  = 1.0f / (float)size;
    float        inv2 = inv * (float)octaves;
    Vector3D     v;
    Noise        noise;
                            // fill the lattice
                            // with noise
    for ( i = 0; i < size; i++ )
        for ( j = 0; j < size; j++ )
            for ( k = 0; k < size; k++ )
            {
                v.x = i * inv2;
                v.y = j * inv2;
                v.z = k * inv2;
                vc.vertex ( i, j, k ) =
                                Vector3D ( noise.noise ( v ),
                                    noise.noise (v+offsGreen),
                                    noise.noise (v+offsBlue ) );
            }
                            // now we want to make
                            // it periodic by // adjuisting  border
                            // values
```

```
    VCube    vc2 ( size );
    for ( i = 0; i < size; i++ )
        for ( j = 0; j < size; j++ )
            for ( k = 0; k < size; k++ )
            {
                float    xDrop = drop ( i * inv );
                float    yDrop = drop ( j * inv );
                float    zDrop = drop ( k * inv );
                vc2.vertex ( i, j, k ) =
xDrop*yDrop*zDrop*vc.vertex (i,j,k)+
(1-xDrop)*yDrop*zDrop*vc.vertex (size-1-i,j,k)+
xDrop*(1-yDrop)*zDrop*vc.vertex (i,size-1-j,k)+
(1-xDrop)*(1-yDrop)*zDrop*vc.vertex (size-1-i,size-1-j,k)+
xDrop*yDrop*(1-zDrop)*vc.vertex (i,j,size-1-k)+
(1-xDrop)*yDrop*(1-zDrop)*vc.vertex (size-1-i,j,size-1-k)+
xDrop*(1-yDrop)*(1-zDrop)*vc.vertex (i,size-1-j,size-1-k)+
(1-xDrop)*(1-yDrop)*(1-zDrop)*vc.vertex (size-1-i,size-1-j,size-1-k ;
            }
                            // now create 3D texture
    byte * noiseTable = (byte *) malloc ( size * size * size * 3 );
    if ( noiseTable == NULL )
        return NULL;
    byte  * ptr  = noiseTable;
    for ( i = 0; i < size; i++ )
        for ( j = 0; j < size; j++ )
            for ( k = 0; k < size; k++ )
            {
                Vector3D    n ( vc2.vertex ( i, j, k ) );
                *ptr++ = (byte)(127.5 * ( 1.0 + n.x ) );
                *ptr++ = (byte)(127.5 * ( 1.0 + n.y ) );
                *ptr++ = (byte)(127.5 * ( 1.0 + n.z ) );
            }
    return noiseTable;
}
bool        saveNoiseTexture3D   ( int size, int octaves,
                                   const char * fileName )
{
    byte * data = createNoiseTextureBytes ( size, octaves );
```

```
    if ( data == NULL )
        return false;
    bool    result = saveTexture3D ( size, size, size, 3, data,
                                     fileName );
    free ( data );
    return result;
}
int    main ( int argc, char * argv [] )
{
    if ( argc != 4 )
    {
        printf ("Usage:\n" );
        printf ("\tmake-3d-noise size octaves filename\n" );
        printf ("Where\n\tsize     is texture size in texels,\n" );
        printf ("\toctaves  is number of \"periods\" of noise\n" );
        printf ("\tfilename is the name (with dds extension) "
"texture will be saved to.\n" );
return 1;
    }
    int    size    = atoi ( argv [1] );
    int    octaves = atoi ( argv [2] );
    if ( size < 1 || octaves < 1 )
    {
        printf ( "Illegal data.\n" );
    return 1;
    }
    printf ( "Building %dx%dx%d noise texture...\n" );
    if ( saveNoiseTexture3D ( size, octaves, argv [3] ) )
    printf ( "3D noise texture successfully created !\n" );
    else
        printf ( "Error creating texture.\n" );
    return 0;
}
```

Мы будем активно использовать шумовые текстуры в *главе 18* при написании шейдеров на языке GLSL.

Глава 7



Расширения ARB_point_paramters и ARB_point_sprite для создания систем частиц

Оба расширения, рассматриваемые в этой главе, предназначены для облегчения создания и работы с так называемыми системами частиц (подробнее о них в [1]). При этом постоянно возникает необходимость вывода большого числа точек различных размеров или небольших прямоугольных изображений — спрайтов (*sprite*).

Хотя в стандартном OpenGL поддерживается возможность задания размеров выводимых точек функцией `glPointSize`, однако при этом на размер выводимой точки не влияет расстояние до нее, и отображение точек маленьких размеров не всегда происходит достаточно красиво.

Расширение ARB_point_parameters позволяет задавать для выводимых точек уменьшение их размеров (ослабление яркости) в зависимости от расстояния и использовать альфа-канал цвета выводимой точки для более точного учета ее размера.

При выводе точки ее размер преобразуется в зависимости от расстояния до наблюдателя согласно формуле

$$size' = clamp\left(size \cdot \sqrt{atten(d)}\right),$$

где через *size* обозначен исходный размер точки, задаваемый функцией `glPointSize`; через d — расстояние от наблюдателя до точки в системе координат наблюдателя.

Функция *clamp* осуществляет приведение значения своего аргумента в отрезок [*sizeMin*, *sizeMax*]:

$$clamp(s) = \begin{cases} s, s \in [sizeMin, sizeMax]; \\ sizeMin, s \le sizeMin; \\ sizeMax, s \ge sizeMax. \end{cases}$$

Коэффициент ослабления с учетом расстояния задается следующей формулой:

$$atten(d) = \frac{1}{a + b \cdot d + c \cdot d^2}.$$

Коэффициенты a, b и c задаются пользователем.

При растеризации точки ее диаметр преобразуется по закону

$$diameter(d') = \begin{cases} d', d' \geq d_t; \\ d_t, d' < d_t; \end{cases} \qquad fade(d') = \begin{cases} 1, d' \geq d_t; \\ \left(\frac{d'}{d_t}\right)^2, \ d' < d_t. \end{cases}$$

В результате выводится точка с новым диаметром, а ее значение альфа-канала умножается на коэффициент *fade*, что позволяет более аккуратно формировать изображение маленьких точек.

В этих формулах величина d_t задает некоторую границу, начиная с которой передача размера происходит путем изменения прозрачности точки.

Для задания всех указанных параметров служат вводимые этим расширением функции:

```
void glPointParameterfARB  ( GLenum pname, GLfloat param );
void glPointParameterfvARB ( GLenum pname, GLfloat *params );
```

Ниже приводится пример задания коэффициентов ослабления a, b и c.

```
float    coeffs [] = { 1, 0, 0.01f };
glPointParameterfvARB ( GL_POINT_DISTANCE_ATTENUATION_ARB, coeffs );
```

Задание величины d_t иллюстрирует следующий фрагмент кода:

```
glPointparameterfARB ( GL_POINT_FADE_THRESHOLD_SIZE_ARB, dt );
```

Коэффициенты `sizeMin` и `sizeMax`, задающие диапазон изменения реального размера точки, могут быть определены при помощи следующего фрагмента кода:

```
glPointParameterfARB ( GL_POINT_SIZE_MIN_ARB, sizeMin );
glPointParameterfARB ( GL_POINT_SIZE_MAX_ARB, sizeMax );
```

Текущие значения используемых параметров можно получить через функцию `glGet*`, как показано в листинге 7.1.

Листинг 7.1. Получение текущих параметров точки

```
float    minSize = 0;
float    maxSize = 0;
float    dt      = 0;
float    coeffs [3];
```

```
glGetFloatv ( GL_POINT_SIZE_MIN_ARB,              &minSize );
glGetFloatv ( GL_POINT_SIZE_MAX_ARB,              &maxSize );
glGetFloatv ( GL_POINT_FADE_THRESHOLD_SIZE_ARB,   &dt      );
glGetFloatv ( GL_POINT_DISTANCE_ATTENUATION_ARB, coeffs   );
```

Однако для систем частиц чаще возникает необходимость вывода не просто точек, а прямоугольных изображений, ориентированных параллельно экрану (так называемых *billboard*, далее мы будем употреблять термин спрайт).

Хотя вывод таких спрайтов легко можно реализовать при помощи стандартных возможностей OpenGL (примеры есть в [1]), но расширение ARB_point_sprite позволяет осуществлять вывод большого числа спрайтов гораздо проще и быстрее.

Данное расширение позволяет при выводе точки на самом деле выводить квадрат (с размером, соответствующим размеру точки), параллельный экрану так, что его ребра параллельны осям экрана. При выводе подобного квадрата на него натягивается выбранная в данный момент текстура (рис. 7.1).

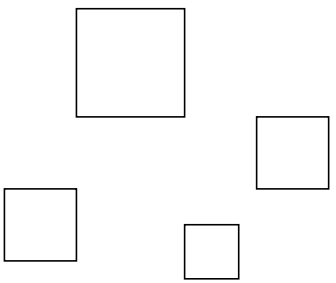


Рис. 7.1. Вывод спрайтов

Команда

```
glEnable ( GL_POINT_SPRITE_ARB );
```

разрешает режим вывода точек в виде спрайтов, причем она действует сразу на все имеющиеся текстурные блоки.

Поскольку в этом режиме вместо одной точки выводится квадрат, то необходим механизм, позволяющий автоматически вычислять корректные значения текстурных координат для всех точек выводимого спрайта.

Этот режим включается при помощи команды

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

На рис. 7.2 показано, как задаются текстурные координаты в этом случае.

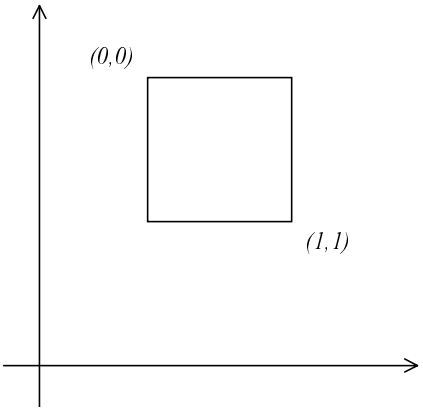


Рис. 7.2. Назначение текстурных координат для выводимого спрайта

Если режим автоматического вычисления текстурных координат выключен, то всем точкам выводимого квадрата будет сопоставлено одно и то же значение текстурных координат, являющееся текущим на момент вывода точки.

Режим автоматического вычисления текстурных координат задается отдельно для каждого текстурного блока.

Помните, что отсечение выводимых спрайтов производится по таким же правилам, как для точек, т. е. если точка, соответствующая спрайту, отсекается, то весь этот спрайт также не выводится.

При этом все выводимые спрайты подвержены действию расширения ARB_point_parameters, т. е. их размер и прозрачность зависят от расстояния до наблюдателя.

Параметры вывода точки (спрайта) могут быть занесены в стек при помощи команды `glPushAttrib` со значением бита `GL_POINT_BIT`. Режим включения/выключения использования спрайтов также сохраняется/восстанавливается при помощи функции `glPushAttrib`/`glPopAttrib` со значением `GL_ENABLE_BIT`.

Посредством расширений ARB_point_parameters и ARB_point_sprite можно легко реализовать широкий класс так называемых систем частиц (*particle systems*).

Системы частиц часто необходимы для моделирования различных сложных природных явлений, таких как огонь, взрывы, фонтаны, водопады и т. п.

Фактически система частиц представляет собой очень большое число простейших объектов (называемых частицами), обычно отображаемых на экране при помощи спрайтов. Каждая частица подчиняется простому закону анимации, определяющему изменение ее положения, цвета и других параметров со

временем. При этом за счет очень большого числа элементарных объектов (частиц) удается получить в итоге достаточно сложное поведение и интересные визуальные эффекты даже при достаточно простых законах анимации.

Ниже мы рассмотрим реализацию нескольких простейших систем частиц при помощи введенных в этой главе расширений ARB_point_parameters и ARB_point_sprite. Мы будем далее считать, что каждая частица обладает следующими атрибутами: положением в пространстве, скоростью, массой, цветом, временем жизни, размером и уже прожитым временем.

Листинг 7.2 содержит описание структуры, инкапсулирующей внутри себя всю информацию о частице.

Листинг 7.2. Описание частицы

```
struct    Particle
{
    Vector3D    pos;
    Vector4D    color;
    Vector3D    velocity;
    float       mass;
    float       time;
    float       lifeTime;
    bool        alive;
};
```

В этой структуре поля `pos`, `color`, `velocity`, `mass`, `lifeTime` соответствуют положению частицы в пространстве, ее цвету, скорости, массе и продолжительности жизни. Поле `time` содержит в себе уже прожитое частицей время (как только его значение превысит `lifeTime`, частица будет уничтожена). Значения поля `alive` являются признаком использования соответствующей частицы в данный момент времени.

Для организации всех частиц в систему мы введем специальный класс, инкапсулирующий внутри себя основные операции с частицами: создание, анимацию и рендеринг.

Листинг 7.3 содержит интерфейс такого класса.

Листинг 7.3. Описание класса `ParticleSystem`

```
class    ParticleSystem
{
protected:
```

```
    Particle  *  particle;
    int          maxParticles;
    int          numParticles;
    float        lastUpdateTime;
    float        lastCreateTime;
    float        birthRate;
    float        size;
    Vector3D     pos;
    int          srcBlend, dstBlend;
    unsigned     textureId;
public:
    ParticleSystem ( int theMaxParticles, float theBirthRate,
                     const Vector3D& thePos, unsigned id = 0 );
    virtual ~ParticleSystem ();
    void    setTexture ( unsigned id )
    {
        textureId = id;
    }
    virtual    void    update         ();
    virtual    void    render         ();
    virtual    void    createParticle ( Particle& ) = 0;
};
```

Важным обстоятельством, которое следует иметь в виду при работе с системами частиц, является необходимость оптимизации, поскольку в противном случае наличие большого числа частиц в системе способно заметно замедлить программу. Наиболее существенным в оптимизации является создание и уничтожение отдельных частиц. Явное выделение и освобождение памяти (например, командами `new` и `delete`) может привести к довольно большим временным затратам. Поэтому обычно заранее выделяется память на достаточно большое количество частиц, и каждая частица снабжается флагом (`alive`), определяющим, активна ли она сейчас или свободна и может быть переиспользована.

Вызов виртуального метода для каждой отдельной частицы может тоже оказаться весьма дорогостоящим, поэтому гораздо лучше в одном виртуальном методе обрабатывать сразу все частицы системы.

Рассмотрим основные методы описанного выше класса `ParticleSystem`.

В конструкторе этого класса осуществляется выделение памяти и инициализация основных переменных. К ним относится скорость рождения частиц (число частиц в секунду), положение системы, выбранная текстура.

В деструкторе класса осуществляется освобождение выделенной под частицы памяти. Пример иллюстрирует листинг 7.4.

Листинг 7.4. Конструктор и деструктор для класса `ParticleSystem`

```
ParticleSystem :: ParticleSystem ( int theMaxParticles,
                                   float theBirthRate,
                                   const Vector3D& thePos, unsigned id )
{
    maxParticles   = theMaxParticles;
    particle       = new Particle [maxParticles];
    numParticles   = 0;
    birthRate      = theBirthRate;
    pos            = thePos;
    textureId      = id;
    lastUpdateTime = 0.001f * glutGet ( GLUT_ELAPSED_TIME );
    lastCreateTime = lastUpdateTime;
    srcBlend       = GL_SRC_ALPHA;
    dstBlend       = GL_ONE;
    memset ( particle, '\0', maxParticles * sizeof ( Particle ) );
}
ParticleSystem :: ~ParticleSystem ()
{
    delete particle;
}
```

Метод `render` для вывода системы частиц крайне прост:

1. Сохраняются текущие параметры состояния OpenGL.
2. Включается режим использования спрайтов.
3. Осуществляется вывод всех активных частиц.
4. По окончании вывода состояние OpenGL восстанавливается.

Обратите внимание, что при выводе системы частиц отключается запись в буфер глубины, однако тест глубины оставляется включенным.

Пример реализации метода `render` содержит листинг 7.5.

Листинг 7.5. Метод `render` класса `ParticleSystem`

```
void    ParticleSystem :: render ()
{
    glPushAttrib  ( GL_ENABLE_BIT | GL_POINT_BIT |
                       GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glEnable      ( GL_TEXTURE_2D );
    glDepthMask   ( GL_FALSE );
    glBindTexture ( GL_TEXTURE_2D, textureId );
    glEnable      ( GL_BLEND );
    glBlendFunc   ( srcBlend, dstBlend );
    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 );
    glBegin ( GL_POINTS );
       for( int i = 0; i < maxParticles; i++ )
            if ( particle [i].alive )
               {
                glColor4fv  ( particle [i].color );
                glVertex3fv ( particle [i].pos   );
               }
    glEnd();
    glPopAttrib ();
}
```

Более сложным является метод `update`, отвечающий за анимацию системы. В нем выполняются сразу три функции:

1. Уничтожение частиц, время жизни которых истекло.
2. Создание новых частиц.
3. Передвижение частиц с учетом скорости каждой из них.

Данный метод иллюстрирует листинг 7.6.

Листинг 7.6. Метод `update` класса `ParticleSystem`

```
void    ParticleSystem :: update ()
{
    float    curTime = 0.001f * glutGet ( GLUT_ELAPSED_TIME );
    float    delta   = curTime - lastUpdateTime;
    register int    i, j;
    for ( i = 0; i < maxParticles; i++ )
        if ( particle [i].alive )
        {
            particle [i].time += delta;
            if ( particle [i].time >= particle [i].lifeTime )
            {
                particle [i].alive = false;
                continue;
            }
            particle [i].pos += delta * particle [i].velocity;
        }
    lastUpdateTime = curTime;
                        // now add enough particles
    int    newParticles = (int)( (curTime - lastCreateTime) *
                                    birthRate + 0.5f );
    for ( i = j = 0; i < newParticles; i++ )
    {
                        // find free slot
        while ( j < maxParticles && particle [j].alive )
            j++;
        if ( j >= maxParticles )
            return;
                        // setup some default values
        particle [j].alive = true;
        particle [j].time  = 0;
        particle [j].mass  = 1;
        createParticle ( particle [j] );
    }
    if ( newParticles > 0 )
        lastCreateTime = curTime;
}
```

При этом, поскольку в различных системах инициализация новой частицы происходит по-разному, то метод, отвечающий за ее инициализацию, сделан виртуальным и может переопределяться в дочерних классах.

Листинг 7.7 содержит примеры нескольких простейших систем частиц на основе класса `ParticleSystem`.

Листинг 7.7. Различные типы систем частиц на основе класса `ParticleSystem`

```
class    Flame : public ParticleSystem
{
protected:
    float        radius;
    float        randCoeff;
    float        centerCoeff;
    ColorSpline  colors;
public:
    Flame ( int theMaxParticles, float theBirthRate,
            const Vector3D& thePos, float r, float r1, float r2 ) :
            ParticleSystem ( theMaxParticles, theBirthRate, thePos )
    {
        radius      = r;
        randCoeff   = r1;
        centerCoeff = r2;
        size        = 15;
        colors.addColorAt ( Vector4D ( 0,   0.3, 1, 0.5 ), 0    );
        colors.addColorAt ( Vector4D ( 0,   0.7, 1, 1   ), 0.15 );
        colors.addColorAt ( Vector4D ( 1,   1,   0, 1   ), 0.2  );
        colors.addColorAt ( Vector4D ( 1,   0,   0, 1   ), 0.8  );
        colors.addColorAt ( Vector4D ( 1,   0,   0, 0   ), 1    );
    }

    virtual     void    createParticle ( Particle& p )
    {
        p.pos      = pos + Vector3D :: getRandomVector ( radius );
        p.lifeTime = 1.5;
        p.velocity = Vector3D ( 0, 0, 0 );
    }
    virtual void    update ()
```

```
    {
        float    saveTime = lastUpdateTime;
        ParticleSystem :: update ();
                            // interval in milliseconds since last update
        float    delta = lastUpdateTime - saveTime;
        for ( register int i = 0; i < maxParticles; i++ )
        {
            if ( !particle [i].alive )
                continue;
                            // create random variation
            Vector3D    v = Vector3D :: getRandomVector (
                                                    randCoeff );
            v.z = 1;        // particle allways goes up

            particle [i].pos += delta * v;
                        // now move projection of
                        // particle to
                        // center (in Oxy plane)
            float d  = centerCoeff*delta*(particle [i].pos.z -
                                                  pos.z);
            float dx = (particle [i].pos.x - pos.x) * d;
            float dy = (particle [i].pos.y - pos.y) * d;
            particle [i].pos.x -= dx;
            particle [i].pos.y -= dy;
                        // compute color
            particle [i].color = colors.valueAt (
                            particle [i].time / particle [i].lifeTime );
        }
    }
};
class    Emitter : public ParticleSystem
{
protected:
    Vector4D    color;
    float        minLifeTime;
    float        maxLifeTime;
    float        velCoeff;
public:
    Emitter ( int theMaxParticles, float theBirthRate,
```

```
                const Vector3D& thePos, const Vector4D& theColor,
                float theMinLife, float theMaxLife ) :
                ParticleSystem ( theMaxParticles, theBirthRate, thePos )
    {
        color       = theColor;
        minLifeTime = theMinLife;
        maxLifeTime = theMaxLife;
        velCoeff    = 0.2;
        size        = 20;
    }
    virtual void    update ()
    {
        ParticleSystem :: update ();
        for ( register int i = 0; i < maxParticles; i++ )
            if ( particle [i].alive )
                particle [i].color.w = 1 -
                                particle [i].time / particle [i].lifeTime;
    }
    virtual void    createParticle ( Particle& p )
    {
        p.pos      = pos;
        p.color    = color;
        p.velocity = Vector3D :: getRandomVector ( velCoeff );
        p.lifeTime = rnd ( minLifeTime, maxLifeTime );
    }
};
```

При этом для интерполирования цветов во времени используется класс `ColorSpline`, исходный текст которого приводится в листинге 7.8.

С помощью этого класса можно задать набор пар (*параметр*, *цвет*), после чего для произвольного значения параметра путем линейной интерполяции определяется значение цвета. Данный класс позволяет легко реализовывать довольно сложные переходы цветов путем задания всего лишь нескольких пар (*параметр*, *цвет*).

Листинг 7.8. Класс `ColorSpline`

```
#include        "Vector4D.h"
#define         MAX_COLOR_SPLINE_ENTRIES    20
class    ColorSpline
```

```
{
    struct Entry
    {
        float     t;            // parameter value
        Vector4D  color;        // corresponding color value
        Vector4D  delta;        // delta value used in
                                // interpolation
    };
    Entry    entries [MAX_COLOR_SPLINE_ENTRIES];
    int    numEntries;
public:
    ColorSpline ()
    {
        numEntries = 0;
    }
    void    addColorAt ( const Vector4D& color, float t )
    {
        if ( numEntries >= MAX_COLOR_SPLINE_ENTRIES )
            return;
                    // store the entry
        entries [numEntries].t     = t;
        entries [numEntries].color = color;
                    // compute delta for prev. entry
        if ( numEntries > 0 )
            entries [numEntries-1].delta =
                         (entries [numEntries].color -
                          entries [numEntries-1].color) /
                    (entries [numEntries].t -
                          entries [numEntries-1].t);
        numEntries++;
    }
    Vector4D    valueAt ( float t ) const
    {
        if ( t < entries [0].t )
            return entries [0].color;
        for ( register int i = 1; i < numEntries; i++ )
            if ( t <= entries [i].t )
```

```
                return entries [i-1].color +
                                (t-entries [i-1].t) * entries [i-1].delta;
        return entries [numEntries - 1].color;
    }
};
```

Метод `addColorAt` добавляет в таблицу очередную пару (*параметр*, *цвет*). Считается, что пары поступают по возрастанию параметра.

Метод `valueAt` возвращает интерполированное значение цвета, соответствующее данному значению параметра. Это значение вычисляется методом кусочно-линейной интерполяции.

На рис. 7.3 приводится изображение двух систем частиц рассмотренных типов, полученное при помощи демонстрационной программы (листинг 7.9).

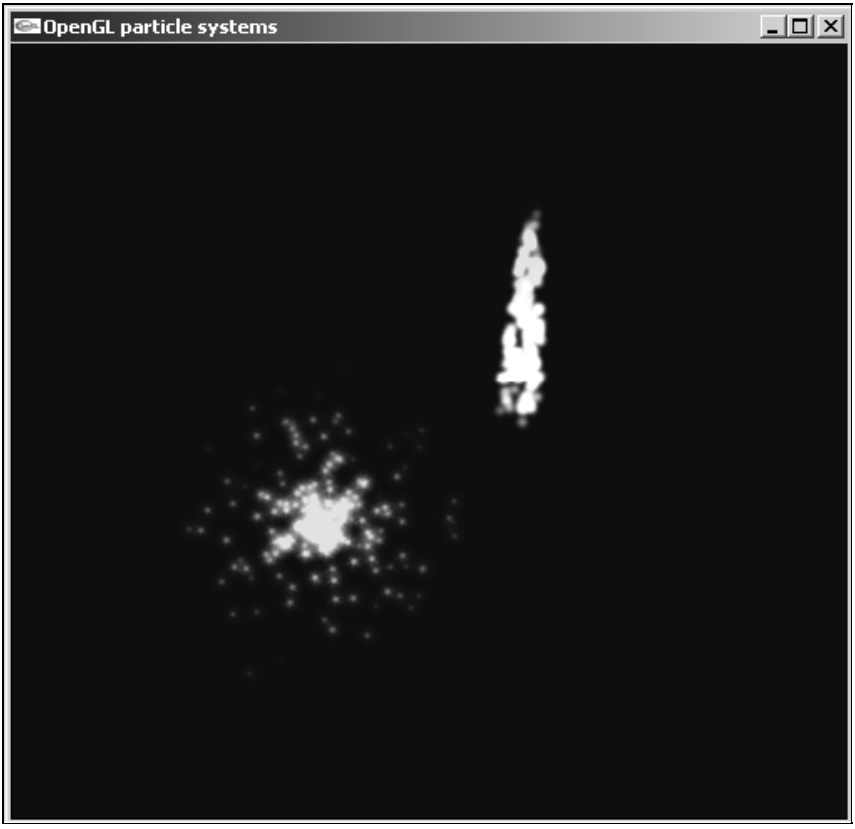


Рис. 7.3. Демонстрация изображения систем частиц

Листинг 7.9. Пример использования систем частиц

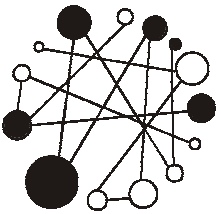
```
//
// Sample to illustrate particle systems via ARB_point_parameters &
// ARB_point_sprite extensions.
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    "libTexture.h"
#include    "Vector2D.h"
#include    "Emitter.h"
#include    "Flame.h"
Vector3D    eye   ( 3, 3, 3 );            // camera position
Flame       flame   ( 3000, 50, Vector3D ( -1, 0, 0 ), 0.15, 0.2, 1 );
Emitter     emitter ( 1000, 30, Vector3D (  1, 0, 0 ),
                 Vector4D ( 1, 0, 0, 1 ), 4, 6 );
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 );
    flame.  render ();
    emitter.render ();
    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 key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
                exit ( 0 );
}
void    animate ()
{
    flame.  update ();
    emitter.update ();

    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                        // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                        // create window
    int    win = glutCreateWindow ( "OpenGL particle systems" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init           ();
    initExtensions ();
    if ( !isExtensionSupported ( "GL_ARB_point_parameters" ) )
    {
        printf ( "ARB_point_parameters NOT supported.\n" );
        return 1;
    }
```

```
    if ( !isExtensionSupported ( "GL_ARB_point_sprite" ) )
    {
        printf ( "GL_ARB_point_sprite NOT supported" );
        return 2;
    }
    unsigned    texture1 = createTexture2D ( true,
                                    "../Textures/particle1.bmp" );
    unsigned    texture2 = createTexture2D ( true,
                                     "../Textures/particle.bmp" );
    flame.setTexture   ( texture1 );
    emitter.setTexture ( texture2 );
    glutMainLoop ();
    return 0;
}
```

Глава 8



Простейшая модель попиксельного освещения, карты нормалей и работа с ними. Расширение ARB_texture_env_dot3

Довольно популярным в аппаратно-ускоренной графике является так называемое попиксельное (*per-pixel*) освещение и отражение.

Рассмотрим, каким образом можно реализовать простейшее попиксельное освещение средствами OpenGL.

В качестве модели освещения мы возьмем простейшее диффузное освещение, когда освещенность каждого видимого пиксела задается формулой

$$I = \max\,[\,(l, n), 0\,], \tag{8.1}$$

где через l и n обозначены единичные векторы направления от точки к источнику света и нормали к поверхности в этой точке (рис. 8.1). Благодаря функции max мы избавлены от появления "отрицательной" освещенности (что соответствует случаю, когда нормаль направлена от источника света, и свет падать на эту точку просто не может). Подробнее о диффузном освещении и о других моделях освещения вы можете прочитать в *Приложении 2.*

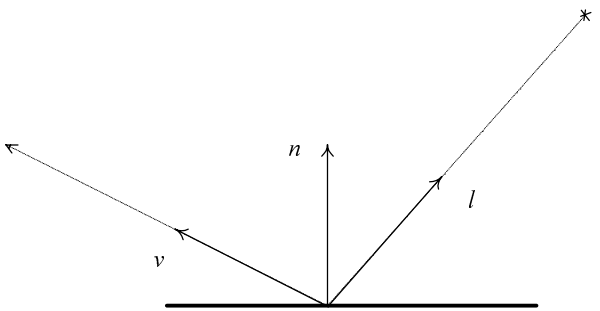


Рис. 8.1. Диффузное освещение в точке

Хотя используемая в OpenGL модель освещения гораздо сложнее (она, в частности, учитывает бликовое освещение и расстояние до источника света), но и здесь значения освещенности вычисляются лишь в вершинах граней (так называемое вершинное освещение, *per-vertex lighting*). Далее эти значения просто интерполируются вдоль граней (путем билинейной интерполяции). Именно поэтому при выборе стандартной модели освещения OpenGL не может быть бликов внутри грани, если их нет ни в одной из вершин (поскольку при полилинейной интерполяции значение функции внутри области не может превышать граничного).

В этой главе мы реализуем простейшее попиксельное освещение, при котором значение освещенности по формуле (8.1) будет независимо вычисляться для каждого пиксела выводимой грани.

Простейший случай

Для построения попиксельного диффузного освещения в первую очередь нам понадобятся значения векторов l и n для каждого пиксела.

Можно задать единичные значения вектора l в виде цвета или текстурных координат в каждой вершине грани (например, как цвет или текстурные координаты, рис. 8.2).

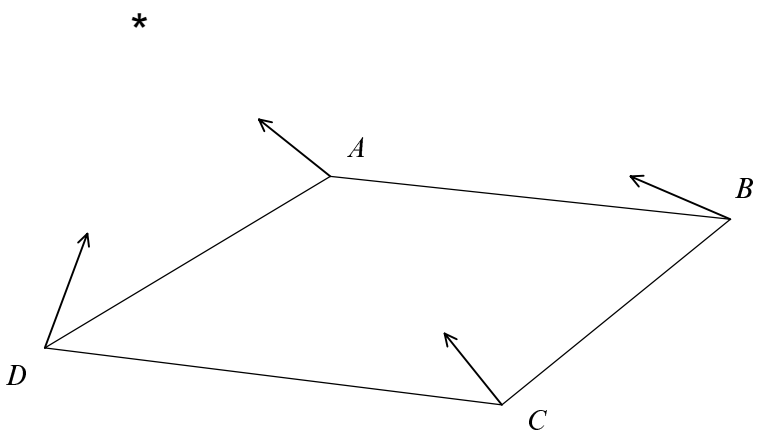


Рис. 8.2. Задание вектора l в вершинах

Если мы в каждой из вершин A, B, C и D зададим значение вектора l, то стандартными средствами OpenGL можно произвести его билинейную интерполяцию (вдоль всего многоугольника ABCD) и для каждого пиксела получить проинтерполированное значение. Простейшим способом этого является задание вектора l через текстурные координаты (s, t, r).

Однако подобный подход обладает весьма серьезным недостатком. Даже если значения, заданные в вершинах, и были единичными векторами, то полученные в результате интерполяции, скорее всего единичными уже не будут (рис. 8.3).

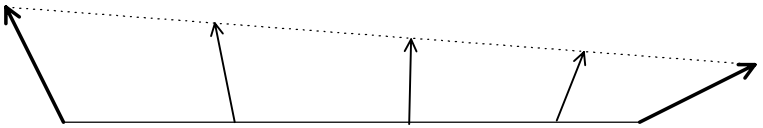


Рис. 8.3. Интерполяция единичных векторов может давать произвольные значения

Поэтому перед вычислением скалярного произведения необходимо провести нормализацию вектора *l*, например, с помощью нормирующих кубических карт (*normalization cube map*) (рис. 8.4).

Как уже упоминалось в *главе 5*, кубическая карта — удобный способ задания функции от направления трехмерного вектора (т. е. функции, зависящей только от направления, а не от длины вектора).

Построим тогда следующую карту (обычно ее делают небольшого размера, например, 32×32): каждому вектору (*s*, *t*, *r*) сопоставим его нормированное значение — $\frac{(s,t,r)}{\sqrt{s^2+t^2+r^2}}$.

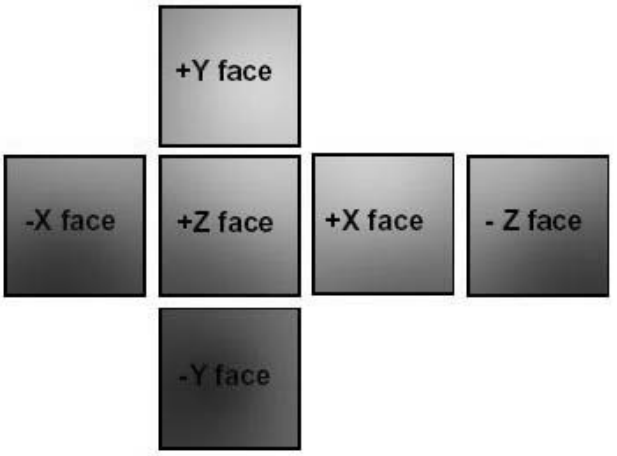


Рис. 8.4. Текстуры нормирующей кубической карты

Если значения вектора l в вершинах задавать как текстурные координаты для такой кубической карты (она называется нормирующей или нормализирующей), то в результате операции текстурирования мы будем получать единичный трехмерный вектор.

Однако поскольку на выходе операции текстурирования мы всегда получаем цвет (3- или 4-компонентный), то для построения такой карты необходим способ кодирования почти единичных (интерполяция существенно не влияет на значение) векторов в RGB-цвета.

Сопоставим вектору (x, y, z), каждая компонента которого лежит на отрезке $[-1, 1]$, цвет (r, g, b) по следующему правилу:

$$\begin{aligned} r &= (x + 1)/2; \\ g &= (y + 1)/2; \\ b &= (z + 1)/2. \end{aligned} \tag{8.2}$$

Полученный при этом вектор (r, g, b) будет содержаться в кубе $[-1, 1]^3$ и являться допустимым значением цвета (в реальных приложениях каждая из этих компонент должна быть умножена на 255 для перевода в 8-разрядный диапазон значений).

Обратите внимание, что поскольку для единичного вектора $(x^2 + y^2 + z^2 = 1)$ все его координаты принадлежат отрезку $[-1, 1]$, то формула (8.2) пригодна для представления любого такого вектора.

По вычисленному таким образом RGB-представлению вектора можно легко восстановить его исходные компоненты:

$$\begin{aligned} x &= 2r - 1; \\ y &= 2g - 1; \\ z &= 2b - 1. \end{aligned} \tag{8.3}$$

Листинг 8.1 содержит процедуру построения нормирующей кубической карты.

Листинг 8.1. Построение нормирующей кубической карты

```
static    void    getCubeVector ( int side, int cubeSize,
                                  int x, int y, Vector3D& v )
{
    float    s  = ((float) x + 0.5f) / (float) cubeSize;
    float    t  = ((float) y + 0.5f) / (float) cubeSize;
    float    sc = 2*s - 1;
    float    tc = 2*t - 1;
    switch ( side )
```

```
    {
        case 0:
            v = Vector3D ( 1, -tc, -sc );
            break;
        case 1:
            v = Vector3D ( -1, -tc, sc );
            break;
        case 2:
            v = Vector3D ( sc, 1, tc );
            break;
        case 3:
            v = Vector3D ( sc, -1, -tc );
            break;

        case 4:
            v = Vector3D ( sc, -tc, 1 );
            break;
        case 5:
            v = Vector3D ( -sc, -tc, -1 );
            break;
    }
    v.normalize ();
}
unsigned    createNormalizationCubemap ( int cubeSize )
{
    Vector3D   v;
    byte     * pixels = (byte *) malloc ( 3 * cubeSize * cubeSize );
    if ( pixels == NULL )
        return 0;
    unsigned    textureId;
    glGenTextures   ( 1, &textureId );
    glEnable        ( GL_TEXTURE_CUBE_MAP_ARB );
    glBindTexture   ( GL_TEXTURE_CUBE_MAP_ARB, textureId );
    glPixelStorei   ( GL_UNPACK_ALIGNMENT, 1 );
    glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_S,
                      GL_CLAMP_TO_EDGE );
    glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_T,
                      GL_CLAMP_TO_EDGE );
```

```
    glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_R,
                      GL_CLAMP_TO_EDGE );
    for ( int side = 0; side < 6; side++ )
    {
        for ( int x = 0; x < cubeSize; x++ )
            for ( int y = 0; y < cubeSize; y++ )
            {
                int    offs = 3 * (y * cubeSize + x);
                getCubeVector ( side, cubeSize, x, y, v );
                pixels [offs    ] = 128 + 127 * v.x;
                pixels [offs + 1] = 128 + 127 * v.y;
                pixels [offs + 2] = 128 + 127 * v.z;
            }

         glTexImage2D ( GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB + side, 0,
                        GL_RGB, cubeSize, cubeSize, 0, GL_RGB,
                        GL_UNSIGNED_BYTE, pixels );
    }
    free ( pixels );
    glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_MAG_FILTER,
                      GL_LINEAR );
    glTexParameteri ( GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_MIN_FILTER,
                      GL_LINEAR );
    glDisable ( GL_TEXTURE_CUBE_MAP_ARB );
    return textureId;
}
```

Функция `getCubeVector` по позиции тексела на заданной грани строит соответствующий единичный вектор. Сама же функция `createNormalizationCubemap` создает кубическую текстурную карты и загружает каждую из ее граней. Обе эти функции входят в библиотеку libTexture, содержащуюся на прилагаемом к книге компакт-диске.

Теперь, если в вершинах грани задать нормированные значения вектора направления на источник света l в виде текстурных координат для нормирующей кубической карты, то после индексации проинтерполированного вектора l в нормирующую текстуру на выходе для каждого пиксела грани мы получим нормированное значение, соответствующее данному пикселу. Правда оно будет иметь вид RGB-компонент согласно формуле (8.2).

Нормаль для каждого пиксела тоже можно задать в виде текстуры, при этом каждому значению единичного вектора нормали сопоставляется определенный RGB-цвет (рис. 8.5). Такие текстуры называются картами нормалей (*bumptmaps*).

Обычно считается, что грань расположена параллельно плоскости *Oxy*, поэтому неискаженному вектору нормали (0, 0, 1) будет соответствовать цвет (0,5, 0,5, 1), имеющий голубоватый оттенок (рис. 8.6).

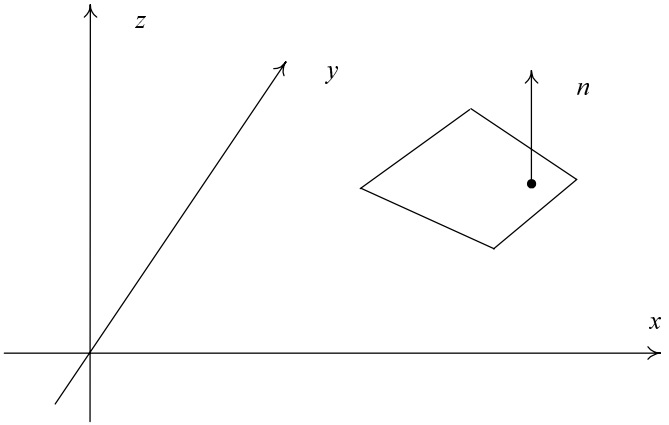


Рис. 8.5. Стандартная ориентация для карт нормали

В большинстве случаев карта нормалей содержит значения, не сильно отличающиеся от (0, 0, 1), и имеет голубовато-розоватый оттенок.

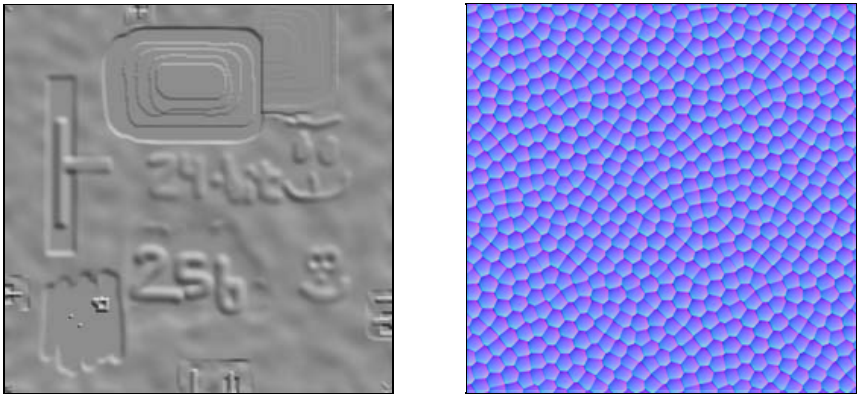


Рис. 8.6. Примеры карт нормалей

Теперь для каждого пиксела грани у нас есть два RGB-значения, каждое из которых соответствует определенному единичному вектору (l и n). Для получения попиксельного диффузного освещения по уравнению (8.1) нам надо вычислить скалярное произведение векторов пространства, соответствующих этим цветам.

Обратите внимание, что нас интересует настоящее скалярное произведение векторов, соответствующих цветам. Поэтому сначала надо для каждого из векторов по формулам (8.3) найти их исходные компоненты, а затем по ним построить скалярное произведение. Следовательно, интересующее нас скалярное произведение в терминах цветовых компонент записывается так:

$$(l, n) = 4(r_l - 0{,}5)(r_n - 0{,}5) + 4(g_l - 0{,}5)(g_n - 0{,}5) + 4(b_l - 0{,}5)(b_n - 0{,}5). \quad (8.4)$$

Для вычисления по этой формуле можно воспользоваться расширением ARB_texture_env_dot3. Оно вводит две новые операции для операндов `Arg0` и `Arg1`: `GL_DOT3_RGB_ARB` и `GL_DOT3_RGBA_ARB`.

Для обеих операций рассчитывается скалярное произведение операндов `Arg0` и `Arg1` по формуле (8.4) и значение, нормированное по величине к интервалу [0, 1] записывается либо в три (`GL_DOT3_RGB_ARB`), либо в четыре компоненты (`GL_DOT3_RGBA_ARB`) результата.

Обратите внимание, что значение скалярного произведения здесь приводится к отрезку [0, 1], а поскольку для двух единичных векторов оно никогда не превышает единицы, то и на выходе мы получаем величину, равную max[0, (l, n)], что и требуется в нашей модели освещенности (8.1).

Рассмотрим теперь, как можно реализовать простейшее попиксельное освещение с использованием нормирующих кубических карт и расширения ARB_texture_env_dot3. Схема формирования диффузного освещения представлена на рис. 8.7.

За счет мультитекстурирования выражение max[(l, n), 0] можно вычислить за один проход. В нулевом текстурном блоке расположим текстуру, задающую карту нормалей (*bumpMap*). В качестве режима наложения выберем `GL_REPLACE`. Текстурными координатами для него будут координаты текстуры с нормалями. В первом текстурном блоке разместим нормирующую кубическую карту (*normCubeMap*). Режимом наложения для нее является `GL_DOT3_RGB_ARB`. В качестве текстурных координат выступает нормированный вектор направления на источник света.

Аргументом `Arg0` будет `GL_TEXTURE`, а `Arg1` — `GL_PREVIOUS_ARB` (в данном случае это значения из предыдущего текстурного блока, т. е. значения вектора нормали).

В листинге 8.2 приводится фрагмент кода, осуществляющий настройку OpenGL для вычисления попиксельного диффузного освещения.

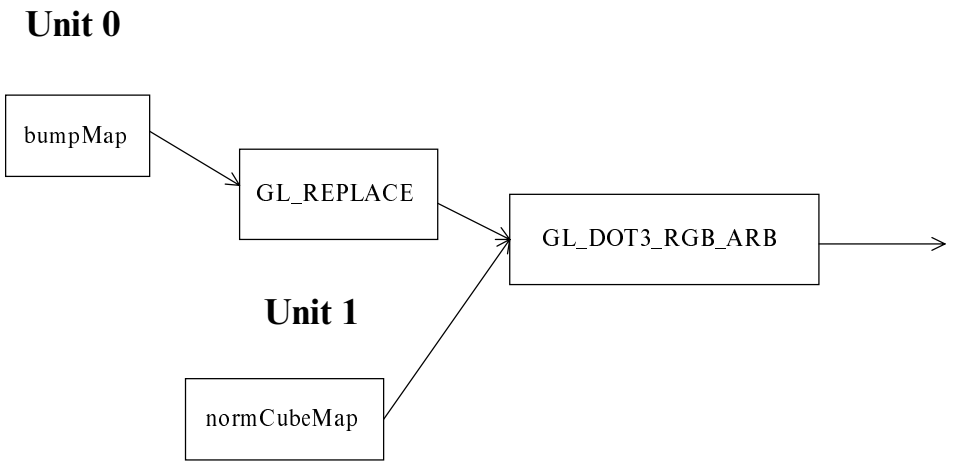


Рис. 8.7. Реализация диффузного освещения

Листинг 8.2. Пример вычисления попиксельного диффузного освещения

```
          // bind bump (normal) map to texture unit 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glBindTexture      ( GL_TEXTURE_2D, bumpMap );
glEnable           ( GL_TEXTURE_2D );
glTexEnvi  ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_ARB );
glTexEnvi  ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,  GL_TEXTURE );
glTexEnvi  ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,  GL_REPLACE );
          // bind normalization cube map to texture unit 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glBindTexture      ( GL_TEXTURE_CUBE_MAP_ARB, normCubeMap );
glEnable           ( GL_TEXTURE_CUBE_MAP_ARB );
glTexEnvi( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_ARB  );
glTexEnvi( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,  GL_TEXTURE      );
glTexEnvi( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,  GL_PREVIOUS_ARB );
glTexEnvi( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,  GL_DOT3_RGB_ARB );
```

На рис. 8.8 представлены изображения, полученные в результате выполнения примера.

За счет добавления второго прохода можно реализовать более сложную модель освещения:

$$I = C \cdot \max[(l, n), 0].$$

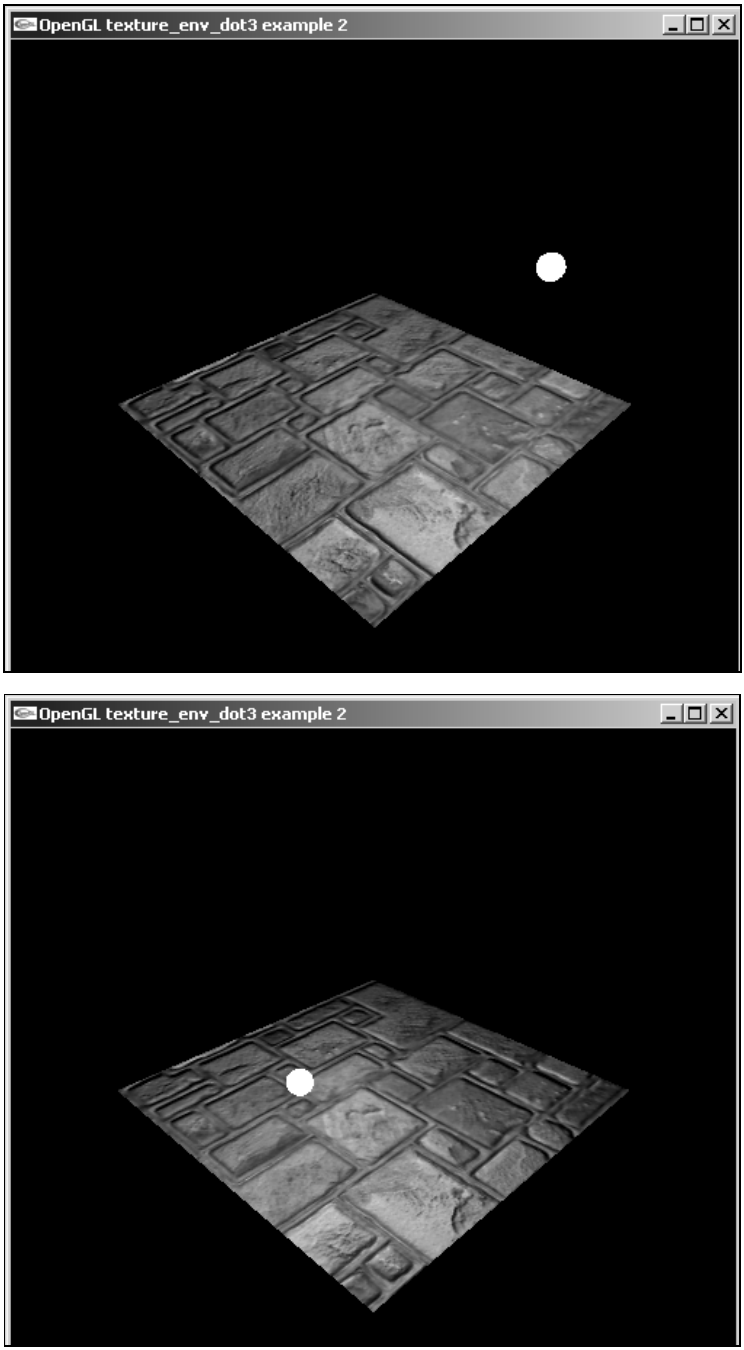


Рис. 8.8. Результаты применения попиксельного диффузного освещения

Здесь параметр *C* соответствует собственному цвету пиксела (т. е. является еще одной текстурой (*decalMap*), который модулируется освещенностью данного пиксела.

Для получения этого эффекта достаточно повторно вывести грань с текстурой *C* и режимом смешения (*blending*) (`GL_DST_COLOR`, `GL_ZERO`). Соответствующий исходный код содержится на прилагаемом к книге компакт-диске.

В случае, когда текстурных блоков больше двух, реализация возможна и за один проход (рис. 8.9). Листинг 8.3 содержит соответствующую программу.

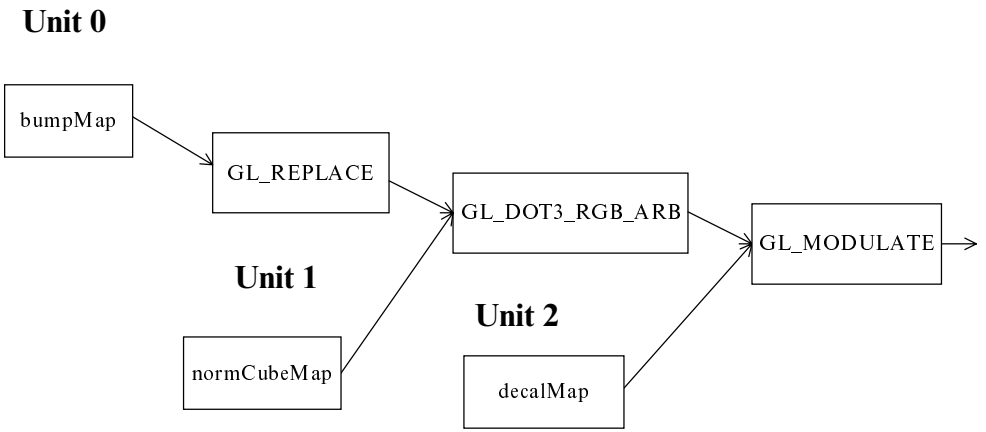


Рис. 8.9. Реализация диффузного освещения с текстурой

Листинг 8.3. Реализация попиксельного диффузного освещения с дополнительной текстурой

```
//
// Sample to to per-pixel dot3 bumpmapping in OpenGL
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
Vector3D    eye   ( 2, 2, 2 );          // camera position
Vector3D    light ( 0.5, 0, 1 );        // light position
```

```
unsigned    normCubeMap;                // normalization cubemap id
unsigned    bumpMap;                    // normal map
unsigned    decalMap;                   // decal (diffuse) texture
float       angle = 0;
Vector3D    v  [4];
Vector2D    t0 [4];
Vector3D    t1 [4];
void init ()
{
    glClearColor ( 0, 0, 0, 1 );
    glEnable     ( GL_DEPTH_TEST );
    glDepthFunc  ( GL_LEQUAL );
    glHint ( GL_POLYGON_SMOOTH_HINT,         GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}
void display ()
{
    int    i;
    glClear   ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glColor3f ( 1, 1, 1 );

                                        // bind bump (normal) map to
                                        // texture unit 0
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, bumpMap );
                                      // draw light source
    glMatrixMode ( GL_MODELVIEW );      glPushMatrix ();
    glTranslatef       ( light.x, light.y, light.z );
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glTexEnvi          ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                         GL_REPLACE );
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glDisable          ( GL_TEXTURE_CUBE_MAP_ARB );
    glActiveTextureARB ( GL_TEXTURE2_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glColor4f          ( 1, 1, 1, 1 );
```

```
    glutSolidSphere    ( 0.05f, 15, 15 );
    glPopMatrix        ();
                                                  // setup environment
                                                  // for dot3
    glTexEnvi  ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_ARB );
    glTexEnvi  ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,  GL_TEXTURE );
    glTexEnvi  ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,  GL_REPLACE );

                                                  // bind normalization
                                                  // cube map to
                                                  // texture unit 1
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnable           ( GL_TEXTURE_CUBE_MAP_ARB );
    glBindTexture      ( GL_TEXTURE_CUBE_MAP_ARB, normCubeMap );
                                                  // setup environment
                                                  // for dot3
    glTexEnvi( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_ARB );
    glTexEnvi( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,  GL_TEXTURE );
    glTexEnvi( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,  GL_PREVIOUS_ARB );
    glTexEnvi( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,  GL_DOT3_RGB_ARB );
                                                  // setup decal
                                                  // texture
    glActiveTextureARB ( GL_TEXTURE2_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, decalMap );
                                                  // setup environment
                                                  // for decal texture
    glTexEnvi( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_ARB );
    glTexEnvi( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,  GL_TEXTURE );
    glTexEnvi( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,  GL_PREVIOUS_ARB );
    glTexEnvi( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,  GL_MODULATE );
                                                   // compute texture
                                                   // coordinates for
                                                   // normalization
                                                   // cube map
    for ( i = 0; i < 4; i++ )
    {
        t1 [i] = light - v [i];
```

```
        t1 [i].normalize ();
    }
                                                    // now draw quad
    glBegin ( GL_QUADS );
    for ( i = 0; i < 4; i++ )
    {
        glMultiTexCoord2fv ( GL_TEXTURE0_ARB, t0 [i] );
        glMultiTexCoord3fv ( GL_TEXTURE1_ARB, t1 [i] );
        glMultiTexCoord2fv ( GL_TEXTURE2_ARB, t0 [i] );
        glVertex3fv        ( v [i]                   );
    }
    glEnd   ();
    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 key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
        exit ( 0 );
    if ( key == 'p' || key == 'P' )                 // screenshot
        saveScreenShot ( "dot3-3.tga" );
}
void    animate ()
{
    angle  = 0.004f * glutGet ( GLUT_ELAPSED_TIME );
    light.x = cos ( angle );
    light.y = sin ( angle );
```

```
    light.z = 1 + 0.3 * sin ( angle / 3 );
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                        // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                        // create window
    int    win = glutCreateWindow (
                         "OpenGL texture_env_dot3 example 3" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    if ( !isExtensionSupported ( "GL_ARB_texture_env_combine" ) )
    {
        printf ( "ARB_texture_env_combine NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_ARB_texture_cube_map" ) )
    {
        printf ( "ARB_texture_cube_map NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_ARB_texture_env_dot3" ) )
    {
        printf ( "ARB_texture_env_dot3 NOT supported.\n" );
        return 1;
    }
    init           ();
    initExtensions ();
                                  // setup vertices
    v [0] = Vector3D ( -1, -1, 0 );
    v [1] = Vector3D (  1, -1, 0 );
    v [2] = Vector3D (  1,  1, 0 );
    v [3] = Vector3D ( -1,  1, 0 );
```

```
                                                    // setup
                                                    // texture coordinates
                                                    // for bump map
    t0 [0] = Vector2D ( 0, 0 );
    t0 [1] = Vector2D ( 1, 0 );
    t0 [2] = Vector2D ( 1, 1 );
    t0 [3] = Vector2D ( 0, 1 );
    bumpMap     = createTexture2D ( true,
                            "../Textures/Bumpmaps/FieldStone-DOT3.tga" );
    decalMap    = createTexture2D  ( true,
                            "../Textures/FieldStone.tga" );
    normCubeMap = createNormalizationCubemap ( 32 );
    glutMainLoop ();
    return 0;
}
```

Работа с произвольно ориентированными гранями

Рассмотрим теперь способ вычисления попиксельной освещенности для произвольно ориентированных граней.

Ранее мы считали, что грань расположена параллельно координатной плоскости *Oxy*, и карту нормалей строили соответствующим образом (полагая, что стандартным значением вектора нормали является вектор (0, 0, 1)).

Пусть теперь у нас имеется произвольно ориентированная грань со стандартным вектором нормали *n* (рис. 8.10).

Можно построить специальную карту нормалей, учитывающую ориентацию данной грани. Однако такой подход неудобен, т. к. фактически для каждой грани нужно строить свою карту нормалей (пропадает возможность переиспользования карт), а в случае изменения ориентации грани (например, при повороте объекта) придется заново перестраивать всю текстурную карту.

Гораздо удобнее (как и для обычных текстур) задавать карту нормалей в некоторой специальной системе координат так, чтобы ориентация грани влияла только на эту систему, а сама карта оставалась бы неизменной.

Будем считать, что для нашей произвольно ориентированной грани кроме вектора *n* задан также касательный вектор *t*, который также является единичным. Обратите внимание, что эти векторы ортогональны.

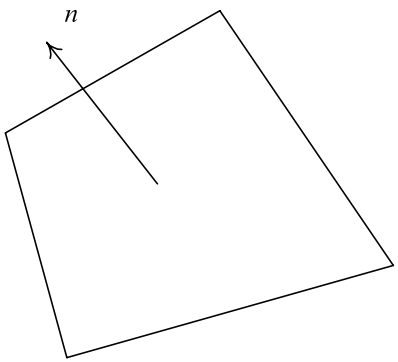


Рис. 8.10. Произвольноориентированная грань

По этим двум векторам легко построить третий, используя для этого векторное произведение: $b = [t, n]$. Его обычно называют *бинормалью* (рис. 8.11).

Все три построенных вектора единичны и попарно ортогональны, из чего следует, что они образуют ортонормированный базис в трехмерном пространстве, т. е. задают с точностью до переноса декартову систему координат.

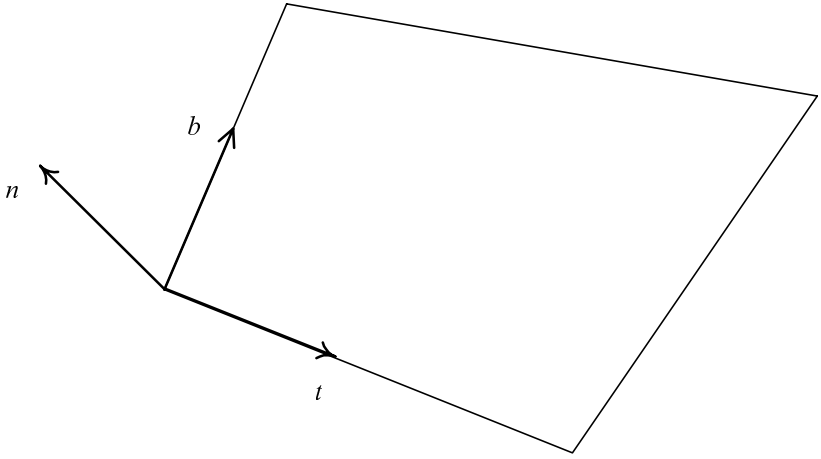


Рис. 8.11. Базис в касательном пространстве

По этим трем векторам (t, b и n) можно построить новую систему координат (называемую касательной, *tangent* или TBN).

Для получения матрицы преобразования в эту систему координат (будем считать, что ее начало совпадает с началом системы, в которой задана рас-

сматриваемая грань) достаточно взять каждый из этих векторов и записать в виде строки матрицы:

$$M = \begin{pmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{pmatrix}.$$

Обратите внимание, что M будет ортогональной, поскольку базисные векторы ортонормированы, поэтому матрица, обратная к ней, равна транспонированной M ($M^{-1} = M^T$).

Пусть p — вектор в исходной системе координат, тогда его представление в новой системе будет задаваться формулой

$$M \cdot p = ((t, p), (b, p), (n, p)).$$

Не забывайте, что вектор нормали к грани в касательной системе координат всегда равен (0, 0, 1), т. е. совпадает со значением в рассмотренном ранее частном случае. Поэтому очень удобно вычисление освещенности проводить именно в касательном пространстве. Так как карта нормалей уже задана в нем, осталось только задать вектор l в этом же пространстве.

Поскольку l — это направление на источник света (т. е. разность двух векторов, определяющих положение источника света и освещаемой точки), то в качестве начала координат для касательной системы можно выбрать любую точку (так как сдвиг системы координат не изменяет направление от одной точки к другой).

Таким образом, вывод произвольно ориентированной грани с попиксельным вычислением диффузной освещенности отличается от рассмотренного простейшего случая только тем, что значения вектора l в вершинах надо переводить в касательное пространство данной грани умножением на соответствующую текущей грани матрицу преобразования M.

В случае, когда у нас имеется сложный объект, состоящий из многих граней, для каждой из них нужно хранить не только вектор нормали n, но и векторы t и b. Это иллюстрирует листинг 8.4.

Листинг 8.4. Структура, описывающая данные в вершине

```
struct    Vertex
{
    Vector3D  pos;           // position of vertex
    Vector2D  tex;           // texture coordinates
    Vector3D  n;             // unit normal
```

```
    Vector3D  t, b;          // tangent and binormal
                             // map vector to tangent (TBN) space
    Vector3D   mapToTangentSpace ( const Vector3D& v ) const
    {
        return Vector3D ( v & t, v & b, v & n );
    }
};
```

Листинг 8.5 содержит код примера объекта (тора), построенного таким образом.

Листинг 8.5. Построение изображения диффузно-освещенного тора

```
//
// Sample to show how to per-pixel dot3 bumpmapping in OpenGL
//
#include    "libExt.h"
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Torus.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
Vector3D    eye   ( 7, 5, 7 );          // camera position
Vector3D    light ( 5, 0, 4 );          // light position
unsigned    normCubeMap;                // normalization cubemap id
unsigned    bumpMap;                    // normal map
unsigned    decalMap;                   // decal (diffuse) texture
float       angle = 0;
Torus       torus ( 1, 3, 30, 30 );
void init ()
{
    glClearColor ( 0, 0, 0, 1 );
    glEnable     ( GL_DEPTH_TEST );
    glDepthFunc  ( GL_LEQUAL );
    glHint ( GL_POLYGON_SMOOTH_HINT,         GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}
```

```
void display ()
{
    int i;

    glClear   ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glColor3f ( 1, 1, 1 );
                                        // draw light source
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();
    glTranslatef       ( light.x, light.y, light.z );
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glDisable          ( GL_TEXTURE_CUBE_MAP_ARB );
    glActiveTextureARB ( GL_TEXTURE2_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glColor4f          ( 1, 1, 1, 1 );
    glutSolidSphere    ( 0.1f, 15, 15 );
    glPopMatrix        ();
                                          // bind bump (normal) map to
                                          // texture unit 0
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, bumpMap );
                                          // setup environment for dot3
    glTexEnvi ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_ARB );
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,  GL_TEXTURE );
    glTexEnvi ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,  GL_REPLACE );
                                        // bind normalization cube
                                          // map to texture unit 1
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnable           ( GL_TEXTURE_CUBE_MAP_ARB );
    glBindTexture      ( GL_TEXTURE_CUBE_MAP_ARB, normCubeMap );
                                          // setup environment for dot3
    glTexEnvi ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_ARB );
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,  GL_TEXTURE );
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,  GL_PREVIOUS_ARB );
    glTexEnvi ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,  GL_DOT3_RGB_ARB );
                                          // setup decal texture
```

```
    glActiveTextureARB ( GL_TEXTURE2_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, decalMap );
                                          // setup environment for
                                          // decal texture
    glTexEnvi ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_ARB );
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE0_RGB_ARB,  GL_TEXTURE );
    glTexEnvi ( GL_TEXTURE_ENV, GL_SOURCE1_RGB_ARB,  GL_PREVIOUS_ARB );
    glTexEnvi ( GL_TEXTURE_ENV, GL_COMBINE_RGB_ARB,  GL_MODULATE );
                                         // now draw torus
    torus.calcLightVectors ( light );
    torus.draw             ();
    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 key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
        exit ( 0 );
    if ( key == 'p' || key == 'P' )                // saving screenshots
        saveScreenShot ( "dot3-4.tga" );
}
void    animate ()
{
    angle  = 0.004f * glutGet ( GLUT_ELAPSED_TIME );
    light.x = 3.5*cos ( angle );
    light.y = sin ( angle );
```

```
    light.z = 4 + sin ( angle / 3 );
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                                // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                                // create window
    int win = glutCreateWindow ( "OpenGL texture_env_dot3 example 5" );
                                // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    if ( !isExtensionSupported ( "GL_ARB_texture_env_combine" ) )
    {
        printf ( "ARB_texture_env_combine NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_ARB_texture_cube_map" ) )
    {
        printf ( "ARB_texture_cube_map NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_ARB_texture_env_dot3" ) )
    {
        printf ( "ARB_texture_env_dot3 NOT supported.\n" );
        return 1;
    }
    init           ();
    initExtensions ();
    bumpMap     = createNormalMapFromHeightMap ( false,
                          "../Textures/Bumpmaps/16tile01Bump.png", 5 );
    decalMap    = createTexture2D              ( true,
                           "../Textures/Bumpmaps/16tile01.jpg" );
```

```
    normCubeMap = createNormalizationCubemap   ( 32 );
    glutMainLoop ();
    return 0;
}
```

На рис. 8.12 приводится результат работы программы — изображение объекта с диффузным попиксельным освещением.

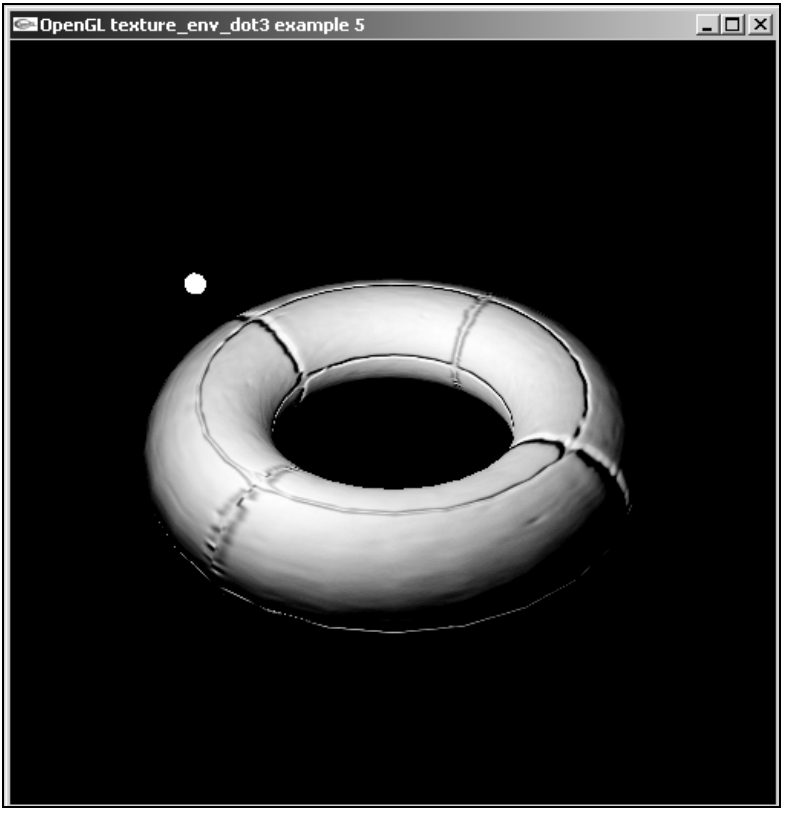


Рис. 8.12. Тор с наложенной на него картой нормалей

Карты высот и работа с ними

Обычно при применении карты нормалей считается, что реальная поверхность слегка отличается от выводимых граней. Это отличие заключается в небольших искажениях поверхности грани, причем по абсолютной величине

отклонения (называемые микрорельефом) настолько малы, что ими можно пренебречь. Это значит, что они играют роль только при нахождении вектора нормали.

Гораздо проще (и удобнее) задать микрорельеф при помощи так называемой карты высот H (*heightmap*), которая обычно представляет собой изображение (текстуру) в оттенках серого цвета, где яркость точки соответствует величине ее подъема вдоль направления нормали (рис. 8.13).

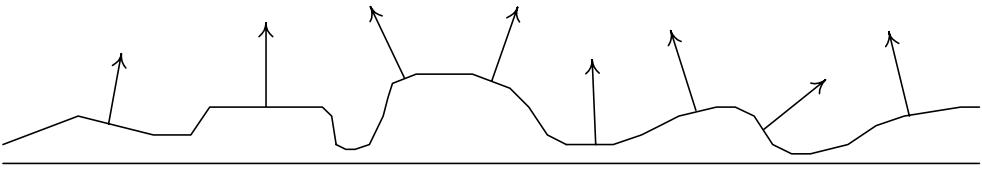


Рис. 8.13. Микрорельеф, задаваемый картой высот

Фактически можно считать, что микрорельеф задается функцией $H(s, t)$, причем справедливо соотношение $H << 1$. В силу этого грань может выводиться как плоская, т. е. влиянием микрорельефа на растеризацию можно пренебречь.

Однако при расчете освещения большую роль играет не столько положение точки, сколько направление вектора нормали. Вектор нормали определяется градиентом функции H, и его значения уже не всегда будут малыми:

$$n' = (dH/ds,\ dH/dt,\ 1).$$

Здесь через n' обозначен ненормированный искаженный вектор нормали.

Поскольку в нашем случае карта высот — дискретная функция, то вместо вычисления производных функции H мы будем использовать их разностные аналоги.

$$n_{ij} = (h_{i+1,\,j} - h_{i,\,j},\ h_{i,\,j+1} - h_{i,\,j},\ 1). \qquad (8.5)$$

Обычно гораздо удобнее задавать микрорельеф при помощи именно карты высот, а потом по ней строить карту нормали. Дело в том, что карта высот интуитивно понятнее, чем карта нормалей. Кроме того, на карту нормалей накладывается условие нормировки. Поэтому художнику гораздо проще работать именно с картами высот. На рис. 8.14 приведен пример карты высот.

Рассмотрим теперь, каким образом по карте высот строится карта нормалей.

Для управления степенью неровности, задаваемой картой высот, в формулу (8.5) нужно ввести некоторый коэффициент:

$$n'_{ij} = ((h_{i+1,\,j} - h_{i,\,j}) \cdot scale,\ (h_{i,\,j+1} - h_{i,\,j}) \cdot scale,\ 1). \qquad (8.6)$$

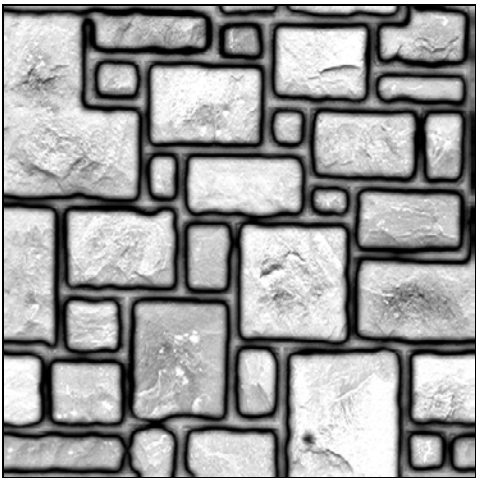


Рис. 8.14. Пример карты высот

В (8.6) величина *scale* позволяет управлять степенью "неровности", чем она больше, тем больше значения в карте нормалей будут отличаться от (0, 0, 1) после нормализации.

Несмотря на то, что для задания карты нормалей достаточно трехкомпонентной (RGB) текстуры, часто используется четырехкомпонентная RGBA-текстура, при этом для текстур, непосредственно построенных из карт высот, в альфа-канал карты нормалей заносится значение 255 (в принципе там можно хранить любую дополнительную информацию).

В листинге 8.6 приводится пример кода, осуществляющего построение карты нормалей по карте высот. При этом карта высот задается однокомпонентной (один байт на пиксел) текстурой. Сама эта функция может быть найдена в библиотеке libTexture.

Листинг 8.6. Построение карты нормалей по карте высот

```
Texture * convertHeight2NormalMap ( byte * pixels, int width, int height,
                                    float scale )
{
    float     oneOver255 = 1.0 / 255.0;
    Texture * normalMap  = new Texture ( width, height, 4 );
    if ( normalMap == NULL )
         return NULL;
    byte    * out = normalMap -> getData ();
    if ( out == NULL )
```

```
    {
        delete normalMap;
        return NULL;
    }

    int    offs = 0;                    // offset to normalMap
    for ( int i = 0; i < height; i++ )
        for ( int j = 0; j < width; j++ )
        {
            float c  = pixels [i*width + j]              * oneOver255;
            float cx = pixels [i*width + (j+1)%width]    * oneOver255;
            float cy = pixels [((i+1)%height)*width + j] * oneOver255;
                                    // find derivatives
            float    dx = (c - cx) * scale;
            float    dy = (c - cy) * scale;
                                    // normalize
            float    len = (float) sqrt ( dx*dx + dy*dy + 1 );
                                    // get normal
            float    nx = dy   / len;
            float    ny = -dx  / len;
            float    nz = 1.0f / len;
                                     // now convert to color and
                                     // store in map
            out [offs    ] = (byte)(128 + 127*nx);
            out [offs + 1] = (byte)(128 + 127*ny);
            out [offs + 2] = (byte)(128 + 127*nz);
            out [offs + 3] = 255;
            offs += 4;
        }
    return normalMap;
}
```

Следующий пример кода демонстрирует загрузку карты нормалей из карты высот.

```
bumpMap     = createNormalMapFromHeightMap ( false,
                    "../Textures/Bumpmaps/16tile01Bump.png", 5 );
```

Иногда карту высот перед построением карты нормалей сглаживают, например, с помощью фильтра 3×3 (для каждого пиксела в качестве значения

берется среднее из него и всех 8 соседей). Это позволяет получить более гладкие карты нормалей.

Одним из важных свойств карт нормалей является то, что все они имеют единичную длину. Нарушение этого свойства может привести к появлению нежелательных артефактов.

Одной из операций, которая может нарушить это свойство, является операция построения промежуточных уровней в процедуре пирамидального фильтрования (`gluBuild2DMipmap`), поскольку при этом происходит усреднение значений (по блоку 2×2 тексела строится новый тексел, значением которого является среднее по всему блоку). В результате получившийся путем усреднения вектор, скорее всего, уже не будет единичным.

Поэтому для корректного выполнения пирамидального фильтрования промежуточные уровни необходимо строить вручную, каждый раз производя операцию нормализации всех текселов очередного уровня.

Листинг 8.7 содержит процедуру, строящую очередной уровень для пирамидального фильтрования.

Листинг 8.7. Построение промежуточного уровня для карты нормалей

```
Texture * downSampleNormaMap ( Texture * old, int w2, int h2 )
{
    if ( old == NULL )
        return NULL;
    Texture * map = new Texture ( w2, h2, 4 );
    if ( map == NULL || map -> getData () == NULL )
    {
        delete map;
        return NULL;
    }
    float    oneOver127 = 1.0f / 127.0f;
    float    oneOver255 = 1.0f / 255.0f;
    byte  * in  = old -> getData   ();
    byte  * out = map -> getData   ();
    int     w   = old -> getWidth  ();
    int     h   = old -> getHeight ();
    float   v [3];                  // here we will store x, y, z
    for ( int i = 0; i < h; i += 2 )
        for ( int j = 0; j < w; j += 2 )
```

```
{
    int    iNext = (i+1) % h;
    int    jNext = (j+1) % w;
    float    mag00 = oneOver255 * in [3 + i*w + j];
    float    mag01 = oneOver255 * in [3 + i*w + jNext];
    float    mag10 = oneOver255 * in [3 + iNext*w + j];
    float    mag11 = oneOver255 * in [3 + iNext*w + jNext];
                          // sum up values for RGB components,
                          // converting to [-1,1]
    for ( int k = 0; k < 3; k++ )
    {
        v [k]  = mag00 * (oneOver127*in[k+i*w    +j]     - 1.0);
        v [k] += mag01 * (oneOver127*in[k+i*w    +jNext] - 1.0);
        v [k] += mag10 * (oneOver127*in[k+iNext*w+j]     - 1.0);
        v [k] += mag11 * (oneOver127*in[k+iNext*w+jNext] - 1.0);
    }
                          // compute length of (x,y,z)
    float    length = (float)sqrt ( v[0] * v[0] + v[1] * v [1] +
                                    v[2] * v[2] );
                          // check for degenerated case
    if ( length < EPS )
    {
        v [0] = 0;
        v [1] = 0;
        v [2] = 1;
    }
    else
    {
        v [0] /= length;
        v [1] /= length;
        v [2] /= length;
    }
    int    index = (i >> 1)*w2 + (j >> 1);
                           // pack normalized vector into
                           // color values
    out [index]     = (byte)(128 + 127*v[0]);
    out [index + 1] = (byte)(128 + 127*v[1]);
    out [index + 2] = (byte)(128 + 127*v[2]);
```

```
                                    // store averaged length as
                                    // alpha component
            length *= 0.25;         // since it was build on 2x2
                                    // summed values
            if ( length > 1.0f - EPS )
                out [index + 3] = 255;
            else
                out [index + 3] = (byte)(255 * length);
        }
    return map;
}
```

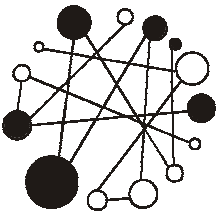
Процедура из библиотеки libTexture, которая по карте нормалей строит и загружает в графический ускоритель все промежуточные уровни, приводится в листинге 8.8.

Листинг 8.8. Загрузка карты нормалей вместе с промежуточными уровнями

```
void    loadNormalMap ( int target, bool mipmap, Texture * map )
{
    int       level  = 0;
    int       width  = map -> getWidth  ();
    int       height = map -> getHeight ();
    Texture * cur    = map;
    Texture * next   = NULL;
                                // load the original map
    glTexImage2D ( target, 0, map -> getNumComponents (), width, height,
                   0, map -> getFormat (), GL_UNSIGNED_BYTE,
                   map -> getData () );
    if ( !mipmap )
         return;
                                // downsample texture
    while ( width > 1 || height > 1 )
    {
         level++;
                                // compute new size for this level
        int    newWidth  = width  >> 1;
        int    newHeight = height >> 1;
```

```
        if ( newWidth < 1 )
            newWidth = 1;
        if ( newHeight < 1 )
            newHeight = 1;
        next = downSampleNormaMap ( cur, newWidth, newHeight );
        if ( next == NULL )
            return;
        glTexImage2D ( target, level, next -> getNumComponents (),
                       newWidth, newHeight, 0, next -> getFormat (),
                       GL_UNSIGNED_BYTE, next -> getData () );
                                // prepare for next iteration
        width  = newWidth;
        height = newHeight;
        if ( cur != map )       // release old temporary textures
            delete cur;
        cur = next;
    }
    if ( cur != map )           // remove last temporary
        delete cur;
}
```

Глава 9



Понятие register combiner. Расширение NV_register_combiners. Реализация с их помощью попиксельного диффузного и бликового освещения

Работа с register combiner

Начиная с графических ускорителей серии GeForce, компания NVIDIA предложила кроме стандартной для OpenGL, новую схему обработки данных (*pipeline*), основанную на так называемом механизме наложения регистров (*register combiner*), который обрабатывает данные гораздо эффективнее и хорошо подходит для целого ряда сложных операций, включая, например, расчет попиксельного диффузного и бликового (*specular*) освещения.

В новой модели все входные данные сначала помещаются в набор регистров, далее значения в регистрах проходят через один или несколько основных блоков обработки (*general combiner*), после чего они попадают в завершающий блок (*final combiner*), определяющий выходное значение цвета и альфа-канала (рис. 9.1).

Используемые при этом регистры представлены в табл. 9.1.

Таблица 9.1. *Регистры*

Название	Значение	Константа в OpenGL
`primary color`	Цвет фрагмента	`GL_PRIMARY_COLOR_NV`
`secondary color`	Дополнительный (вторичный) цвет	`GL_SECONDARY_COLOR_NV`
`texture0, texture1 ...`	Значение из соответствующего текстурного блока	`GL_TEXTURE0_ARB GL_TEXTURE1_ARB ...`

Таблица 9.1 *(окончание)*

Название	Значение	Константа в OpenGL
`spare0` `spare1`	Временные регистры, их начальные значения не определены	`GL_SPARE0` `GL_SPARE1`
`fog`	Степень затуманивания	`GL_FOG`

`constant color 0`
 	
`constant color 1`
	Задаваемые пользователем дополнительные цвета	`GL_CONSTANT_COLOR0_NV`
		`GL_CONSTANT_COLOR1_NV`

`zero`	Нулевой цвет (0, 0, 0, 0). Доступен только для чтения	`GL_ZERO`

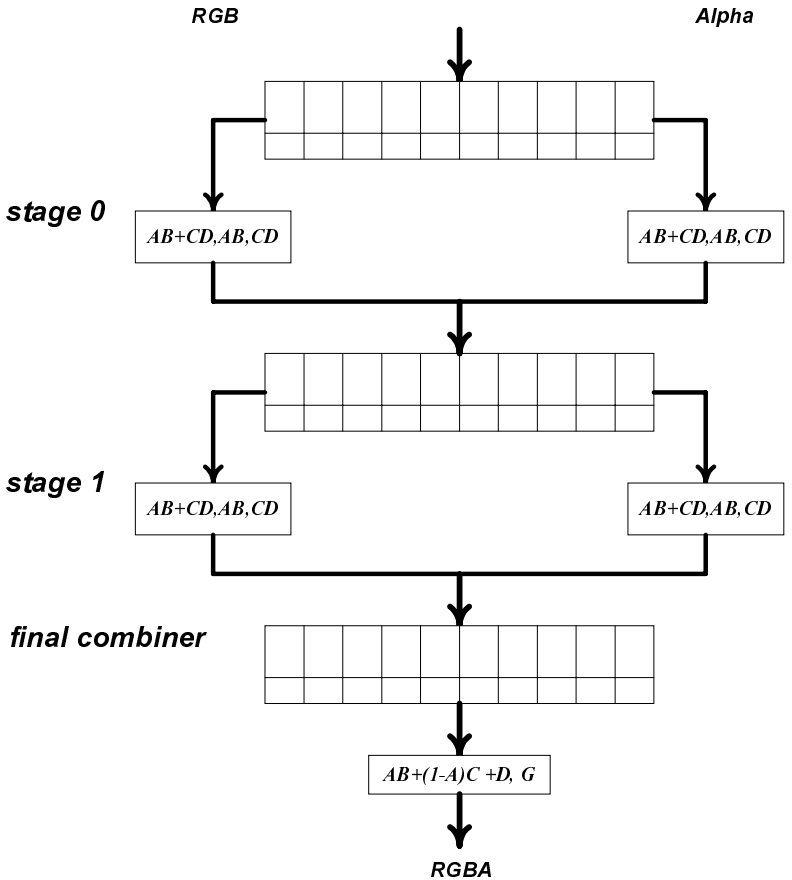


Рис. 9.1. Общая схема работы механизма register combiner

Все эти регистры содержат цветовую (RGB) и альфа-части (A), которые могут обрабатываться совершенно независимо.

Каждый блок general combiner работает следующим образом: на основе входных регистров из табл. 9.1 с использованием простых преобразований (*mapping*) строятся четыре внутренние переменные — *A*, *B*, *C* и *D*.

При этом для каждой из них (отдельно для RGB- и альфа-частей) задается, из какого входного регистра следует взять требуемое значение и какому преобразованию нужно подвергнуть его перед записью в соответствующую переменную (рис. 9.2).

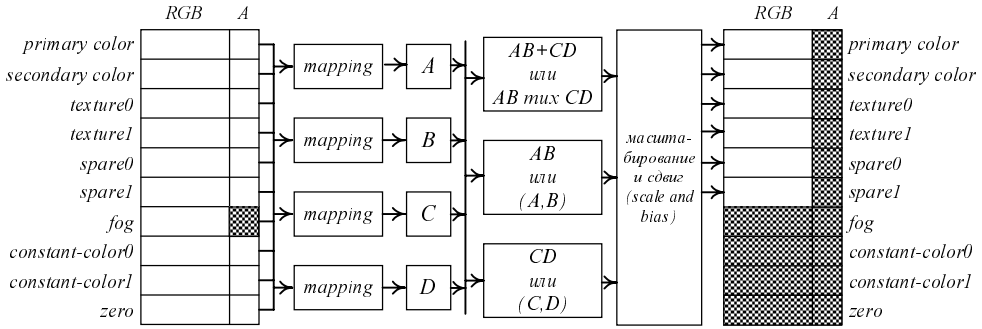


Рис. 9.2. Работа блока general combiner

На основе переменных *A*, *B*, *C* и *D* вычисляются три выходных RGBA-значения, которые (после масштабирования и сдвига) отсекаются по отрезку [−1, 1] и записываются в выходные регистры.

При этом соответствия между переменными, значения которых будут записываться, и выходными регистрами, в которые будет происходить запись, может задаваться отдельно как для цветовой, так и для альфа-части.

На рис. 9.3 и 9.4 приводятся схемы, показывающие как по входным переменным *A*, *B*, *C* и *D* вычисляются три выходных значения (RGB и альфа).

Используемая при этом операция mux вводится следующим образом: *a* mux *b* возвращает значение *a*, если альфа-часть регистра `spare0` превышает 0,5. В противном случае возвращается *b*. Обратите внимание, что по умолчанию альфа-часть регистра `spare0` инициализируется альфа-значением из нулевого текстурного блока `texture0`.

Операции, допустимые для получения значений внутренних переменных *A*, *B*, *C* и *D* по значениям соответствующих регистров, приведены в табл. 9.2 и проиллюстрированы на рис. 9.5.

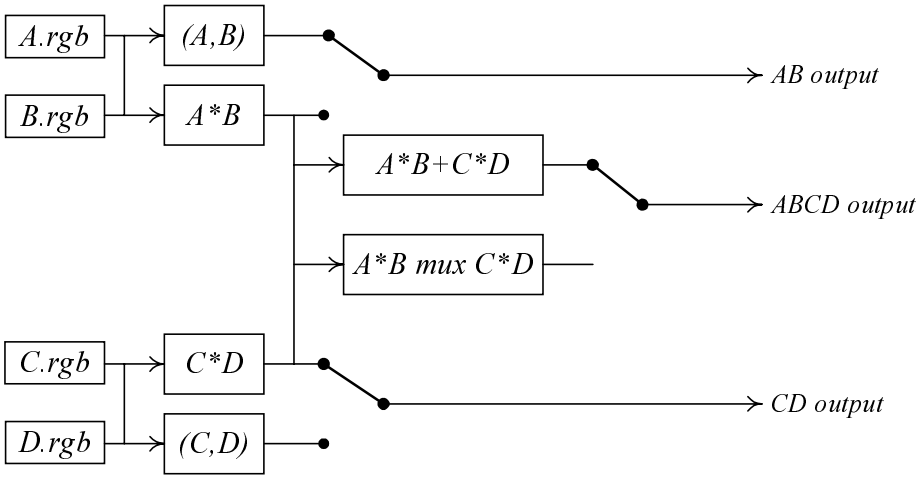


Рис. 9.3. Вычисление выходных значений для RGB-части

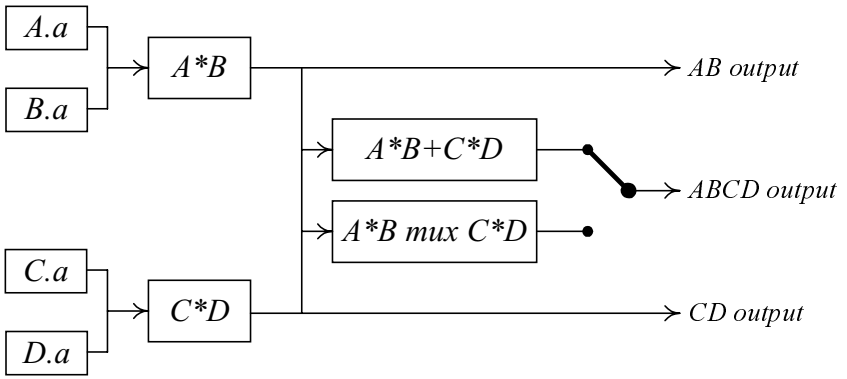


Рис. 9.4. Вычисление выходных значений для альфа-части

Таблица 9.2. *Допустимые операции*

Название	**Формула**	**Константа в OpenGL**
`unsigned indenity`	max(0, *x*)	`GL_UNSIGNED_IDENTITY_NV`
`unsigned invert`	1 – min(max(0, *x*))	`GL_UNSIGNED_INVERT_NV`
`expand normal`	2·max(0, *x*) – 1	`GL_EXPAND_NORMAL_NV`
`expand negate`	1 – 2 · max(0, *x*)	`GL_EXPAND_NEGATE_NV`

Таблица 9.2 *(окончание)*

Название	Формула	Константа в OpenGL
`half-bias normal`	max(0, *x*) – 0,5	`GL_HALF_BIAS_NORMAL_NV`
`half-bias negate`	0,5 – max (0, *x*)	`GL_HALF_BIAS_NEGATE_NV`
`signed identity`	*x*	`GL_SIGNED_IDENTITY_NV`
`signed negate`	–*x*	`GL_SIGNED_NEGATE_NV`

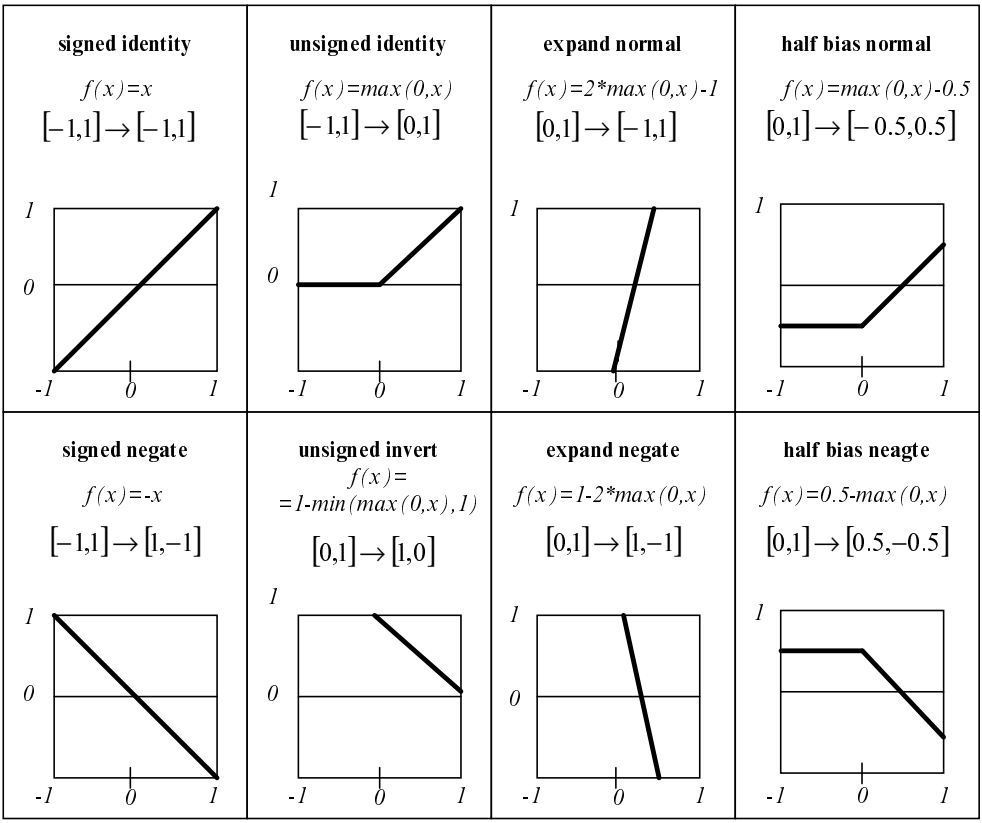


Рис. 9.5. Допустимые преобразования для блоков general combiner

Выходные значения (перед записью в регистры) подвергаются масштабированию (*scale*) и сдвигу (*bias*). Допустимые значения для этих операций приведены в табл. 9.3 и 9.4 соответственно.

Таблица 9.3. *Допустимые значения для масштабирования*

Константа в OpenGL	Множитель
GL_NONE	1
GL_SCALE_BY_TWO_NV	2
GL_SCALE_BY_FOUR_NV	4
GL_SCALE_BY_ONE_HALF_NV	0,5

Таблица 9.4. *Допустимые значения для сдвига*

Константа в OpenGL	Величина сдвига
GL_NONE	0
GL_BY_NEGATIVE_ONE_HALF_NV	–0,5

В результате на выходе получаются три RGB- и три альфа-значения. RGB-значения могут быть записаны в RGB-части регистров (доступных для записи), альфа-значения — как в альфа-, так и в RGB-часть (в этом случае получается цвет, все три компоненты которого равны между собой). Записанные в регистры значения являются входными для следующего блока register combiner. Любое из трех выходных значений может быть просто отброшено (т. е. никуда не записано).

Внутри каждого блока general combiner все вычисления выполняются в диапазоне [−1, 1]. Конфигурация и обработка RGB- и альфа-частей происходят полностью независимо.

Обычно блоки general combiner образуют цепочку: выход одного является входом для следующего, хотя это необязательно.

Заключительная обработка осуществляется блоком final combiner (рис. 9.6).

Блок final combiner вычисляет RGBA-значение, которое и будет выходным цветом. Знак операнда при вычислениях не учитывается. Все отрицательные значения на входе в final combiner преобразуются в ноль (рис. 9.7).

Для вычисления выходного RGB-значения final combiner содержит шесть внутренних переменных (A, B, C, D, E и F). Каждая из них является RGB-величиной и может быть построена на основе любого из входных регистров, причем как из RGB-, так и из альфа-части. Значение любой из входных частей может быть инвертировано.

Имеются также два псевдорегистра EF и spare0+secondary color. Первый псевдорегистр равен произведению переменных E и F, он может быть ис-

пользован в качестве исходного значения для любой из переменных `A`, `B`, `C` и `D`, второй также может быть входным значением для переменных `B`, `C` и `D`.

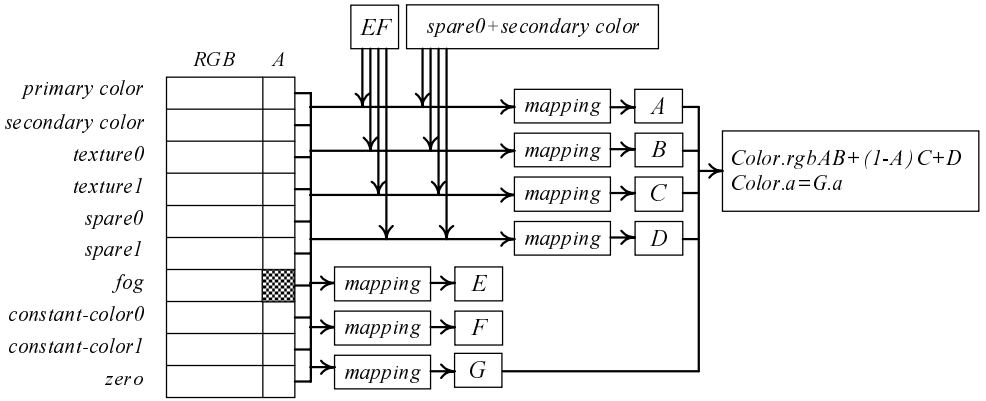


Рис. 9.6. Работа блока final combiner

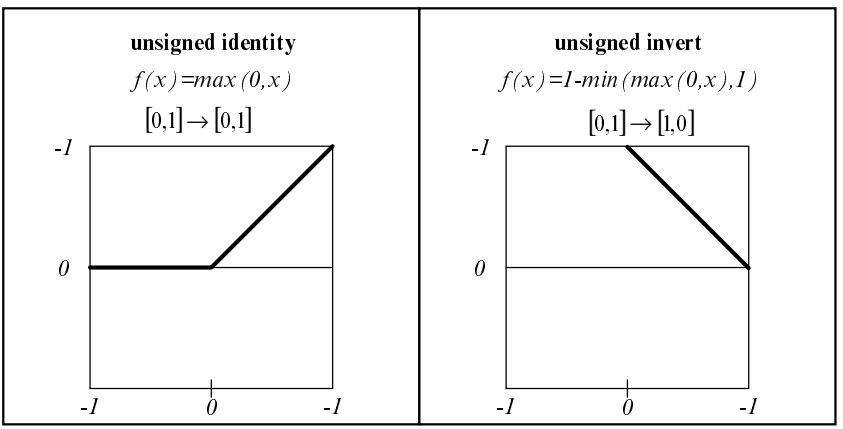


Рис. 9.7. Допустимые преобразования для блока final combiner

Для получения RGB-части выхода final combiner вычисляет значение `AB+(1-A)C+D`.

Значение седьмой переменной `G` определяет альфа-канал выходного RGBA-значения. Эта переменная состоит только из альфа-части и может получить свое значение из альфа-части любого регистра за исключением двух псевдорегистров. Значение `G` также может быть инвертировано. При этом величина

spare0+secondary color может либо отсекаться по отрезку [0, 1], либо принимать значения из [0, 2].

Фактически путем задания цепочки преобразований в блоках general combiner и в блоке final combiner мы строим свой конвейер, определяющий обработку данных. Тогда при выводе примитивов будет действовать именно этот алгоритм обработки, построенный на основе механизма register combiner (если включено его использование).

Рассмотрим теперь, как осуществляется программирование (настройка) блоков register combiner.

1. Первым шагом является их включение командой

```
glEnable ( GL_REGISTER_COMBINERS_NV );
```

2. Далее следует указать число блоков general comber:

```
glCombinerParamteriNV ( GL_NUM_GENERAL_COMBINERS_NV, num );
```

3. Узнать максимальное количество блоков general combiner, поддерживаемое графической картой и драйвером, можно при помощи следующего фрагмента кода:

```
int maxCombiners;

glGetIntegerv ( GL_MAX_GENERAL_COMBINERS_NV, &maxCombiners    );
```

4. Далее для каждого из используемых блоков general combiner следует задать входные и выходные величины.

   Задание входных величин осуществляется при помощи функции

```
glCombinerInputNV ( GLenum stage, GLenum portion, GLenum var,
                    GLenum input, GLenum mapping, GLenum usage );
```

   Назначение ее параметров следующее:

   - stage задает номер используемого блока и принимает одно из значений от GL_COMBINER0_NV до GL_COMBINER7_NV;
   - portion определяет, какая часть переменной (RGB или альфа) будет задаваться, и принимает одно из двух значений: GL_RGB или GL_ALPHA;
   - var определяет используемую внутреннюю переменную и принимает одно из значений: GL_VARIABLE_A_NV, GL_VARIABLE_B_NV, GL_VARIABLE_C_NV и GL_VARIABLE_D_NV;
   - input определяет, из какого входного регистра следует взять значение для соответствующей переменной. Допустимые значения для этого параметра были приведены в табл. 9.1;
   - mapping задает отображение, применяемое к значению регистра перед записью в переменную (см. табл. 9.2);

- `usage` определяет, какая часть (RGB или альфа) регистра будет записана (после преобразования) в соответствующую переменную.

Значение для RGB-части переменной может быть получено из RGB- или альфа-части любого регистра. Значение для альфа-части переменной может быть получено либо из альфа-части, либо из синей (*blue*) цветовой компоненты регистра.

Задание выходных величин осуществляется при помощи функции

```
glCombinerOutputNV ( GLenum stage,    GLenum portion,
                     GLenum abOutput, GLenum cdOutput, GLenum sumOutput,
                     GLenum scale,    GLenum bias,
                     GLboolean abDotProduct, GLboolean cdDotProduct,
                     GLboolean muxSum );
```

Назначение ее параметров следующее:

- `abOutput`, `cdOutput` и `sumOutput` определяют, в какой регистр пойдет каждое из трех выходных значений. Допустимыми значениями являются `GL_DISCARD_NV` (отбросить соответствующую величину), `GL_PRIMARY_COLOR_NV`, `GL_SECONDARY_COLOR_NV`, `GL_SPARE0_NV`, `GL_SPARE1_NV`, `GL_TEXTURE0_ARB` и `GL_TEXTURE1_ARB`;
- `scale` задает масштабирующий множитель, на который будет умножено значение переменной, допустимые значения были указаны в табл. 9.3;
- `bias` позволяет задать значение, которое будет вычтено из переменной после операции масштабирования (см. табл. 9.4);
- `abDotProduct` и `cdDotProduct` определяют, каким образом происходит вычисление двух выходных значений (`AB` и `CD`). Если параметр принимает значение `GL_TRUE`, то на соответствующем выходе будет величина, все компоненты которой равны соответствующему скалярному произведению ((A,B) или (C,D)). В противном случае на выходе будет покомпонентное произведение входных величин. Если хотя бы один из параметров `abDotProduct` или `cdDotProduct` не равен нулю, то параметр `sumOutput` должен принимать значение `GL_DISCARD_NV`;
- если параметр `muxSum` равен `GL_FALSE`, то на выходе `ABCD` мы получим `AB + CD`. В противном случае — `AB mux CD`.

Параметры `stage` и `portion` имеют тот же смысл, что и для функции `glCombinerInputNV`.

5. Заключительным шагом является настройка блока final combiner, командой

```
glFinalCombinerInputNV ( GLenum var,    GLenum input,
                         GLenum mapping, GLenum usage );
```

Назначение ее параметров следующее:

- `var` указывает имя переменной, которая будет задаваться, и принимает одно из значений от `GL_VARIABLE_A_NV` до `GL_VARIABLE_G_NV`;
- `input` может принимать значения как из табл. 9.1, так и дополнительные, соответствующие псевдорегистрам (`GL_E_TIMES_F_NV` и `GL_SPARE0_PLUS_SECONDARY_COLOR_NV`);
- `mapping` задает преобразование входных значений и может принимать только одно из двух значений `GL_UNSIGNED_IDENTITY_NV` и `GL_UNSIGNED_INVERT_NV` (см. табл. 9.2);
- `usage` определяет, какая из частей (RGB или альфа) входного регистра будет использоваться для задания соответствующей переменной и принимает одно из значений `GL_RGB` или `GL_ALPHA`. Однако если задается переменная `G`, то значение `usage` должно равняться `GL_ALPHA`. Если переменная принимает входное значение либо из псевдорегистра `EF`, либо из псевдорегистра `spare0+secondary color`, то значением `usage` должно быть `GL_RGB`.

Как было замечено ранее, значения псевдорегистра `spare0+secondary color` принадлежат отрезку [0, 2], либо обрезаются по отрезку [0, 1]. Для установки отсечения по отрезку [0, 1] используется команда:

```
glCombinerParameteriNV ( GL_COLOR_SUM_CLAMP_NV, GL_TRUE );
```

Установка отсечения по отрезку [0,2] обеспечивается командой:

```
glCombinerParameteriNV ( GL_COLOR_SUM_CLAMP_NV, GL_FALSE );
```

Для установки значений регистров `constant color`, можно использовать следующий фрагмент кода.

```
float c1 [4] = { 1, 0.2, 0.5, 1 };
float c2 [4] = { 1, 1, 1, 0.5 };
glCombinerParamterfvNV ( GL_CONSTANT_COLOR0_NV, c0 );
glCombinerParamterfvNV ( GL_CONSTANT_COLOR1_NV, c1 );
```

По умолчанию оба регистра `constant color` являются нулевыми, и включен режим отсечения по отрезку [0, 1].

Листинг 9.1 содержит пример выполнения проверки поддержки режима register combiners. Полный исходный код для примера, проверяющего поддержку режима register combiners и печатающего максимально доступное количество блоков general combiner, находится на прилагаемом к книге компакт-диске (файл rc-info.cpp).

Листинг 9.1. Пример, выдающий информацию о поддержке register combiner

```
//
// Sample to to show Register Combiners 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
    int    win = glutCreateWindow ( "OpenGL example 1" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    init ();
    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_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_NV_register_combiners" ) )
    {
        printf ( "NV_register_combiners NOT supported" );
        return 2;
    }
    int    maxTextureUnits;
    int    maxCombiners;
    glGetIntegerv ( GL_MAX_TEXTURE_UNITS_ARB,    &maxTextureUnits );
    glGetIntegerv ( GL_MAX_GENERAL_COMBINERS_NV, &maxCombiners    );
    printf ( "ARB_multitexture supported.\nMax texture units %d.\n",
                maxTextureUnits );
    printf ( "NV_register_combiners supported.\nMax general combiners
%d.\n", maxCombiners );
    return 0;
}
```

Реализация диффузного освещения через register combiner

Использование механизма register combiner позволяет достаточно легко реализовывать попиксельное диффузное освещение, не прибегая при этом к расширению texture_env_dot3. Этот механизм является гораздо более гибким, чем расширение texture_env_dot3, как мы убедимся далее, с его помощью легко можно реализовать также бликовое (*specular*) освещение и ряд других эффектов.

Выберем более общую модель диффузной освещенности:

$$I = I_{amb} + \max(0, (l, n)) \cdot I_l,$$

где коэффициент I_{amb} задает фоновую освещенность, а I_l — цвет источника света.

Как и ранее, вектор направления на источник света l мы будем задавать в вершинах грани в виде текстурных координат, а нормализацию значений этого вектора, получаемых в процессе линейной интерполяции вдоль грани, будем выполнять с помощью нормирующей кубической карты.

В простейшем случае нам понадобится всего один блок general combiner.

В нулевом текстурном блоке (регистр `texture0`) мы разместим карту нормалей, а в первом (регистр `texture1`) — нормирующую кубическую карту.

Фоновый цвет I_{amb} мы разместим в регистре `constant-color1`, а цвет источника света I_l — в `constant-color1`.

Теперь нам надо включить использование режима register combiner и задать число блоков general combiner и постоянные цвета. Это делается с помощью следующего фрагмента кода:

```
                                    // setup register combiners
glEnable                ( GL_REGISTER_COMBINERS_NV );
glCombinerParameteriNV  ( GL_NUM_GENERAL_COMBINERS_NV, 1 );
glCombinerParameterfvNV ( GL_CONSTANT_COLOR0_NV, lightColor );
glCombinerParameterfvNV ( GL_CONSTANT_COLOR1_NV, ambientColor );
```

Рассмотрим теперь конфигурацию нулевого блока general combiner (рис. 9.8).

В качестве RGB-части переменной `A` мы возьмем значение из карты нормалей, т. е. `texture0`. При этом преобразованием (*mapping*) будет `GL_EXPAND_NORMAL_NV`, чтобы перевести представление вектора в виде цвета в нормальное знаковое представление (x, y, z).

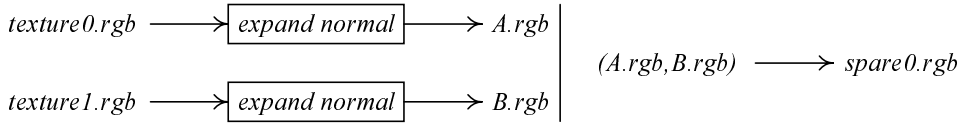


Рис. 9.8. Настройки блока general combiner 0

Соответствующий фрагмент кода представлен далее:

```
                        // configure A = expand (tex0) (bumpmap)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
                    GL_TEXTURE0_ARB, GL_EXPAND_NORMAL_NV, GL_RGB );
```

Аналогичным образом в качестве RGB-части переменной `B` возьмем значение с первого текстурного блока (регистра `texture1`), которое будет представлять собой нормированное значение вектора *l* в виде цвета. Здесь опять следует применить преобразование `GL_EXPAND_NORMAL_NV` для перевода вектора в координатное представление:

```
                        // configure B = expand (tex1) (norm. map)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV, GL_RGB );
```

Рассмотрим теперь, каким образом следует настроить выход нулевого блока register combiner для получения на выходе скалярного произведения (*l*, *n*).

Для этого RGB-выход `AB` направим в регистр `spare0`, остальные два выходных значения отбросим. Ни масштабирования, ни сдвига полученного значения нам не потребуется, но необходимо задать вычисление скалярного (а не покомпонентного) произведения для выхода `AB`. Этому соответствует фрагмент кода:

```
                        // setup output of (l,n)
glCombinerOutputNV ( GL_COMBINER0_NV, GL_RGB,
                     GL_SPARE0_NV,          // AB output
                     GL_DISCARD_NV,         // CD output
                     GL_DISCARD_NV,         // sum output
                     GL_NONE,               // no scale
                     GL_NONE,               // no bias
                     GL_TRUE,               // AB = A dot B
                     GL_FALSE, GL_FALSE );
```

В результате в регистре `spare0.rgb` у нас окажется нужное нам скалярное произведение во всех его цветовых компонентах.

Рассмотрим теперь конфигурацию блока final combiner.

В качестве значения переменной `A` следует взять `spare0.rgb`, подвергнув его преобразованию `GL_UNSIGNED_IDENTITY_NV`, в результате чего в `A` мы получим max(0, (*l*, *n*)). Иллюстрацией будет фрагмент кода:

```
                          // A.rgb = max ( spare0.rgb, 0 )
glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_SPARE0_NV,
                          GL_UNSIGNED_IDENTITY_NV, GL_RGB );
```

В качестве переменной `B` мы возьмем RGB-значение из регистра `constant-color0`, содержащего значение цвета источника света. Преобразованием будет `GL_UNSIGNED_IDENTITY_NV`:

```
                          // B.rgb = constant_color0.rgb
glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_CONSTANT_COLOR0_NV,
                          GL_UNSIGNED_IDENTITY_NV, GL_RGB );
```

В переменную `C` мы запишем нулевой вектор:

```
                          // C = 0
glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                          GL_UNSIGNED_IDENTITY_NV, GL_RGB );
```

В переменную `D` запишем значение фоновой освещенности, т. е. значение цвета из регистра `constant-color1`:

```
                          // D = constant_color1.rgb
glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_CONSTANT_COLOR1_NV,
                          GL_UNSIGNED_IDENTITY_NV, GL_RGB );
```

В RGB-части переменных `E` и `F` также запишем `0`, а в альфа-часть переменной `G` — единицу:

```
                          // E = 0
glFinalCombinerInputNV ( GL_VARIABLE_E_NV, GL_ZERO,
                          GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                          // F = 0
glFinalCombinerInputNV ( GL_VARIABLE_F_NV, GL_ZERO,
                          GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                          // G.alpha = 1
glFinalCombinerInputNV ( GL_VARIABLE_G_NV, GL_ZERO,
                          GL_UNSIGNED_INVERT_NV, GL_ALPHA );
```

При такой конфигурации мы на выходе получим следующие результаты:

$$RGB = I_{amb} + \max(0, (l, n)) \cdot I_l,$$

$$Alpha = 1$$

На рис. 9.9 приводится диаграмма настройки блока final combiner, на рис. 9.10 — скриншот результата работы программы.

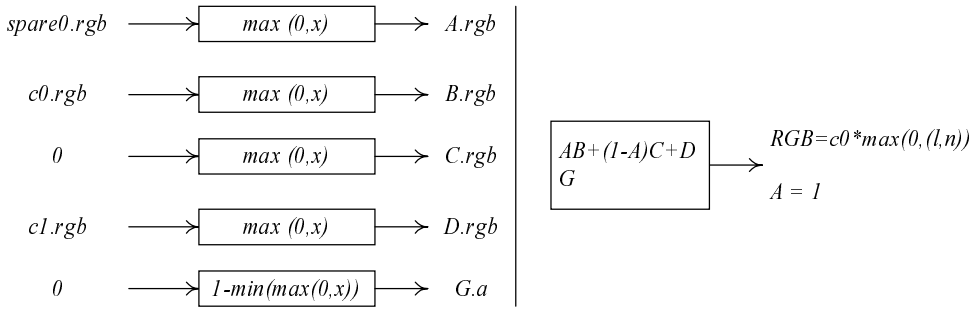


Рис. 9.9. Настройка блока final combiner

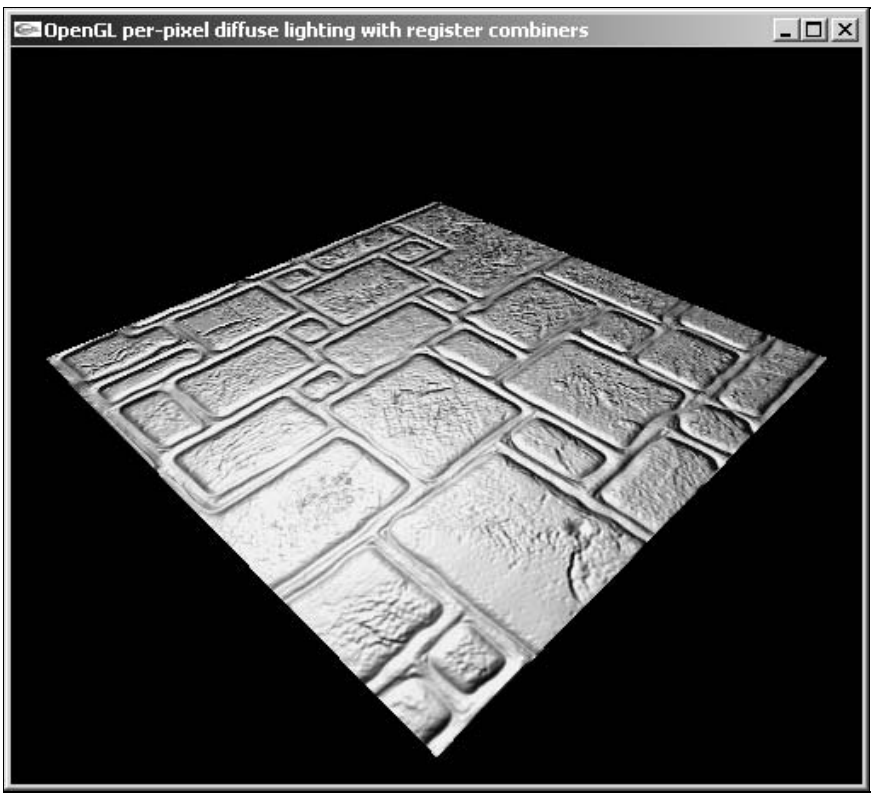


Рис. 9.10. Результат работы программы попиксельного диффузного освещения

Листинг 9.2 содержит полный код программы, реализующей попиксельное диффузное освещение.

Листинг 9.2. Простейшая реализация попиксельного диффузного освещения с использованием register combiner

```
//
// Sample to to show diffuse lighting via Register Combiners support
// in OpenGL
//
#include    "libExt.h"
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
Vector3D    eye   ( 2, 2, 2 );        // camera position
Vector3D    light ( 0.5, 0, 1 );      // light position
unsigned    normCubeMap;              // normalization cubemap id
unsigned    bumpMap;                  // normal map
unsigned    decalMap;                 // decal (diffuse) texture
float       angle = 0;
Vector3D    v  [4];
Vector2D    t0 [4];
Vector3D    t1 [4];
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 ()
{
    float    lightColor   [4] = { 1, 1, 0, 1 };       // yellow light
    float    ambientColor [4] = { 0, 0, 0.5f, 1 };    // dark blue
```

```
glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
                    // setup texture units
                    // bind bump (normal) map
                    // to texture unit 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glEnable           ( GL_TEXTURE_2D );
glBindTexture      ( GL_TEXTURE_2D, bumpMap );
                    // bind normalization cube map
                    // to texture unit 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glEnable           ( GL_TEXTURE_CUBE_MAP_ARB );
glBindTexture      ( GL_TEXTURE_CUBE_MAP_ARB, normCubeMap );
                    // setup register combiners
glEnable                ( GL_REGISTER_COMBINERS_NV );
glCombinerParameteriNV  ( GL_NUM_GENERAL_COMBINERS_NV, 1 );
glCombinerParameterfvNV ( GL_CONSTANT_COLOR0_NV, lightColor );
glCombinerParameterfvNV ( GL_CONSTANT_COLOR1_NV, ambientColor );
                    // configure A = expand (tex0)
                    // (bump map)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
                    GL_TEXTURE0_ARB, GL_EXPAND_NORMAL_NV,
                    GL_RGB );
                    // configure B = expand (tex1)
                    // (norm. map)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                    GL_RGB );
                    // setup output of (l,n)
glCombinerOutputNV ( GL_COMBINER0_NV, GL_RGB,
                     GL_SPARE0_NV,         // AB output
                     GL_DISCARD_NV,        // CD output
                     GL_DISCARD_NV,        // sum output
                     GL_NONE,              // no scale
                     GL_NONE,              // no bias
                     GL_TRUE,              // AB = A dot B
                     GL_FALSE, GL_FALSE );
                    // now spare0.rgb
                    // contains (l,n)
```

```
                    // configure final combiner
                    // A.rgb = max (spare0.rgb, 0)
glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // B.rgb = constant_color0.rgb
glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_CONSTANT_COLOR0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // C = 0
glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // D = constant_color1.rgb
glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_CONSTANT_COLOR1_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // E = 0
glFinalCombinerInputNV ( GL_VARIABLE_E_NV, GL_ZERO,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // F = 0
glFinalCombinerInputNV ( GL_VARIABLE_F_NV, GL_ZERO,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // G.alpha = 1
glFinalCombinerInputNV ( GL_VARIABLE_G_NV, GL_ZERO,
                         GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                    // so at output we'll get
                    // RGB   = c1+c0*max((l, n),0)
                    // alpha = 1
int    i;
                    // compute texture coordinates
                    // for normalization cibe map
for (  i = 0; i < 4; i++ )
    t1 [i] = light - v [i];
                    // now draw quad
glBegin ( GL_QUADS );
for ( i = 0; i < 4; i++ )
{
    glMultiTexCoord2fv ( GL_TEXTURE0_ARB, t0 [i] );
    glMultiTexCoord3fv ( GL_TEXTURE1_ARB, t1 [i] );
    glVertex3fv        ( v [i]                    );
}
```

```
    glEnd   ();
    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 key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )
            exit ( 0 );
}
void    animate ()
{
    angle  = 0.004f * glutGet ( GLUT_ELAPSED_TIME );
    light.x = cos ( angle );
    light.y = sin ( angle );
    light.z = 1 + 0.3 * sin ( angle / 3 );
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                        // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                        // create window
    int    win = glutCreateWindow("OpenGL per-pixel diffuse "
                                  "lighting with register combiners");
                        // register handlers
```

```
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init       ();
    printfInfo ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_NV_register_combiners" ) )
    {
        printf ( "NV_register_combiners NOT supported" );
        return 2;
    }
    initExtensions ();
                        // setup vertices
    v [0] = Vector3D ( -1.3, -1.3, 0 );
    v [1] = Vector3D (  1.3, -1.3, 0 );
    v [2] = Vector3D (  1.3,  1.3, 0 );
    v [3] = Vector3D ( -1.3,  1.3, 0 );
                        // setup texture coordinates
                        // for bump map
    t0 [0] = Vector2D ( 0, 0 );
    t0 [1] = Vector2D ( 1, 0 );
    t0 [2] = Vector2D ( 1, 1 );
    t0 [3] = Vector2D ( 0, 1 );
    bumpMap     = createTexture2D           ( false,
                            "../Textures/Bumpmaps/FieldStone-DOT3.tga" );
    decalMap    = createTexture2D            ( true,
                            "../Textures/FieldStone.tga" );
    normCubeMap = createNormalizationCubemap ( 32 );
    glutMainLoop ();
return 0;
}
```

Самозатенение поверхностей

Рассмотрим теперь еще одно явление, связанное с микрорельефом, — самозатенение поверхности. Обратите внимание на ситуации, представленные на рис. 9.11.

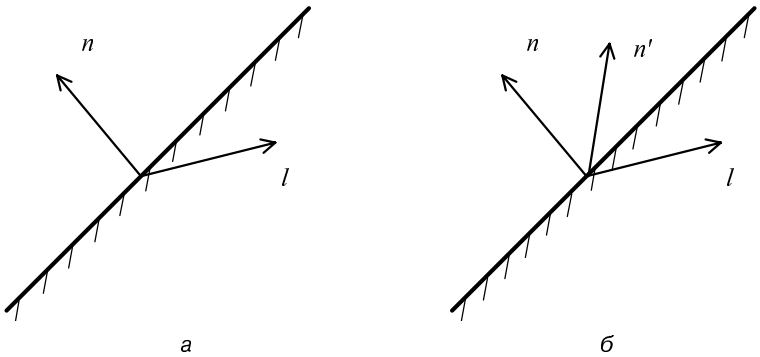


Рис. 9.11. Случаи самозатенения

В случае, изображенном на рис. 9.11, *а*, исходная (неизмененная картой нормали) нормаль к поверхности смотрит от источника света и не может освещать эту поверхность.

Второй случай (9.11, *б*) гораздо интереснее. Хотя исходный вектор нормали n и направлен от источника света, но измененная нормаль n' теперь уже направлена на источник света, т. е. при описанном ранее алгоритме эта точка будет освещена, чего не должно быть.

Данное явление и есть самозатенение, для его компенсации в уравнение освещенности вводится специальный коэффициент S_{self}.

$$I = I_{amb} + \max(0, (l, n)) \cdot I_l \cdot S_{self}.$$

В качестве коэффициента самозатенения S_{self} в простейшем случае можно взять единицу, если $(l, n) > 0$ и ноль в противном случае. Однако в подобном случае, из-за того, что такой коэффициент самозатенения является разрывной функцией, могут возникнуть нежелательные артефакты. Поэтому на практике обычно коэффициент самозатенения вычисляется так:

$$S_{self} = \begin{cases} 1, & \text{если } (l, n) > c; \\ S_{self}(l, n)/c, & \text{если } 0 > (l, n) \geq c; \\ 0, & \text{если } (l, n) > c. \end{cases}$$

Коэффициент c обычно равен 1/8.

Рассмотрим теперь, как можно вычислить коэффициент самозатенения по данной формуле.

Поскольку все векторы считаются заданными в касательном пространстве, то вектор n равен (0, 0, 1) и $(l, n) = l_z$. Если теперь эту величину умножить на 8 (для этого можно ее сначала сложить с собой, а затем умножить на 4), то после отсечения по отрезку [0, 1] мы и получим требуемый коэффициент самозатенения.

Вычисление коэффициента самозатенения удобно проводить в альфа-частях переменных и регистров. Рисунок 9.12 и листинг 9.3 иллюстрируют, как это реализуется на практике.

Листинг 9.3. Вычисление коэффициента самозатенения

```
                        // configure A.alpha = 1
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_A_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                        // configure B.alpha = lz
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_B_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV, GL_BLUE );
                        // configure C.alpha = 1
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_C_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                        // configure D.alpha = lz
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_D_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV, GL_BLUE );
glCombinerOutputNV ( GL_COMBINER0_NV, GL_ALPHA,
                     GL_DISCARD_NV,        // AB
                     GL_DISCARD_NV,        // CD
                     GL_SPARE0_NV,         // AB+CD
                     GL_SCALE_BY_FOUR_NV,  // scale by 4
                     GL_NONE,              // bias
                     GL_FALSE,
                     GL_FALSE, GL_FALSE );
```

После выполнения этого фрагмента в `spare0.alpha` у нас окажется величина, равная $8l_z$.

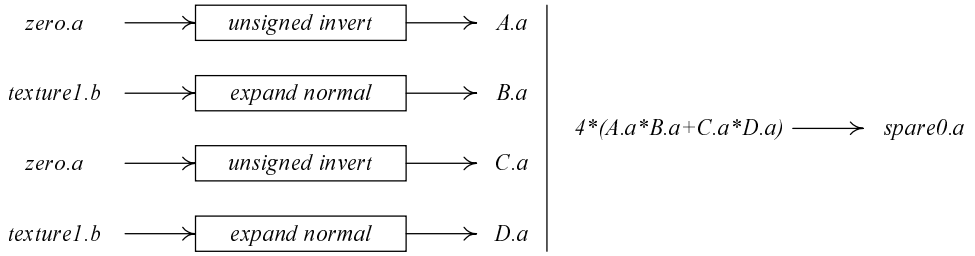


Рис. 9.12. Настройка блока general combiner 0

Рассмотрим теперь, каким образом следует сконфигурировать блок final combiner для учета коэффициента самозатенения (рис. 9.13). Предлагается следующее задание переменных:

```
A.rgb   = max ( 0, spare0.rgb )    // unsigned_identity
B.rgb   = EF
C.rgb   = 0
D.rgb   = constant-color1.rgb
E.rgb   = constant-color0.rgb
F.rgb   = max ( 0,spare0.alpha )   // unsigned_identity
G.alpha = 1
```

Как несложно убедиться, полученное на выходе значение и будет диффузной освещенностью с учетом самозатенения. Код для данной конфигурации final combiner приводится в листинге 9.4.

Листинг 9.4. Пример конфигурации блока final combiner

```
                          // A.rgb = max ( spare0.rgb, 0 )
glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                          // B.rgb = EF
glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_E_TIMES_F_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                          // C = 0
glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                          // D = constant_color1.rgb
glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_CONSTANT_COLOR1_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                          // E = constant_color_0
```

```
glFinalCombinerInputNV ( GL_VARIABLE_E_NV, GL_CONSTANT_COLOR0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                           // F = selfShadow = 8*lz
glFinalCombinerInputNV ( GL_VARIABLE_F_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_ALPHA );
                           // G.alpha = 1
glFinalCombinerInputNV ( GL_VARIABLE_G_NV, GL_ZERO,
                         GL_UNSIGNED_INVERT_NV, GL_ALPHA );
```

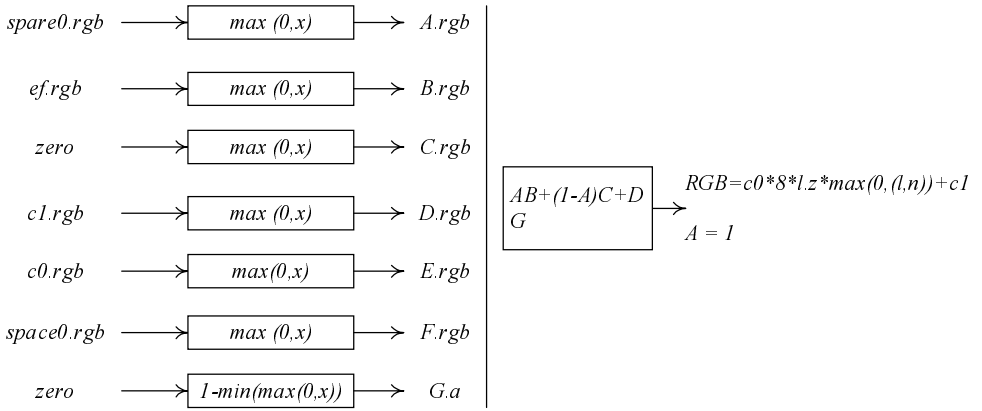


Рис. 9.13. Настройка блока final combiner

Листинг 9.5 содержит полный исходный код программы, реализующей попиксельное освещение с самозатенением.

Листинг 9.5. Реализация попиксельного диффузного освещения с самозатенением

```
//
// Sample to to show Register Combiners support in OpenGL card and driver
//
#include    "libExt.h"
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
```

```
Vector3D    eye   ( 2, 2, 2 );        // camera position
Vector3D    light ( 0.5, 0, 1 );      // light position
unsigned    normCubeMap;              // normalization cubemap id
unsigned    bumpMap;                  // normal map
unsigned    decalMap;                 // decal (diffuse) texture
float       angle = 0;
Vector3D    v  [4];
Vector2D    t0 [4];
Vector3D    t1 [4];
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 ()
{
    float    lightColor   [4] = { 1, 1, 0, 1 };       // yellow light
    float    ambientColor [4] = { 0, 0, 0.5f, 1 };    // dark blue

    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
                        // setup texture units
                        // bind bump (normal) map to
                        // texture unit 0
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, bumpMap );
                        // bind normalization cube map
                        // to texture unit 1
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnable           ( GL_TEXTURE_CUBE_MAP_ARB );
    glBindTexture      ( GL_TEXTURE_CUBE_MAP_ARB, normCubeMap );
                        // setup register combiners
    glEnable               ( GL_REGISTER_COMBINERS_NV );
    glCombinerParameteriNV  ( GL_NUM_GENERAL_COMBINERS_NV, 1 );
    glCombinerParameterfvNV ( GL_CONSTANT_COLOR0_NV, lightColor );
```

```
glCombinerParameterfvNV ( GL_CONSTANT_COLOR1_NV, ambientColor );
                    // A.rgb = expand (tex0)
                    // (bumpmap)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
                    GL_TEXTURE0_ARB, GL_EXPAND_NORMAL_NV,
                    GL_RGB );
                    // B.rgb = expand (tex1)
                    // (norm. map)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_TEXTURE1_ARB,  GL_EXPAND_NORMAL_NV,
                    GL_RGB );
                    // configure C.rgb = 0
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_C_NV,
                    GL_ZERO, GL_UNSIGNED_IDENTITY_NV,
                    GL_RGB );
                    // configure D.rgb = 0
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_D_NV,
                    GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // setup RGB output of (l,n)
glCombinerOutputNV ( GL_COMBINER0_NV, GL_RGB,
                     GL_SPARE0_NV,       // AB output
                     GL_DISCARD_NV,      // CD output
                     GL_DISCARD_NV,      // sum output
                     GL_NONE,            // no scale
                     GL_NONE,            // no bias
                     GL_TRUE,            // AB = A dot B
                     GL_FALSE, GL_FALSE );
                    // configure A.alpha = 1
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_A_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                    // configure B.alpha = lz
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_B_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                    GL_BLUE );
                    // configure C.alpha = 1
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_C_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                    // configure D.alpha = lz
```

```
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_D_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                    GL_BLUE );
glCombinerOutputNV ( GL_COMBINER0_NV, GL_ALPHA,
                     GL_DISCARD_NV,        // AB
                     GL_DISCARD_NV,        // CD
                     GL_SPARE0_NV,         // AB+CD
                     GL_SCALE_BY_FOUR_NV,
                     GL_NONE,              // bias
                     GL_FALSE, GL_FALSE, GL_FALSE );
                    // now spare0.rgb = (l,n)
                    // spare0.alpha contains 8*lz
                    // configure final combiner
                    // A.rgb = max (spare0.rgb, 0)
glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // B.rgb = EF
glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_E_TIMES_F_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // C = 0
glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // D = constant_color1.rgb
glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_CONSTANT_COLOR1_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // E = constant_color_0
glFinalCombinerInputNV ( GL_VARIABLE_E_NV, GL_CONSTANT_COLOR0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // F = selfShadow = 8*lz
glFinalCombinerInputNV ( GL_VARIABLE_F_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_ALPHA );
                    // G.alpa = 1
glFinalCombinerInputNV ( GL_VARIABLE_G_NV, GL_ZERO,
                         GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                    // so at output we'll get
                    // RGB   = c1+c0*max((l, n),0)
                    // alpha = 1
                    // compute texture coordinates
                    // for normalization cube map
```

```
    int    i;
    for ( i = 0; i < 4; i++ )
        t1 [i] = light - v [i];
                        // now draw quad
    glBegin ( GL_QUADS );
    for ( i = 0; i < 4; i++ )
    {
        glMultiTexCoord2fv ( GL_TEXTURE0_ARB, t0 [i] );
        glMultiTexCoord3fv ( GL_TEXTURE1_ARB, t1 [i] );
        glVertex3fv        ( v [i]                    );
    }
    glEnd   ();
    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 key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
            exit ( 0 );
    if ( key == 'p' || key == 'P' )
        saveScreenShot ( "rc-diffuse-2.tga" );
}
void    animate ()
{
    angle  = 0.004f * glutGet ( GLUT_ELAPSED_TIME );
    light.x = cos ( angle );
    light.y = sin ( angle );
```

```
    light.z = 1 + 0.3 * sin ( angle / 3 );
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                        // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                        // create window
    int    win = glutCreateWindow("OpenGL per-pixel diffuse lighting"
                                  "with register combiners (atten)" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init       ();
    printfInfo ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_NV_register_combiners" ) )
    {
        printf ( "NV_register_combiners NOT supported" );
        return 2;
    }
    initExtensions ();
                        // setup vertices
    v [0] = Vector3D ( -1.3, -1.3, 0 );
    v [1] = Vector3D (  1.3, -1.3, 0 );
    v [2] = Vector3D (  1.3,  1.3, 0 );
    v [3] = Vector3D ( -1.3,  1.3, 0 );
                       // setup texture coordinates
                        // for bump map
    t0 [0] = Vector2D ( 0, 0 );
    t0 [1] = Vector2D ( 1, 0 );
```

```
    t0 [2] = Vector2D ( 1, 1 );
    t0 [3] = Vector2D ( 0, 1 );
    bumpMap     = createTexture2D ( false,
                          "../Textures/Bumpmaps/FieldStone-DOT3.tga" );
    decalMap    = createTexture2D ( true,
                          "../Textures/FieldStone.tga" );
    normCubeMap = createNormalizationCubemap ( 32 );
    glutMainLoop ();
    return 0;
}
```

Рассмотрим теперь, как можно построить изображение тора с наложенной на него картой нормалей.

Тор будем представлять в виде совокупности треугольных граней, построенных по набору вершин. При этом каждая грань тора задается тремя индексами в массив вершин.

Для каждой вершины тора, помимо положения в пространстве, текстурных координат и единичного вектора внешней нормали n, следует также задать касательный вектор t и бинормаль b. Это позволяет в каждой вершине построить свое касательное пространство с собственной системой координат. Это иллюстрирует листинг 9.6.

Листинг 9.6. Структуры, используемые для задания тора

```
struct    Vertex
{
    Vector3D  pos;              // position of vertex
    Vector2D  tex;              // texture coordinates
    Vector3D  n;                // unit normal
    Vector3D  t, b;             // tangent and binormal
    Vector3D  l;                // light vector in the tangent space
                                // map vector to tangent (TBN) space
    Vector3D  mapToTangentSpace ( const Vector3D& v ) const
    {
         return Vector3D ( v & t, v & b, v & n );
    }
};
struct  Face                    // triangular face
```

```
{
    int index [3];              // indices to Vertex array
};
```

В описании вершины также содержится вектор *l* направления на источник света. Это удобно тем, что можно сначала вычислить его значения для всех вершин, а затем при построении тора использовать вершинные массивы, повысив быстродействие программы.

Здесь также добавлен метод `mapToTangentSpace`, служащий для перевода произвольного вектора в систему координат касательного пространства.

Следующий фрагмент кода дает описание класса `Torus`.

```
class    Torus
{
    int      numRings;
    int      numSides;
    int      numVertices;
    int      numFaces;
    Vertex * vertices;
    Face   * faces;
public:
    Torus  ( float r1, float r2, int rings, int sides );
    ~Torus ()
    {
        delete [] vertices;
        delete [] faces;
    }
    void  calcLightVectors ( const Vector3D& light );
    void  draw             ();
};
```

В конструкторе класса осуществляется заполнение массивов вершин и граней, используя при этом параметрическое уравнение тора.

Метод `calcLightVectors` по заданному положению источника света осуществляет вычисление вектора *l* для каждой вершины и перевод его в систему координат касательного пространства соответствующей вершины.

```
void    Torus :: calcLightVectors ( const Vector3D& light )
{
                                // compute texture coordinates for
                                // normalization cibe map
```

```
    for ( int i = 0; i < numVertices; i++ )
        vertices [i].l = vertices [i].mapToTangentSpace ( light -
                                                    vertices [i].pos );
}
```

Метод `draw` служит для отрисовки тора.

Листинг 9.7 содержит полный код программы, реализующей диффузное освещение поверхности тора с самозатенением, а на рис. 9.14 представлен результат ее работы.

Рис. 9.14. Тор с диффузным освещением

Листинг 9.7. Программа построения изображения тора с попиксельным диффузным освещением с самозатенением

```
//
// Sample to to show diffuse lighting via Register Combiners in OpenGL
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Torus.h"
Vector3D    eye   ( 7, 5, 7 );      // camera position
Vector3D    light ( 5, 0, 4 );      // light position
unsigned    normCubeMap;            // normalization cubemap id
unsigned    bumpMap;                // normal map
unsigned    decalMap;               // decal (diffuse) texture
float       angle = 0;
Torus       torus ( 1, 3, 30, 30 );
void display ()
{
    float    lightColor   [4] = { 1, 1, 0, 1 };     // yellow light
    float    ambientColor [4] = { 0, 0, 0.5f, 1 };  // dark blue
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
                                                    // draw light
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();
    glDisable          ( GL_REGISTER_COMBINERS_NV );
    glTranslatef       ( light.x, light.y, light.z );
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glDisable          ( GL_TEXTURE_CUBE_MAP_ARB );
    glActiveTextureARB ( GL_TEXTURE2_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glColor4f          ( 1, 1, 1, 1 );
```

```
glutSolidSphere    ( 0.07f, 15, 15 );
glPopMatrix        ();
                    // setup texture units
                    // bind bump (normal) map to
                    // texture unit 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glEnable           ( GL_TEXTURE_2D );
glBindTexture      ( GL_TEXTURE_2D, bumpMap );
glMatrixMode       ( GL_TEXTURE );
glLoadIdentity     ();
glScalef           ( 1, 2, 1 );
glMatrixMode       ( GL_MODELVIEW );
                    // bind normalization cube map
                    // to texture unit 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glEnable           ( GL_TEXTURE_CUBE_MAP_ARB );
glBindTexture      ( GL_TEXTURE_CUBE_MAP_ARB, normCubeMap );
                    // bind decal texture to
                    // texture unit 2
glActiveTextureARB ( GL_TEXTURE2_ARB );
glEnable           ( GL_TEXTURE_2D );
glBindTexture      ( GL_TEXTURE_2D, decalMap );
glMatrixMode       ( GL_TEXTURE );
glLoadIdentity     ();
glScalef           ( 1, 2, 1 );
glMatrixMode       ( GL_MODELVIEW );
                    // setup register combiners
glEnable                ( GL_REGISTER_COMBINERS_NV );
glCombinerParameteriNV  ( GL_NUM_GENERAL_COMBINERS_NV, 1 );
glCombinerParameterfvNV ( GL_CONSTANT_COLOR0_NV, lightColor );
glCombinerParameterfvNV ( GL_CONSTANT_COLOR1_NV, ambientColor );
                    // A = expand (tex0) (bumpmap)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
                    GL_TEXTURE0_ARB, GL_EXPAND_NORMAL_NV,
                    GL_RGB );
                    // B = expand (tex1)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                    GL_RGB );
```

```
                    // C = 0
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_C_NV,
                    GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // D = 0
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_D_NV,
                    GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // setup output of (l,n)
glCombinerOutputNV ( GL_COMBINER0_NV, GL_RGB,
                     GL_SPARE0_NV,        // AB output
                     GL_DISCARD_NV,       // CD output
                     GL_DISCARD_NV,       // sum output
                     GL_NONE,             // no scale
                     GL_NONE,             // no bias
                     GL_TRUE,             // AB = A dot B
                     GL_FALSE, GL_FALSE );
                    // configure A.alpha = 1
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_A_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                    // configure B.alpha = lz
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_B_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                    GL_BLUE );
                    // configure C.alpha = 1
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_C_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                    // configure D.alpha = lz
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_D_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                    GL_BLUE );
glCombinerOutputNV ( GL_COMBINER0_NV, GL_ALPHA,
                     GL_DISCARD_NV,       // AB
                     GL_DISCARD_NV,       // CD
                     GL_SPARE0_NV,        // AB+CD
                     GL_SCALE_BY_FOUR_NV,
                     GL_NONE,             // bias
                     GL_FALSE, GL_FALSE, GL_FALSE );
                    // now spare0.rgb contains l,n)
```

```
                        // spare0.alpha contains 8*lz
                        // configure final combiner
                        // A.rgb = max (spare0.rgb, 0)
    glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_SPARE0_NV,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                        // B.rgb = EF
    glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_E_TIMES_F_NV,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                        // C = 0
    glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                        // D = 0
    glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                        // E = texture2
    glFinalCombinerInputNV ( GL_VARIABLE_E_NV, GL_TEXTURE2_ARB,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                        // F = selfShadow = 8*lz
    glFinalCombinerInputNV ( GL_VARIABLE_F_NV, GL_SPARE0_NV,
                             GL_UNSIGNED_IDENTITY_NV, GL_ALPHA );
                        // G.alpa = 1
    glFinalCombinerInputNV ( GL_VARIABLE_G_NV, GL_ZERO,
                             GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                        // so at output we'll get
                        // RGB   = c1 + c0*max((l,n),0)
                        // alpha = 1
                        // now draw torus
    torus.calcLightVectors ( light );
    torus.draw             ();
    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, 1 );              // up
}
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 );
    glutPostRedisplay ();
}
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 );
}
int main ( int argc, char * argv [] )
{
                            // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                            // create window
    int    win = glutCreateWindow ( "Torus with  per-pixel diffuse
    lighting via register combiners" );
                            // register handlers
    glutDisplayFunc  ( display );
```

```
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init ();
    printfInfo ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_NV_register_combiners" ) )
    {
        printf ( "NV_register_combiners NOT supported" );
        return 2;
    }
    initExtensions ();
    bumpMap     = createTexture2D            ( false,
                          "../Textures/Bumpmaps/FieldStone-DOT3.tga" );
    decalMap    = createTexture2D            ( true,
                          "../Textures/FieldStone.tga" );
    normCubeMap = createNormalizationCubemap   ( 32 );
    glutMainLoop ();
    return 0;
}
```

Реализация бликового (*specular*) освещения через механизм register combiner

С помощью механизма register combiner можно достаточно легко реализовать попиксельное бликовое (*specular*) освещение.

Формула для расчета интенсивности бликового освещения следующая:

$$I = I_{amb} + \max(0, (h, n))^p.$$

Здесь коэффициент I_{amb} задает фоновую освещенность, а h — бисектор единичного вектора на источник света l и на наблюдателя v: $h = \frac{l+v}{|l+v|}$ (рис. 9.15).

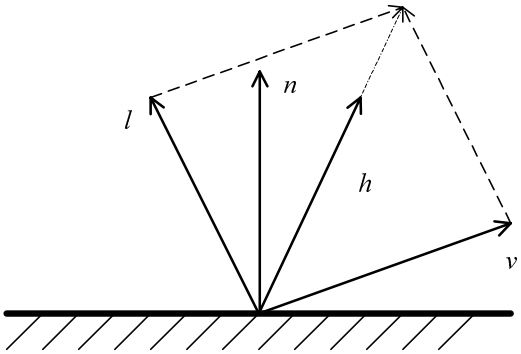


Рис. 9.15. Вычисление вектора *h*

Степень *p*, в которую возводится скалярное произведение, характеризует величину неровности поверхности (чем она больше, тем более резким получается блик).

Как и в случае диффузного освещения, в нулевом текстурном блоке разместим карту нормалей, а в первом — нормирующую кубическую текстуру, для которой вместо координат вектора *l* будем в каждой вершине задавать значения *h*. Тогда после интерполяции текстурных координат и применения их к нормирующей кубической текстуре мы получим для каждого пиксела выводимой грани нормированное значение вектора *h*.

К сожалению, через механизм register combiner достаточно сложно реализовать возведение в произвольную степень *p*.

Проще всего реализовать возведение в степень 8. Для этого нам понадобятся два general combiner. Первый из них конфигурируется аналогично случаю диффузного освещения, и на выходе он записывает в регистр `spare0.rgb` скалярное произведение (n, h). Второй — в переменные `A` и `B` записывает значение $\max(0, (n, h))$ и на выходе размещает покоординатное произведение этих переменных опять в `spare0.rgb`, т. е. там мы получаем уже величину $\max(0, (n, h))^2$.

Листинг 9.8 и рис. 9.16 иллюстрируют выполнение этих операций.

Листинг 9.8. Настройка блока register combiner для вычисления max(0, (n, h))[8]

```
                                    // A = max ( (n,h), 0 )
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_A_NV,
                    GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
```

```
                        // B = max ( (n,h), 0 )
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                        // setup output of (h,n)^2
glCombinerOutputNV ( GL_COMBINER1_NV, GL_RGB,
                     GL_SPARE0_NV,      // AB output
                     GL_DISCARD_NV,     // CD output
                     GL_DISCARD_NV,     // sum output
                     GL_NONE,           // no scale
                     GL_NONE,           // no bias
                     GL_FALSE,          // we do not need dot product
                     GL_FALSE, GL_FALSE );
```

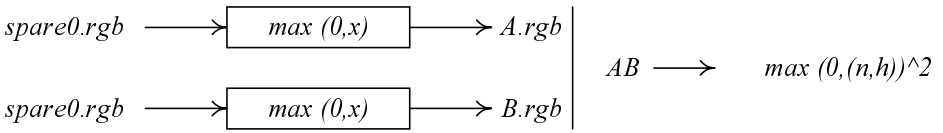


Рис. 9.16. Настройка второго блока general combiner

Чуть более сложной является настройка блока final combiner (рис. 9.17). Здесь в качестве переменных `A` и `B` выступает псевдорегистр `EF`, `C = 0`, `D = = constant-color0`, а в переменные `E` и `F` записывается значение из `spare0.rgb` (листинг 9.9).

Листинг 9.9. Настройка блока final combiner

```
                          // A.rgb = EF
glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_E_TIMES_F_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                          // B.rgb = EF
glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_E_TIMES_F_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                          // C = 0
glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
```

```
                          // D = constant_color1.rgb
glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_CONSTANT_COLOR1_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                          // E = spare0.rgb
glFinalCombinerInputNV ( GL_VARIABLE_E_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                          // F = spare0.rgb
glFinalCombinerInputNV ( GL_VARIABLE_F_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                          // G.alpa = 1
glFinalCombinerInputNV ( GL_VARIABLE_G_NV, GL_ZERO,
                         GL_UNSIGNED_INVERT_NV, GL_ALPHA );
```

В результате на выходе получается величина интенсивности, равная:

$$I = I_{amb} + \max(0, (h, n))^8.$$

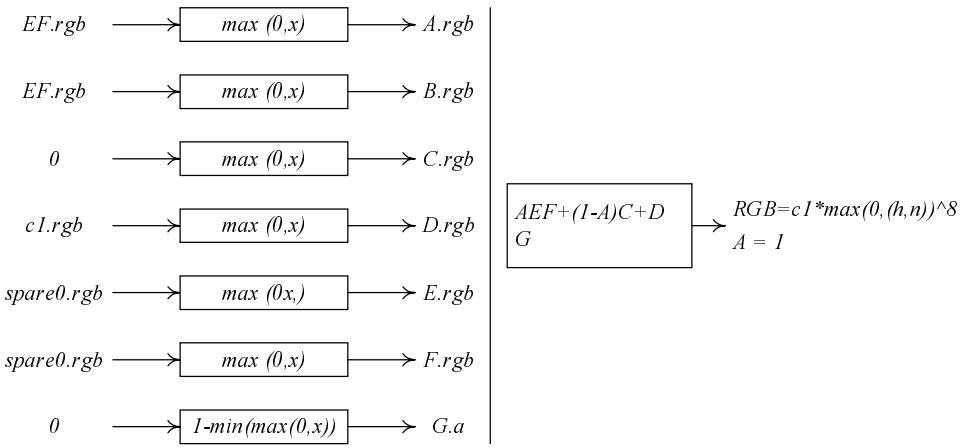


Рис. 9.17. Настройка блока final combiner

На рис. 9.18 вы видите результат работы предложенного алгоритма, полный код программы приведен в листинге 9.10.

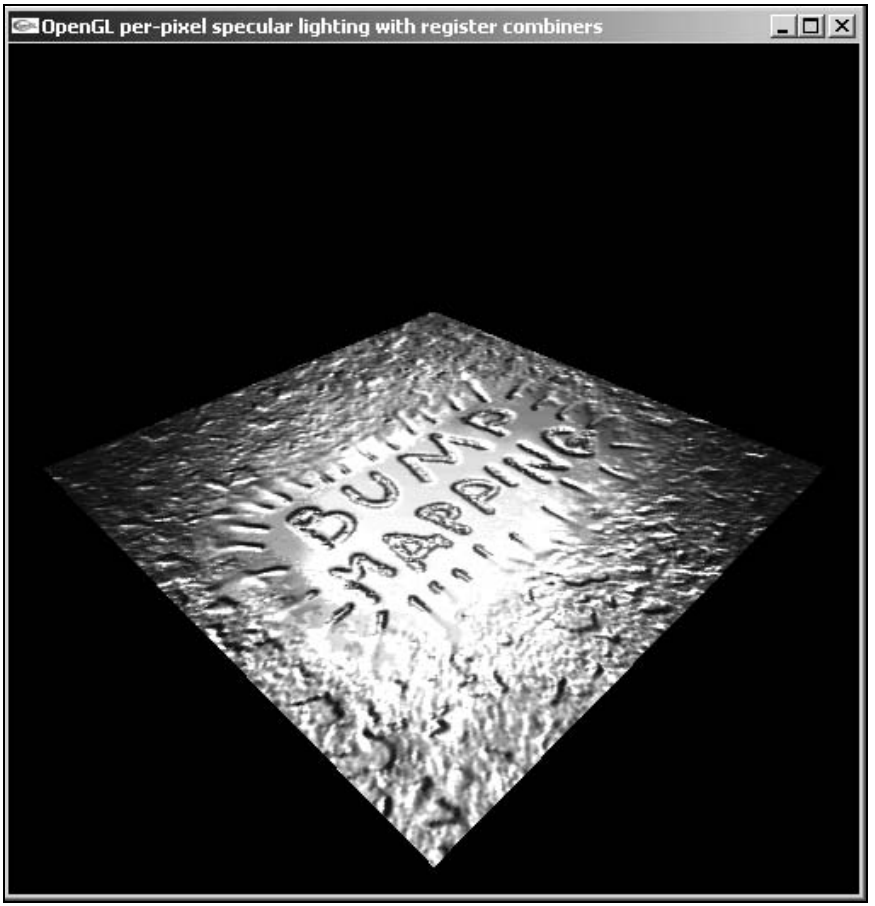


Рис. 9.18. Реализация попиксельного бликового освещения с показателем степени 8

Листинг 9.10. Реализация попиксельного бликового освещения с показателем степени 8

```
//
// Sample to to specular per-pixel lighting with Register Combiners
// in OpenGL
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
```

```
#include    <stdlib.h>
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
Vector3D    eye   ( 2, 2, 2 );      // camera position
Vector3D    light ( 0.5, 0, 1 );    // light position
unsigned    normCubeMap;            // normalization cubemap id
unsigned    bumpMap;                // normal map
unsigned    decalMap;               // decal (diffuse) texture
float       angle = 0;
Vector3D    v  [4];
Vector2D    t0 [4];
Vector3D    t1 [4];
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 ()
{
    float    lightColor   [4] = { 1, 1, 0, 1 };       // yellow light
    float    ambientColor [4] = { 0, 0, 0.5f, 1 };    // dark blue
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
                        // setup texture units
                        // bind bump (normal) map to
                        // texture unit 0
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, bumpMap );
                        // bind normalization cube map
                        // to texture unit 1
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnable           ( GL_TEXTURE_CUBE_MAP_ARB );
    glBindTexture      ( GL_TEXTURE_CUBE_MAP_ARB, normCubeMap );
```

```
                    // setup register combiners
glEnable                ( GL_REGISTER_COMBINERS_NV );
glCombinerParameteriNV  ( GL_NUM_GENERAL_COMBINERS_NV, 2 );
glCombinerParameterfvNV ( GL_CONSTANT_COLOR0_NV, lightColor );
glCombinerParameterfvNV ( GL_CONSTANT_COLOR1_NV, ambientColor );
                    // setup general combiner 0
                    // configure A = expand (tex0)
                    // (bumpmap)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
                    GL_TEXTURE0_ARB, GL_EXPAND_NORMAL_NV,
                    GL_RGB );
                    // configure B = expand (tex1)
                    // (norm. map)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                    GL_RGB );
                    // configure C = 0
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_C_NV,
                    GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // configure D = 0
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_D_NV,
                    GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // setup output of (h,n)
glCombinerOutputNV ( GL_COMBINER0_NV, GL_RGB,
                     GL_SPARE0_NV,        // AB output
                     GL_DISCARD_NV,       // CD output
                     GL_DISCARD_NV,       // sum output
                     GL_NONE,             // no scale
                     GL_NONE,             // no bias
                     GL_TRUE,             // AB = A dot B
                     GL_FALSE, GL_FALSE );
                    // now spare0.rgb = (h,n)
                    // setup general combiner 1
                    // A = max ( (n,h), 0 )
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_A_NV,
                    GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV,
                    GL_RGB );
```

```
                    // B = max ( (n,h), 0 )
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV,
                    GL_RGB );
                    // setup output of (h,n)^2
glCombinerOutputNV ( GL_COMBINER1_NV, GL_RGB,
                     GL_SPARE0_NV,        // AB output
                     GL_DISCARD_NV,       // CD output
                     GL_DISCARD_NV,       // sum output
                     GL_NONE,             // no scale
                     GL_NONE,             // no bias
                     GL_FALSE,            // we do not need here
                                          // dot product
            GL_FALSE, GL_FALSE );
                    // now spare0.rgb = (n,h) ^ 2
                    // configure final combiner
                    // A.rgb = EF
glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_E_TIMES_F_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // B.rgb = EF
glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_E_TIMES_F_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // C = 0
glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // D = constant_color1.rgb
glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_CONSTANT_COLOR1_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // E = spare0.rgb
glFinalCombinerInputNV ( GL_VARIABLE_E_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // F = spare0.rgb
glFinalCombinerInputNV ( GL_VARIABLE_F_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // G.alpa = 1
glFinalCombinerInputNV ( GL_VARIABLE_G_NV, GL_ZERO,
                         GL_UNSIGNED_INVERT_NV, GL_ALPHA );
```

```
                        // so at output we'll get
                        // RGB   = max ( (l, n), 0 )^8
                        // alpha = 1
                        // compute texture coordinates
                        // for normalization cibe map
    int    i;
    for ( i = 0; i < 4; i++ )
    {
        Vector3D    li = light - v [i];
        Vector3D    vi = eye   - v [i];
                        // compute vector h
        t1 [i] = li.normalize () + vi.normalize ();
    }
                        // now draw quad
    glBegin ( GL_QUADS );
    for ( i = 0; i < 4; i++ )
    {
        glMultiTexCoord2fv ( GL_TEXTURE0_ARB, t0 [i] );
        glMultiTexCoord3fv ( GL_TEXTURE1_ARB, t1 [i] );
        glVertex3fv        ( v [i]                    );
    }
    glEnd   ();
    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 key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )
```

```
            exit ( 0 );
}
void    animate ()
{
    angle  = 0.004f * glutGet ( GLUT_ELAPSED_TIME );
    light.x = 1.75*cos ( angle );
    light.y = 1.75*sin ( angle );
    light.z = 0.75 + 0.1 * sin ( angle / 3 );
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                        // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                        // create window
    int    win = glutCreateWindow ( "OpenGL per-pixel specular lighting
with register combiners" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init ();
    printfInfo ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_NV_register_combiners" ) )
    {
        printf ( "NV_register_combiners NOT supported.\n" );
        return 2;
    }
    initExtensions ();
```

```
                        // setup vertices
    v [0] = Vector3D ( -1.3, -1.3, 0 );
    v [1] = Vector3D (  1.3, -1.3, 0 );
    v [2] = Vector3D (  1.3,  1.3, 0 );
    v [3] = Vector3D ( -1.3,  1.3, 0 );
                        // setup texture coordinates
                        // for bump map
    t0 [0] = Vector2D ( 0, 0 );
    t0 [1] = Vector2D ( 1, 0 );
    t0 [2] = Vector2D ( 1, 1 );
    t0 [3] = Vector2D ( 0, 1 );
    bumpMap     = createTexture2D ( true,
                                    "../Textures/Bumpmaps/bump.tga" );
    decalMap    = createTexture2D ( true, "../Textures/block.bmp" );
    normCubeMap = createNormalizationCubemap ( 32 );
    glutMainLoop ();
    return 0;
}
```

К сожалению, значение степени, равное восьми, не всегда подходит, поэтому в ряде случаев приходится допускать различные приближения.

Достаточно хорошие результаты дает следующая приближенная формула:

$$\max (0, 4 \cdot (\max(0, (h, n)) - 0{,}75))^2).$$

При ее реализации в нулевом блоке general combiner, как и ранее, вычисляет значение (n, h) и записывает его в регистр `spare0.rgb`.

Следующий блок general combiner конфигурируется так (рис. 9.19):

```
A = max ( 0, (n,h) )
B = 1
C = -constant-color0
D = 1
```

Использование в качестве выхода `AB+CD` с масштабирующим коэффициентом `GL_SCALE_BY_FOUR_NV` дает в результате значение, равное $4 \cdot (\max(0, (n, h)) - 0{,}75)$. Листинг 9.11 содержит соответствующий фрагмент кода.

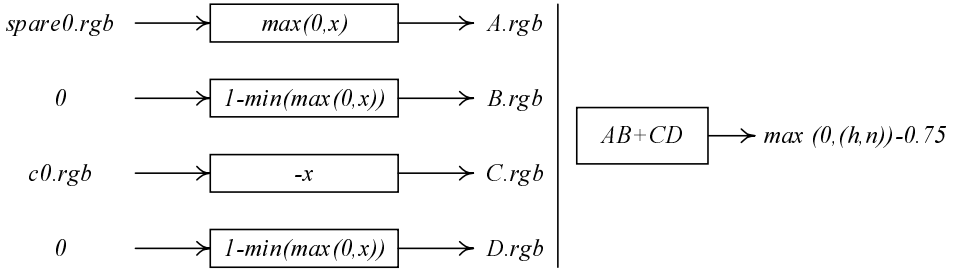


Рис. 9.19. Настройка второго блока general combiner

Листинг 9.11. Конфигурация второго блока general combiner

```
                        // A = max ( (n,h), 0 )
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_A_NV,
                    GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                        // B = 1
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_RGB );
                        // C = -constant color0.rgb
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_C_NV,
                    GL_CONSTANT_COLOR0_NV, GL_SIGNED_NEGATE_NV,
                    GL_RGB );
                        // D = 1
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_D_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_RGB );
                        // setup output of (h,n) - 0.75
glCombinerOutputNV ( GL_COMBINER1_NV, GL_RGB,
                     GL_DISCARD_NV,     // AB output
                     GL_DISCARD_NV,     // CD output
                     GL_SPARE0_NV,      // sum output
                     GL_SCALE_BY_FOUR_NV,
                     GL_NONE,           // no bias
                     GL_FALSE,          // we do not need dot product
                     GL_FALSE,
                     GL_FALSE );        // neither we need mux here
```

Все, что остается сделать блоку final combiner, — отсечь отрицательные значения и возвести результат в квадрат. Это иллюстрируют рис. 9.20 и листинг 9.12.

Листинг 9.12. Конфигурация блока final combiner

```
                              // A.rgb = max ( 0, spare0.rgb )
    glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_SPARE0_NV,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                              // B.rgb = max ( 0, spare0.rgb )
    glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_SPARE0_NV,
                           GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                              // C = 0
    glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                           GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                              // D = constant_color1.rgb
    glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_CONSTANT_COLOR1_NV,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                              // E = 0
    glFinalCombinerInputNV ( GL_VARIABLE_E_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                              // F = 0
    glFinalCombinerInputNV ( GL_VARIABLE_F_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                              // G.alpa = 1
    glFinalCombinerInputNV ( GL_VARIABLE_G_NV, GL_ZERO,
                             GL_UNSIGNED_INVERT_NV, GL_ALPHA );
```

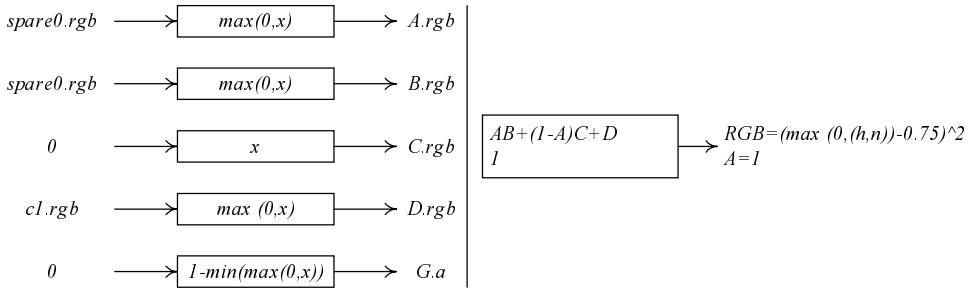


Рис. 9.20. Вариант настройки блока final combiner

На рис. 9.21 приводится изображение, полученное с помощью данного алгоритма.

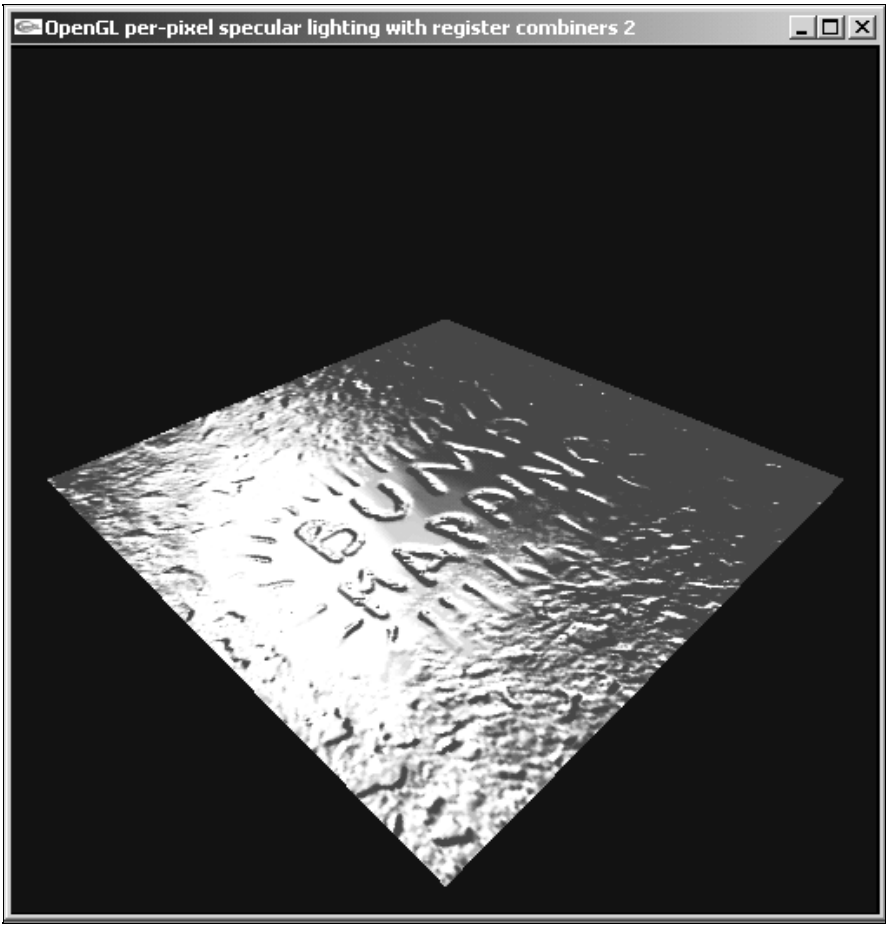


Рис. 9.21. Результат работы попиксельного бликового освещения с приближенным вычислением возведения в степень

Соответствующий исходный текст вы можете найти на прилагаемом к книге компакт-диске (файл rc-specular-2.cpp).

Аналогично случаю диффузного освещения строится изображение сложных тел, например, тора. Листинг 9.13 содержит описания соответствующих классов и структур.

Листинг 9.13. Класс `Torus` и основные структуры для реализации бликового освещения

```
struct    Vertex
{
    Vector3D  pos;              // position of vertex
    Vector2D  tex;              // texture coordinates
    Vector3D  n;                // unit normal
    Vector3D  t, b;             // tangent and binormal
    Vector3D  l;                // light vector in the tangent space
    Vector3D  h;
                                // map vector to tangent (TBN) space
    Vector3D mapToTangentSpace ( const Vector3D& v ) const
    {
        return Vector3D ( v & t, v & b, v & n );
    }
};
struct Face                     // triangular face
{
    int index [3];              // indices to Vertex array
};
class    Torus
{
    int      numRings;
    int      numSides;
    int      numVertices;
    int      numFaces;
    Vertex * vertices;
    Face   * faces;
public:
    Torus  ( float r1, float r2, int rings, int sides );
    ~Torus ()
    {
        delete [] vertices;
        delete [] faces;
    }
```

```
    void calcLightVectors ( const Vector3D& light,
                            const Vector3D& eye );
    void draw             ();
};
```

Обратите внимание, что в структуре `Vertex` теперь хранится не только вектор *l*, но и *h*.

Метод `calcLightVectors` получает на входе два параметра: положение источника света `light` и наблюдателя — `eye`. Вот соответствующий фрагмент кода:

```
void  Torus :: calcLightVectors ( const Vector3D& light,
                                  const Vector3D& eye )
{
                                // compute texture coordinates for
                                // normalization cube map
    for ( int i = 0; i < numVertices; i++ )
    {
        Vector3D  l = (light - vertices [i].pos).normalize ();
        Vector3D  v = (eye   - vertices [i].pos).normalize ();
        vertices [i].l = vertices [i].mapToTangentSpace ( l );
        vertices [i].h = vertices [i].mapToTangentSpace ( l + v );
    }
}
```

Во всех ранее рассмотренных примерах попиксельного освещения сам освещаемый объект был неподвижен, а перемещался только источник света. Рассмотрим теперь, как следует модифицировать алгоритм попиксельного освещения, чтобы осуществить преобразования освещаемого объекта.

Добавим в наш пример обработчик сообщений мыши, позволяющий поворачивать тор с ее помощью (листинг 9.14).

Листинг 9.14. Вариант обработчика сообщений мыши

```
Vector3D    rot ( 0, 0, 0 );
int        mouseOldX = 0;
int        mouseOldY = 0;
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;
    }
}
```

Тогда фрагмент кода для поворота тора на соответствующий угол будет таким:

```
   glMatrixMode   ( GL_MODELVIEW );
   glPushMatrix   ();
    glRotatef    ( rot.x, 1, 0, 0 );
    glRotatef    ( rot.y, 0, 1, 0 );
    glRotatef    ( rot.z, 0, 0, 1 );
torus.calcLightVectors ( light, eye, r );
    torus.draw             ();
    glPopMatrix     ();
```

Однако при повороте нам придется учитывать, что поворачиваются не только точки тора, но и базис касательного пространства (векторы *t*, *b* и *n*), как это видно из рис. 9.22.

Тогда для обеспечения возможности поворотов тора нам придется постоянно подвергать преобразованиям базисы касательного пространства для каж-

дой грани, что довольно неудобно, поскольку требует определенных вычислительных затрат.

На самом деле подобное преобразование большого числа векторов необязательно. Заметим, что вместо преобразования самого тора (вместе со всеми базисами касательного пространства) можно подвергнуть обратному преобразованию вектора положения наблюдателя и источника света (рис. 9.23).

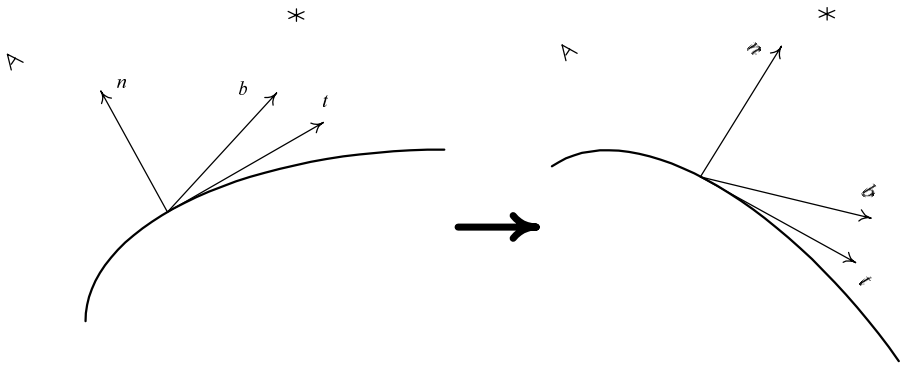


Рис. 9.22. Преобразование базиса касательного пространства при преобразовании самого объекта

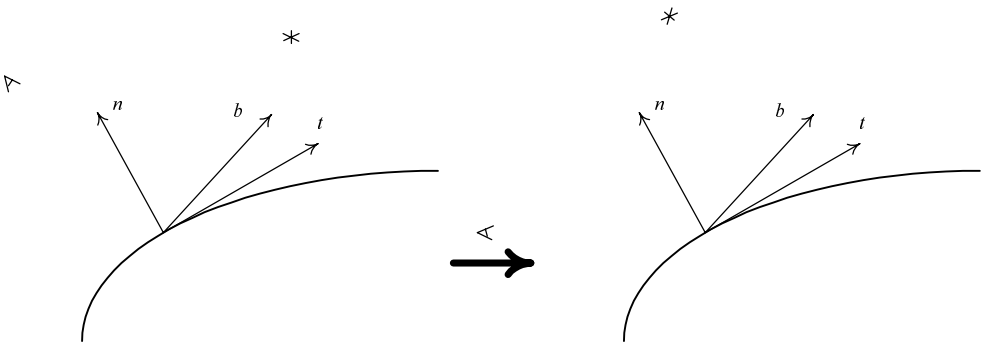


Рис. 9.23. Преобразование положения источника света и наблюдателя

Таким образом, достаточно преобразовать всего две точки, а все базисы остаются неизменными. Именно этот подход иллюстрирует листинг 9.14. На рис. 9.24 приводится изображение бликового освещения тора, полученное в результате работы этой программы. Как видно на приведенном скриншоте,

на выводимый тор, помимо бликов, накладывается и базовая текстура. Поэтому пример требует наличия как минимум трех текстурных блоков.

Рис. 9.24. Тор с бликовым освещением

Листинг 9.15. Построение изображения тора с попиксельным бликовым освещением и приближенным вычислением степени

```
//
// Sample to to specular per-pixel lighting with Register Combiners in
// OpenGL
//
#include    "libExt.h"
```

```
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Matrix4x4.h"
#include    "Torus.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
Vector3D    eye   ( 7, 5, 7 );        // camera position
Vector3D    light ( 5, 0, 4 );        // light position
unsigned    normCubeMap;              // normalization cubemap id
unsigned    bumpMap;                  // normal map
unsigned    decalMap;                 // decal (diffuse) texture
float       angle = 0;
Torus       torus ( 1, 3, 30, 30 );
Vector3D    rot ( 0, 0, 0 );
int         mouseOldX = 0;
int         mouseOldY = 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 ()
{
    float    bias         [4] = { 0.75, 0.75, 0.75, 0.75 };
    float    ambientColor [4] = { 0, 0, 0.5f, 1 };    // dark blue
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
                        // draw the light
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glDisable          ( GL_REGISTER_COMBINERS_NV );
    glTranslatef       ( light.x, light.y, light.z );
```

```
glActiveTextureARB ( GL_TEXTURE0_ARB );
glDisable          ( GL_TEXTURE_2D );
glActiveTextureARB ( GL_TEXTURE1_ARB );
glDisable          ( GL_TEXTURE_CUBE_MAP_ARB );
glActiveTextureARB ( GL_TEXTURE2_ARB );
glDisable          ( GL_TEXTURE_2D );
glColor4f          ( 1, 1, 1, 1 );
glutSolidSphere    ( 0.1f, 15, 15 );
glPopMatrix        ();
                    // setup texture units
                    // bind bump (normal) map to
                    // texture unit 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glEnable           ( GL_TEXTURE_2D );
glBindTexture      ( GL_TEXTURE_2D, bumpMap );
                    // bind normalization cube map
                    // to texture unit 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glEnable           ( GL_TEXTURE_CUBE_MAP_ARB );
glBindTexture      ( GL_TEXTURE_CUBE_MAP_ARB, normCubeMap );
glActiveTextureARB ( GL_TEXTURE2_ARB );
glEnable           ( GL_TEXTURE_2D );
glBindTexture      ( GL_TEXTURE_2D, decalMap );
glMatrixMode       ( GL_TEXTURE );
glLoadIdentity     ();
glScalef           ( 1, 2, 1 );
glMatrixMode       ( GL_MODELVIEW );
                    // setup register combiners
glEnable               ( GL_REGISTER_COMBINERS_NV );
glCombinerParameteriNV  ( GL_NUM_GENERAL_COMBINERS_NV, 2 );
glCombinerParameterfvNV ( GL_CONSTANT_COLOR0_NV, bias );
glCombinerParameterfvNV ( GL_CONSTANT_COLOR1_NV, ambientColor );
                    // setup general combiner 0
                    // configure A = expand (tex0)
                    // (bumpmap)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
                    GL_TEXTURE0_ARB, GL_EXPAND_NORMAL_NV,
                    GL_RGB );
```

```
                    // configure B = expand (tex1)
                    // (norm. map)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                    GL_RGB );
                    // configure C = 0
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_C_NV,
                    GL_ZERO,  GL_UNSIGNED_IDENTITY_NV, GL_RGB );

                    // configure D = 0
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_D_NV,
                    GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // setup output of (h,n)
glCombinerOutputNV ( GL_COMBINER0_NV, GL_RGB,
                     GL_SPARE0_NV,         // AB output
                     GL_DISCARD_NV,        // CD output
                     GL_DISCARD_NV,        // sum output
                     GL_NONE,              // no scale
                     GL_NONE,              // no bias
                     GL_TRUE,              // AB = A dot B
                     GL_FALSE, GL_FALSE );
                    // now spare0.rgb = (h,n)
                    // setup general combiner 1
                    // A = max ( (n,h), 0 )
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_A_NV,
                    GL_SPARE0_NV, L_UNSIGNED_IDENTITY_NV,
                    GL_RGB );
                    // B = 1
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_RGB );
                    // C = -constant color0.rgb
                    // (-0.75)
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_C_NV,
                    GL_CONSTANT_COLOR0_NV, GL_SIGNED_NEGATE_NV,
                    GL_RGB );
                    // configure D = 1
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_D_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_RGB );
```

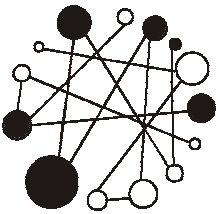
```
                    // setup output of (h,n) - 0.75
glCombinerOutputNV ( GL_COMBINER1_NV, GL_RGB,
                     GL_DISCARD_NV,        // AB output
                     GL_DISCARD_NV,        // CD output
                     GL_SPARE0_NV,         // sum output
                     GL_SCALE_BY_FOUR_NV,  // scale = 4
                     GL_NONE,              // no bias
                     GL_FALSE,             // we do not need here
                                           // dot product
                     GL_FALSE,
          GL_FALSE );                      // neither we need mux
                                           // here
                    // now spare0.rgb =
                    //          4*((n,h) - 0.75)
                    // configure final combiner
                    // A.rgb = max (0, spare0.rgb)
glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // B.rgb = max (0, spare0.rgb)
glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // C = 0
glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // D = texture2
glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_TEXTURE2_ARB,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // E = 0
glFinalCombinerInputNV ( GL_VARIABLE_E_NV, GL_ZERO,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // F = 0
glFinalCombinerInputNV ( GL_VARIABLE_F_NV, GL_ZERO,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // G.alpa = 1
glFinalCombinerInputNV ( GL_VARIABLE_G_NV, GL_ZERO,
                         GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                    // so at output we'll get
                    // RGB   = max ( 0,
```

```
                        //     max ((h,n),0)-0.75)^2)
                        // alpha = 1
                        // now setup rotation
    glMatrixMode   ( GL_MODELVIEW );
    glPushMatrix   ();
       glRotatef    ( rot.x, 1, 0, 0 );
       glRotatef    ( rot.y, 0, 1, 0 );
       glRotatef    ( rot.z, 0, 0, 1 );
                        // transform light and eye
                        // positions with inverse
                        // rotation matrix
    Matrix4x4    rx = Matrix4x4 :: rotateX ( -rot.x * M_PI / 180 );
    Matrix4x4    ry = Matrix4x4 :: rotateY ( -rot.y * M_PI / 180  );
    Matrix4x4    rz = Matrix4x4 :: rotateZ ( -rot.z * M_PI / 180  );
    Matrix4x4    r  = rx * ry * rz;
    r.invert ();
    torus.calcLightVectors ( light, eye, r );
    torus.draw             ();
    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 );
    light.x = 2*cos ( angle );
    light.y = 2*sin ( angle );
    light.z = 3 + 0.3 * sin ( angle / 3 );
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                        // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
```

```
                        // create window
    int    win = glutCreateWindow ( "OpenGL per-pixel specular lit torus
with register combiners" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    glutMouseFunc    ( mouse   );
    glutMotionFunc   ( motion  );
    init ();
    printfInfo ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_NV_register_combiners" ) )
    {
        printf ( "NV_register_combiners NOT supported" );
        return 2;
    }
    initExtensions ();
    bumpMap     = createNormalMapFromHeightMap ( false,
                          "../Textures/BumpMaps/light06.tga", 1 );
    decalMap    = createTexture2D ( true,  "../Textures/block.bmp" );
    normCubeMap = createNormalizationCubemap   ( 32 );
    glutMainLoop ();
    return 0;
}
```

Глава 10



Вершинные и индексные буферы и работа с ними при помощи расширения ARB_vertex_buffer_object

Одним из самых "узких" мест в работе с современным графическим ускорителем является передача ему данных — текстур, вершин, нормалей и т. п.

Для повышения быстродействия следует уменьшить количество запросов на передачу данных и передавать их как можно бóльшими частями.

Если с текстурами все достаточно просто, они один раз загружаются в память графического ускорителя и больше не изменяются, то с геометрическими данными (координаты вершин, нормали, текстурные координаты) дело обстоит значительно хуже.

Стандартный способ передачи данных через команды `glVertex`, `glNormal` и т. п. является крайне неэффективным, поскольку осуществляется очень маленькими частями и путем большого числа вызовов.

Можно заметно повысить эффективность обмена данными при помощи так называемых вершинных массивов (*vertex arrays*). При этом данные хранятся в памяти CPU, и ускорителю передаются только указатель на них и размер (тогда сам драйвер определяет способ передачи данных графическому ускорителю и делает это большими блоками).

В листинге 10.1 приводится вариант примера с тором из *главы 9* с использованием вершинных массивов. В данном случае изменению подлежат лишь сам класс `Torus` и его реализация.

Листинг 10.1. Реализация класса `Torus` с использованием вершинных массивов

```
#include    "libExt.h"
#include    "Torus-1.h"
```

```
#include    "Matrix4x4.h"
Torus :: Torus ( float r1, float r2, int rings, int sides )
{
    float    ringDelta = 2.0 * M_PI / rings;
    float    sideDelta = 2.0 * M_PI / sides;
    float    invRings  = 1.0f / (float) rings;
    float    invSides  = 1.0f / (float) sides;
    int      i, j;

    numRings = rings;
    numSides = sides;
    vertices = new Vertex [(numSides+1)*(numRings+1)];
    faces    = new Face   [numSides*numRings*2];
                    // create vertices
    int    index = 0;
    for ( i = 0; i <= rings; i++ )
    {
        float theta    = i * ringDelta;
        float cosTheta = cos ( theta );
        float sinTheta = sin ( theta );
        for ( j = 0; j <= sides; j++ )
        {
            float phi    = j * sideDelta;
            float cosPhi = cos ( phi );
            float sinPhi = sin ( phi );
            float dist   = r2 + r1 * cosPhi;
                           // (x,y,z) below is the parametric
                           // equation for a torus where
                           // theta and phi both vary
                           // from 0 to pi.
            vertices [index].pos.x = cosTheta * dist;
            vertices [index].pos.y = -sinTheta * dist;
            vertices [index].pos.z = r1 * sinPhi;
                           // now setup texture coordinates
            vertices [index].tex.x = j * invSides;
            vertices [index].tex.y = i * invRings;
                           // normalize the partial derivative of
                           // (x,y,z) with respect to theta.
```

```
            float    dxdtheta = -sinTheta * dist;
            float    dydtheta = -cosTheta * dist;
            float    d        = 1 / sqrt ( dxdtheta*dxdtheta +
                                          dydtheta*dydtheta );
            vertices [index].t.x = dxdtheta * d;
            vertices [index].t.y = dydtheta * d;
            vertices [index].t.z = 0;

                        // compute unit normal
            vertices [index].n.x =  cosTheta * cosPhi;
            vertices [index].n.y = -sinTheta * cosPhi;
            vertices [index].n.z = sinPhi;
                        // B = N cross T
            vertices [index].b = vertices [index].n ^
                                 vertices [index].t;
            index++;
        }
    }
    numVertices = index;
                                                // Create faces
    index = 0;
    for ( i = 0; i < rings; i++ )
        for ( j = 0; j < sides; j++, index += 2 )
        {
            faces [index].index [2] = i*(sides+1) + j;
            faces [index].index [1] = (i+1)*(sides+1) + j;
            faces [index].index [0] = (i+1)*(sides+1) + j + 1;
            faces [index+1].index [2] = i*(sides+1) + j;
            faces [index+1].index [1] = (i+1)*(sides+1) + j + 1;
            faces [index+1].index [0] = i*(sides+1) + j + 1;
        }
    numFaces = index;
}
void    Torus :: calcLightVectors ( const Vector3D& light )
{
                        // compute texture coordinates
                        // for normalization cibe map
    for ( int i = 0; i < numVertices; i++ )
```

```
        vertices [i].l = vertices [i].mapToTangentSpace ( light -
                                                    vertices [i].pos );
}
void    Torus :: calcLightVectors ( const Vector3D& light,
                                    const Vector3D& eye,
                                    const Matrix4x4& m )
{
    Vector3D    lt = m.transformPoint ( light );
    Vector3D    et = m.transformPoint ( eye   );
                        // compute texture coordinates
                        // for normalization cube map
    for ( int i = 0; i < numVertices; i++ )
    {
        Vector3D    l = (lt - vertices [i].pos).normalize ();
        Vector3D    v = (et - vertices [i].pos).normalize ();
        vertices [i].l = vertices [i].mapToTangentSpace ( l );
        vertices [i].h = vertices [i].mapToTangentSpace ( l + v );
    }
}
void    Torus :: draw ()
{
    glPushClientAttrib  ( GL_CLIENT_VERTEX_ARRAY_BIT );
                           // setup vertices array
    glEnableClientState ( GL_VERTEX_ARRAY );
    glVertexPointer ( 3, GL_FLOAT, sizeof ( Vertex ),
                         &vertices [0].pos );
                           // setup texcoord0 array
    glClientActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnableClientState      ( GL_TEXTURE_COORD_ARRAY );
    glTexCoordPointer        ( 2, GL_FLOAT, sizeof ( Vertex ),
                                  &vertices [0].tex );
                           // setup l array
    glClientActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnableClientState      ( GL_TEXTURE_COORD_ARRAY );
    glTexCoordPointer        ( 3, GL_FLOAT, sizeof ( Vertex ),
                                  &vertices [0].h );
                           // setup tex0 array
    glClientActiveTextureARB ( GL_TEXTURE2_ARB );
    glEnableClientState      ( GL_TEXTURE_COORD_ARRAY );
```

```
    glTexCoordPointer        ( 2, GL_FLOAT, sizeof ( Vertex ),
                                 &vertices [0].tex );
    glDrawElements ( GL_TRIANGLES, 3*numFaces, GL_UNSIGNED_INT,
                        &faces [0] );
    glPopClientAttrib ();
}
```

При таком способе передача осуществляется сразу большими блоками, количество обращений к графическому ускорителю (GPU) заметно сокращается. Это приводит к гораздо более эффективной его работе и повышению быстродействия программы.

Однако этот способ требует постоянной передачи массивов данных от CPU к GPU. Каждый вызов, содержащий указатель на массив, приводит к передаче его графическому ускорителю, и именно эта постоянная необходимость передачи большого объема данных ограничивает эффективность приложения.

В то же время далеко не все передаваемые данные действительно изменяются при каждом обращении, достаточно большой объем информации или вообще остается постоянным (данные для статических объектов, текстурные координаты и т. п.), или изменяется сравнительно редко и небольшими частями. Поэтому было бы гораздо эффективнее сразу загрузить такие данные в память GPU, после чего их использование графическим ускорителем не потребует обращения к CPU.

Подобную возможность предоставляет пользователю расширение ARB_vertex_buffer_object. Оно позволяет кэшировать (хранить) различные типы данных (как информацию о вершинах, так и индексы в массив вершин) в быстрой памяти графического ускорителя. Для этого блоки данных помещаются в так называемые вершинные буферы (*vertex buffer objects*), при этом каждый такой буфер представляет собой просто массив байтов.

Для работы с вершинным буфером пользователь может выбрать два способа. При первом данные через команды OpenGL (`glBufferDataARB`, `glBufferSubDataARB`, `glGetBufferSubDataARB`) передаются сразу большими блоками в (из) GPU.

Второй способ — обмен через указатель на данные. При этом пользователь запрашивает отображение (*mapping*) вершинного буфера в оперативную память CPU и получает указатель на область памяти, в которой находятся данные вершинного буфера. Далее пользователь может работать с данными через этот указатель как с обычной памятью, но по завершении работы необходимо закрыть отображение (*unmap*) буфера в память, после чего указатель уже перестает указывать на данные, а вершинный буфер снова может быть источником данных для графического ускорителя.

Вершинный буфер доступен в любой команде OpenGL, принимающей указатель на данные. Такой механизм позволяет GPU более эффективно работать с памятью, повышая быстродействие приложения в целом.

Перед началом работы с вершинными буферами следует проверить поддержку данного расширения. Это может быть сделано следующим запросом:

```
isExtensionSupported ( "GL_ARB_vertex_buffer_object" )
```

Каждый вершинный буфер идентифицируется при помощи беззнакового целого числа (идентификатора), причем нулевое значение зарезервировано.

Вторым шагом в работе с вершинным буфером (после проверки поддержки расширения) является получение свободного идентификатора, который будет в дальнейшем обозначать этот буфер.

Получение группы свободных идентификаторов осуществляется командой

```
void glGenBuffersARB ( GLsizei n, GLuint * buffers );
```

Этот запрос выделяет `n` свободных идентификаторов и записывает их в массив `buffers`.

Следующим шагом (обратите внимание на аналогию с текстурами) является выбор вершинного буфера как текущего (*bound*). После этого его можно проинициализировать или использовать в качестве источника данных. Для этого имеется команда

```
void glBindBufferARB ( GLenum target, GLuint buffer );
```

Параметр `target` задает тип данных, которые будет содержать буфер: информацию о вершинах (`GL_ARRARY_BUFFER_ARB`) или индексы в другие массивы (`GL_ELEMENTS_ARRAY_BUFFER_ARB`). Вторым параметром является идентификатор буфера (`buffer`), который станет текущим. Если в качестве идентификатора передать нулевое значение, то это будет означать, что текущего вершинного буфера нет.

Для уничтожения уже существующего вершинного буфера предусмотрена команда

```
void glDeleteBuffersARB ( GLsizei n, const GLuint * buffers );
```

Она уничтожает сразу `n` буферов, номера которых содержатся в массиве `buffers`.

После того как получен свободный идентификатор для вершинного буфера и он сделан текущим, буфер следует проинициализировать и наполнить данными. Это делается командой

```
void glBufferDataARB ( GLenum target, GLsizeiptrARB size,
                       const void * data, GLenum usage );
```

Параметр `target` имеет тот же смысл, что и в команде `glBindBufferARB`. Параметры `size` и `data` задают размер буфера в байтах и указатель на область памяти, содержащую значения, необходимые для инициализации данного вершинного буфера.

Обратите внимание на выбор типа `GLsizeiptrARB` (и `GLintptrARB`) вместо обычных целочисленных. Это связано с поддержкой 64-битовых значений.

Последний параметр, `usage`, несет в себе информацию о предполагаемом использовании буфера. Эта информация является лишь "намеком" и служит для того, чтобы система могла более эффективно управлять выделением памяти для вершинного буфера. Данный параметр никак не ограничивает пользователя в работе с буфером и предназначен лишь для целей оптимизации. Он может принимать следующие значения: `GL_STREAM_DRAW_ARB`, `GL_DYNAMIC_DRAW_ARB`, `GL_STATIC_DRAW_ARB`, `GL_STREAM_READ_ARB`, `GL_DYNAMIC_READ_ARB`, `GL_STATIC_READ_ARB`, `GL_STREAM_COPY_ARB`, `GL_DYNAMIC_COPY_ARB` и `GL_STATIC_COPY_ARB`.

Каждая из этих констант несет в себе информацию о том, для чего будет предназначен буфер (`draw`, `read`, `copy`), и о частоте изменения его содержимого (`stream`, `dynamic`, `static`).

Таблицы 10.1 и 10.2 разъясняют смысл указанных констант.

Таблица 10.1. *Цель использования буфера*

Назначение	**Комментарий**
`DRAW`	Буфер будет использоваться для передачи данных GPU, например, для вывода объектов
`READ`	Буфер будет использоваться пользователем для чтения из него. Важна скорость для чтения данных CPU
`COPY`	Будет использоваться как для чтения данных из GPU, так и для вывода объектов

Таблица 10.2. *Частота изменения содержимого буфера*

Частота обновления/ использования	**Комментарий**
`STREAM`	Предполагается, что на каждый вывод данных будет их изменение
`DYNAMIC`	Предполагается, что будет частое использование и изменение содержимого буфера
`STATIC`	Предполагается, что данные будут один раз заданы и потом много раз использованы

Для изменения данных в уже существующем буфере служит команда

```
void glBufferSubDataARB ( GLenum target, GLintptrARB offs,
                          GLsizeiptrARB size, const void * data );
```

Параметр `target` имеет тот же смысл, что и ранее, `offs` и `size` задают начало и размер изменяемого блока данных внутри вершинного буфера, а `data` — указывает на область памяти, содержащую новые значения.

Чтение блока данных из вершинного буфера можно выполнить командой

```
void glGetBufferSubDataARB ( GLenum target, GLintptrARB offs,
                             GLsizeiptrARB size, void * data );
```

Она копирует блок данных из вершинного буфера, начиная со смещения `offs` размером `size` в область памяти, заданную параметром `data`.

Вторым способом доступа к содержимому вершинного буфера является его отображение (*mapping*) в адресное пространство приложения.

При этом пользователь получает указатель на область памяти, содержащий образ вершинного буфера, и может свободно использовать данный указатель для работы с содержимым буфера. При этом указатель нельзя передавать в команды OpenGL в качестве параметра, команда `glBufferSubDataARB` для изменения содержимого буфера также недопустима. Буфер не может выступать и как источник данных в командах OpenGL.

По окончании работы с буфером следует закрыть отображение (*unmap*), после чего указатель перестает быть доступным, и вершинный буфер может быть снова источником данных для GPU.

Для отображения текущего вершинного буфера в системную память служит команда

```
void * glMapBufferARB ( GLenum target, GLenum access );
```

Параметр `target` имеет то же смысл, что и ранее. Параметр `access` определяет, какие именно операции планируется проводить с данными через полученный указатель. Допустимыми значениями являются `GL_READ_ONLY_ARB` (только чтение), `GL_WRITE_ONLY_ARB` (только запись) и `GL_READ_WRITE_ARB` (и чтение, и запись). Этот параметр позволяет системе оптимизировать способ отображения.

Если отображение буфера прошло успешно, то возвращается ненулевой указатель на блок памяти, с которым можно работать как в самим вершинным буфером.

Для закрытия отображения предназначена команда

```
GLboolean glUnmapBufferARB ( GLenum target );
```

После ее выполнения полученный ранее указатель использовать нельзя.

Если закрытие отображения прошло успешно, то функция `glUnmapBufferARB` возвращает значение `GL_TRUE`. Значение `GL_FALSE` означает, что содержимое буфера было необратимо повреждено за то время, пока буфер был отображен (например, из-за переключения разрешения монитора или изменения других параметров оконной системы). В этом случае содержимое буфера может оказаться неопределенным.

С другой стороны, гарантируется, что подобное повреждение может произойти, только пока буфер отображен в системную память.

Для проверки того, является ли беззнаковое число идентификатором какого-либо вершинного буфера, служит команда

```
GLboolean glIsBufferARB ( GLuint buffer );
```

Вершинный буфер пригоден для задания данных во всех командах OpenGL, где передается указатель на блок памяти с данными (например, `glVectorPointer` или `glDrawElements`).

Для работы с содержимым вершинного буфера его следует сделать текущим (при помощи команды `glBindBufferARB`), после этого в команде, принимающей указатель на данные, вместо указателя передается смещение от начала вершинного буфера до начала требуемых данных в байтах:

```
glBindBufferARB ( GL_ARRAY_BUFFER_ARB, bufId );
glVertexPointer ( 3, GL_FLOAT, 0, 0 );
```

В приведенном примере вершинный буфер с идентификатором `bufId` является источником координат вершин в команде `glVertexPointer`. При этом данные начинаются прямо с начала вершинного буфера.

Если активным был выбран буфер с нулевым идентификатором, то это означает, что текущего вершинного буфера нет, и команды сохраняют стандартное толкование аргументов. Таким образом, можно сохранить возможность старого способа работы (например, через вершинные массивы) наряду с применением вершинных буферов.

Обратите внимание, что в качестве источников разных данных можно указать различные вершинные массивы, например, часто изменяемые координаты вершин хранить в одном массиве, а цвета и текстурные координаты — в другом. Это позволит уменьшить объем передаваемых данных, поскольку будут передаваться только данные из часто изменяемого буфера, а остальные могут все время находиться в памяти GPU.

В следующем примере (листинги 10.2 и 10.3) создается один вершинный массив, в котором хранятся все данные для вершин. Обратите внимание, что из-за определения векторов *l* и *h* в виде текстурных координат возникает необходимость изменения вершинных данных при каждом выводе тора.

Еще один вершинный массив предназначен для хранения индексов вершин для всех граней тора. Заметьте, что этот массив ни разу не изменяется с момента своего создания.

Листинг 10.2. Описание класса `Torus` при использовании одного вершинного буфера

```
class    Torus
{
    int      numRings;
    int      numSides;
    int      numVertices;
    int      numFaces;
    Vertex * vertices;
    Face   * faces;
    unsigned vertexId;            // id of vertex VBO
public:
    Torus  ( float r1, float r2, int rings, int sides );
    ~Torus ()
    {
        delete [] vertices;
        delete [] faces;
        glDeleteBuffersARB ( 1, &vertexId );
    }
    void    setupBuffers     ();
    void    calcLightVectors ( const Vector3D& light );
    void    calcLightVectors ( const Vector3D& light,
                               const Vector3D& eye,
                               const Matrix4x4& m );
    void    draw             ();
};
```

Листинг 10.3. Реализация класса `Torus` при использовании одного вершинного буфера

```
#include    "libExt.h"
#include    "Torus-2.h"
#include    "Matrix4x4.h"
Torus :: Torus ( float r1, float r2, int rings, int sides )
```

```
{
    float   ringDelta = 2.0 * M_PI / rings;
    float   sideDelta = 2.0 * M_PI / sides;
    float   invRings  = 1.0f / (float) rings;
    float   invSides  = 1.0f / (float) sides;
    int     i, j;
    numRings = rings;
    numSides = sides;
    vertices = new Vertex [(numSides+1)*(numRings+1)];
    faces    = new Face   [numSides*numRings*2];
                              // create vertices
    int index = 0;
    for ( i = 0; i <= rings; i++ )
    {
        float theta    = i * ringDelta;
        float cosTheta = cos ( theta );
        float sinTheta = sin ( theta );
        for ( j = 0; j <= sides; j++ )
        {
            float phi    = j * sideDelta;
            float cosPhi = cos ( phi );
            float sinPhi = sin ( phi );
            float dist = r2 + r1 * cosPhi;
        // (x,y,z) below is the parametric equation for a torus
        // when theta and phi both vary from 0 to pi.
                    // x = cos(theta) * (R + r * cos(phi))
                    // y = -sin(theta) * (R + r * cos(phi))
                    // z = r * sin(phi)
            vertices [index].pos.x = cosTheta * dist;
            vertices [index].pos.y = -sinTheta * dist;
            vertices [index].pos.z = r1 * sinPhi;
      // now setup texture coordinates
            vertices [index].tex.x = j * invSides;
            vertices [index].tex.y = i * invRings;
       // T = normalize([dx/dtheta,dy/dtheta,dz/dtheta])
            float   dxdtheta = -sinTheta * dist;
            float   dydtheta = -cosTheta * dist;
            float   d        = 1 / sqrt ( dxdtheta*dxdtheta +
                                          dydtheta*dydtheta );
```

```
            vertices [index].t.x = dxdtheta * d;
            vertices [index].t.y = dydtheta * d;
            vertices [index].t.z = 0;
        // compute unit normal
            vertices [index].n.x =  cosTheta * cosPhi;
            vertices [index].n.y = -sinTheta * cosPhi;
            vertices [index].n.z = sinPhi;
        // B = N cross T
            vertices [index].b = vertices [index].n ^ vertices [index].t;
            index++;
        }
    }
    numVertices = index;
                                            // Create faces
    index = 0;
    for ( i = 0; i < rings; i++ )
        for ( j = 0; j < sides; j++, index += 2 )
        {
            faces [index].index [2] = i*(sides+1) + j;
            faces [index].index [1] = (i+1)*(sides+1) + j;
            faces [index].index [0] = (i+1)*(sides+1) + j + 1;
            faces [index+1].index [2] = i*(sides+1) + j;
            faces [index+1].index [1] = (i+1)*(sides+1) + j + 1;
            faces [index+1].index [0] = i*(sides+1) + j + 1;
        }
    numFaces = index;
}
void    Torus :: setupBuffers ()
{
                                            // create vertex data VBO
    glGenBuffersARB ( 1, &vertexId );
    glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vertexId );
    glBufferDataARB ( GL_ARRAY_BUFFER_ARB,
                      numVertices * sizeof ( Vertex ), vertices,
                      GL_STATIC_DRAW_ARB );
                                           // unbind buffers
    glBindBufferARB ( GL_ARRAY_BUFFER_ARB, 0 );
}
```

```
void    Torus :: calcLightVectors ( const Vector3D& light )
{
                                    // compute texture coordinates for
                                   // normalization cibe map
    for ( int i = 0; i < numVertices; i++ )
        vertices [i].l = vertices [i].mapToTangentSpace ( light -
                                                    vertices [i].pos );
}
void    Torus :: calcLightVectors ( const Vector3D& light,
                                    const Vector3D& eye,
                                    const Matrix4x4& m )
{
    Vector3D    lt = m.transformPoint ( light );
    Vector3D    et = m.transformPoint ( eye   );
                                    // compute texture coordinates for
                                    // normalization cube map
    for ( int i = 0; i < numVertices; i++ )
    {
        Vector3D    l = (lt - vertices [i].pos).normalize ();
        Vector3D    v = (et - vertices [i].pos).normalize ();
        vertices [i].l = vertices [i].mapToTangentSpace ( l );
        vertices [i].h = vertices [i].mapToTangentSpace ( l + v );
    }
}
void    Torus :: draw ()
{
    int vertexStride = sizeof ( Vertex );
    int texOffs      = ((int)&vertices [0].tex) - ((int)&vertices [0]);
    int lOffs        = ((int)&vertices [0].h) -   ((int)&vertices [0]);
    glBindBufferARB     ( GL_ARRAY_BUFFER_ARB, vertexId );
                                        // update buffer data
    glBufferSubDataARB ( GL_ARRAY_BUFFER_ARB, 0,
                         numVertices * sizeof ( Vertex ), vertices );
                                        // draw it now
    glPushClientAttrib  ( GL_CLIENT_VERTEX_ARRAY_BIT );
    glEnableClientState ( GL_TEXTURE_COORD_ARRAY );
    glEnableClientState ( GL_VERTEX_ARRAY );
    glVertexPointer     ( 3, GL_FLOAT, vertexStride, (void *)0 );
```

```
    glClientActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnableClientState      ( GL_TEXTURE_COORD_ARRAY );
    glTexCoordPointer        ( 2, GL_FLOAT, vertexStride,
                               (void *)texOffs );
    glClientActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnableClientState      ( GL_TEXTURE_COORD_ARRAY );
    glTexCoordPointer        ( 3, GL_FLOAT, vertexStride,
                               (void *)lOffs );
    glClientActiveTextureARB ( GL_TEXTURE2_ARB );
    glEnableClientState      ( GL_TEXTURE_COORD_ARRAY );
    glTexCoordPointer        ( 2, GL_FLOAT, vertexStride,
                               (void *)texOffs );
                                          // request draw
    glDrawElements ( GL_TRIANGLES, 3*numFaces, GL_UNSIGNED_INT,
                     &faces [0] );
                                          // unbind array buffer
    glBindBufferARB ( GL_ARRAY_BUFFER_ARB, 0 );
    glPopClientAttrib ();
}
```

Приведенная выше реализация страдает следующим недостатком: при каждом выводе тора весь массив индексов заново передается графическому ускорителю, несмотря на то, что никаких изменений в нем не происходит.

Поэтому можно создать еще один буфер (буфер для индексов), имеющий тип `GL_ELEMENT_ARRAY_BUFFER_ARB`. В него один раз будут записаны индексы, и далее они останутся неизменными, т. е. идеально подходит тип `GL_STATIC_DRAW_ARB`.

Это позволит сократить объем передаваемых графическому ускорителю данных (в дальнейшем нам удастся еще больше уменьшить объем этих данных за счет их непосредственного вычисления в GPU при помощи вершинных программ).

Листинги 10.4 и 10.5 содержат исходные тексты для соответствующей реализации.

Листинг 10.4. Описание класса `Torus` при использовании двух вершинных массивов

```
//
// Simple torus class
//
```

```
#include    "Vector3D.h"
#include    "Vector2D.h"
#ifndef     __TORUS__
#define     __TORUS__
struct    Vertex
{
    Vector3D    pos;          // position of vertex
    Vector2D    tex;          // texture coordinates
    Vector3D    n;            // unit normal
    Vector3D    t, b;         // tangent and binormal
    Vector3D    l;            // light vector in the tangent space
    Vector3D    h;
                              // map vector to tangent (TBN) space
    Vector3D    mapToTangentSpace ( const Vector3D& v ) const
    {
        return Vector3D ( v & t, v & b, v & n );
    }
};
struct    Face                // triangular face
{
    int    index [3];         // indices to Vertex array
};
class    Matrix4x4;
class    Torus
{
    int      numRings;
    int      numSides;
    int      numVertices;
    int      numFaces;
    Vertex * vertices;
    Face   * faces;
    unsigned vertexId;         // id of vertex VBO
    unsigned indexId;          // id of index VBO
public:
    Torus  ( float r1, float r2, int rings, int sides );
    ~Torus ()
    {
        delete [] vertices;
        delete [] faces;
```

```
        glDeleteBuffersARB ( 1, &vertexId );
        glDeleteBuffersARB ( 1, &indexId  );
    }
    void    setupBuffers     ();
    void    calcLightVectors ( const Vector3D& light );
    void    calcLightVectors ( const Vector3D& light,
                               const Vector3D& eye,
                               const Matrix4x4& m );
    void    draw             ();
};
#endif
```

Листинг 10.5. Реализация класса `Torus` при использовании двух вершинных массивов

```
#include    "libExt.h"
#include    "Torus-3.h"
#include    "Matrix4x4.h"
Torus :: Torus ( float r1, float r2, int rings, int sides )
{
    float   ringDelta = 2.0 * M_PI / rings;
    float   sideDelta = 2.0 * M_PI / sides;
    float   invRings  = 1.0f / (float) rings;
    float   invSides  = 1.0f / (float) sides;
    int     i, j;
    numRings = rings;
    numSides = sides;
    vertices = new Vertex [(numSides+1)*(numRings+1)];
    faces    = new Face   [numSides*numRings*2];
                                // create vertices
    int index = 0;
    for ( i = 0; i <= rings; i++ )
    {
        float theta    = i * ringDelta;
        float cosTheta = cos ( theta );
        float sinTheta = sin ( theta );
        for ( j = 0; j <= sides; j++ )
```

```
    {
        float phi    = j * sideDelta;
        float cosPhi = cos ( phi );
        float sinPhi = sin ( phi );
        float dist = r2 + r1 * cosPhi;
    // (x,y,z) below is the parametric equation for a torus
    // when theta and phi both vary from 0 to pi.
                // x = cos(theta) * (R + r * cos(phi))
                // y = -sin(theta) * (R + r * cos(phi))
                // z = r * sin(phi)
        vertices [index].pos.x = cosTheta * dist;
        vertices [index].pos.y = -sinTheta * dist;
        vertices [index].pos.z = r1 * sinPhi;
     // now setup texture coordinates
        vertices [index].tex.x = j * invSides;
        vertices [index].tex.y = i * invRings;
   // T = normalize([dx/dtheta,dy/dtheta,dz/dtheta])
        float   dxdtheta = -sinTheta * dist;
        float   dydtheta = -cosTheta * dist;
        float   d        = 1 / sqrt ( dxdtheta*dxdtheta +
                                      dydtheta*dydtheta );
        vertices [index].t.x = dxdtheta * d;
        vertices [index].t.y = dydtheta * d;
        vertices [index].t.z = 0;
    // compute unit normal
        vertices [index].n.x =  cosTheta * cosPhi;
        vertices [index].n.y = -sinTheta * cosPhi;
        vertices [index].n.z = sinPhi;
    // B = N cross T
        vertices [index].b = vertices [index].n ^ vertices [index].t;
        index++;
    }
}
numVertices = index;
                                        // Create faces
index = 0;
for ( i = 0; i < rings; i++ )
    for ( j = 0; j < sides; j++, index += 2 )
```

```
        {
            faces [index].index [2] = i*(sides+1) + j;
            faces [index].index [1] = (i+1)*(sides+1) + j;
            faces [index].index [0] = (i+1)*(sides+1) + j + 1;
            faces [index+1].index [2] = i*(sides+1) + j;
            faces [index+1].index [1] = (i+1)*(sides+1) + j + 1;
            faces [index+1].index [0] = i*(sides+1) + j + 1;
        }
    numFaces = index;
}
void    Torus :: setupBuffers ()
{
                                            // create vertex data VBO
    glGenBuffersARB ( 1, &vertexId );
    glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vertexId );
    glBufferDataARB ( GL_ARRAY_BUFFER_ARB,
                      numVertices * sizeof ( Vertex ), vertices,
                      GL_STREAM_DRAW_ARB );
                                            // create index data VBO
    glGenBuffersARB ( 1, &indexId );
    glBindBufferARB ( GL_ELEMENT_ARRAY_BUFFER_ARB, indexId );
    glBufferDataARB ( GL_ELEMENT_ARRAY_BUFFER_ARB,
                      numFaces * sizeof ( Face ), faces,
                      GL_STATIC_DRAW_ARB );
                                            // unbind buffers
    glBindBufferARB ( GL_ARRAY_BUFFER_ARB,         0 );
    glBindBufferARB ( GL_ELEMENT_ARRAY_BUFFER_ARB, 0 );
}
void    Torus :: calcLightVectors ( const Vector3D& light )
{
                                    // compute texture coordinates for
                                    // normalization cibe map
    for ( int i = 0; i < numVertices; i++ )
        vertices [i].l = vertices [i].mapToTangentSpace ( light -
                                                    vertices [i].pos );
}
void    Torus :: calcLightVectors ( const Vector3D& light, const
                                    Vector3D& eye, const Matrix4x4& m )
```

```
{
    Vector3D    lt = m.transformPoint ( light );
    Vector3D    et = m.transformPoint ( eye   );
                                    // compute texture coordinates for
                                    // normalization cube map
    for ( int i = 0; i < numVertices; i++ )
    {
        Vector3D    l = (lt - vertices [i].pos).normalize ();
        Vector3D    v = (et - vertices [i].pos).normalize ();
        vertices [i].l = vertices [i].mapToTangentSpace ( l );
        vertices [i].h = vertices [i].mapToTangentSpace ( l + v );
    }
}
void    Torus :: draw ()
{
    int vertexStride = sizeof ( Vertex );
    int texOffs      = ((int)&vertices [0].tex) - ((int)&vertices [0]);
    int lOffs        = ((int)&vertices [0].h) -   ((int)&vertices [0]);
    glBindBufferARB     ( GL_ARRAY_BUFFER_ARB, vertexId );
                                        // update buffer data
    glBufferSubDataARB  ( GL_ARRAY_BUFFER_ARB, 0,
                          numVertices * sizeof ( Vertex ), vertices );
                                        // prepare drawing
    glPushClientAttrib  ( GL_CLIENT_VERTEX_ARRAY_BIT );
    glEnableClientState ( GL_TEXTURE_COORD_ARRAY );
    glEnableClientState ( GL_VERTEX_ARRAY );
    glVertexPointer     ( 3, GL_FLOAT, vertexStride, (void *)0 );
    glClientActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnableClientState      ( GL_TEXTURE_COORD_ARRAY );
    glTexCoordPointer        ( 2, GL_FLOAT, vertexStride,
                               (void *)texOffs );
    glClientActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnableClientState      ( GL_TEXTURE_COORD_ARRAY );
    glTexCoordPointer        ( 3, GL_FLOAT, vertexStride,
                               (void *)lOffs );
    glClientActiveTextureARB ( GL_TEXTURE2_ARB );
    glEnableClientState      ( GL_TEXTURE_COORD_ARRAY );
    glTexCoordPointer        ( 2, GL_FLOAT, vertexStride,
                               (void *)texOffs );
```

```
    glBindBufferARB          ( GL_ARRAY_BUFFER_ARB, 0 );
    glEnableClientState      ( GL_INDEX_ARRAY  );
    glBindBufferARB          ( GL_ELEMENT_ARRAY_BUFFER_ARB, indexId );
    glIndexPointer           ( GL_UNSIGNED_INT, 0, 0 );
                                                  // 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 ();
}
```

В листинге 10.6 приводится программа на C++, использующая только что введенный класс для вывода тора.

Листинг 10.6. Использование введенного класса с двумя вершинными буферами

```
//
// Sample to using VBO's in OpenGL
//
#include    "libExt.h"
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Matrix4x4.h"
#include    "Torus-3.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
Vector3D    eye   ( 7, 5, 7 );          // camera position
Vector3D    light ( 5, 0, 4 );          // light position
unsigned    normCubeMap;                // normalization cubemap id
unsigned    bumpMap;                    // normal map
unsigned    decalMap;                   // decal (diffuse) texture
float       angle = 0;
Torus       torus ( 1, 3, 30, 30 );
Vector3D    rot ( 0, 0, 0 );
```

```
int         mouseOldX = 0;
int         mouseOldY = 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 ()
{
    float   bias         [4] = { 0.75, 0.75, 0.75, 0.75 };
    float   ambientColor [4] = { 0, 0, 0.5f, 1 };   // dark blue
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
                                                  // draw the light
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glDisable          ( GL_REGISTER_COMBINERS_NV );
    glTranslatef       ( light.x, light.y, light.z );
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glDisable          ( GL_TEXTURE_CUBE_MAP_ARB );
    glActiveTextureARB ( GL_TEXTURE2_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glColor4f          ( 1, 1, 1, 1 );
    glutSolidSphere    ( 0.1f, 15, 15 );
    glPopMatrix        ();
                                          // setup texture units
                                          // bind bump (normal) map to
                                          // texture unit 0
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, bumpMap );
```

```
                                    // bind normalization cube map
                                    // to texture unit 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glEnable           ( GL_TEXTURE_CUBE_MAP_ARB );
glBindTexture      ( GL_TEXTURE_CUBE_MAP_ARB, normCubeMap );
glActiveTextureARB ( GL_TEXTURE2_ARB );
glEnable           ( GL_TEXTURE_2D );
glBindTexture      ( GL_TEXTURE_2D, decalMap );
glMatrixMode       ( GL_TEXTURE );
glLoadIdentity     ();
glScalef           ( 1, 2, 1 );
glMatrixMode       ( GL_MODELVIEW );
                                    // setup register combiners
glEnable               ( GL_REGISTER_COMBINERS_NV );
glCombinerParameteriNV  ( GL_NUM_GENERAL_COMBINERS_NV, 2 );
glCombinerParameterfvNV ( GL_CONSTANT_COLOR0_NV, bias );
glCombinerParameterfvNV ( GL_CONSTANT_COLOR1_NV, ambientColor );
                                    // setup general combiner 0
                                    // configure A = expand (tex0)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
                    GL_TEXTURE0_ARB, GL_EXPAND_NORMAL_NV, GL_RGB );
                                    // configure B = expand (tex1)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV, GL_RGB );
                                    // configure C = 0
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_C_NV,
                    GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                                    // configure D = 0
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_D_NV,
                    GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                                    // setup output of (h,n)
glCombinerOutputNV ( GL_COMBINER0_NV, GL_RGB,
                     GL_SPARE0_NV,       // AB output
                     GL_DISCARD_NV,      // CD output
                     GL_DISCARD_NV,      // sum output
                     GL_NONE,            // no scale
                     GL_NONE,            // no bias
                     GL_TRUE,            // AB = A dot B
                     GL_FALSE, GL_FALSE );
```

```
                                        // setup general combiner 1
                                          // A = max ( (n,h), 0 )
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_A_NV,
                    GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                                          // B = 1
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_RGB );
                                          // C = -constant color0.rgb
                                         // g.e. -0.75
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_C_NV,
                    GL_CONSTANT_COLOR0_NV, GL_SIGNED_NEGATE_NV,
                    GL_RGB );
                                          // configure D = 1
glCombinerInputNV ( GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_D_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_RGB );
                                       // setup output of (h,n)
                                         //    - 0.75
glCombinerOutputNV ( GL_COMBINER1_NV, GL_RGB,
                     GL_DISCARD_NV,      // AB output
                     GL_DISCARD_NV,      // CD output
                     GL_SPARE0_NV,       // sum output
                     GL_SCALE_BY_FOUR_NV,// scale = 4
                     GL_NONE,            // no bias
                     GL_FALSE,           // we do not need dot
                                         // product here
                     GL_FALSE,
                     GL_FALSE );         // neither we need mux here
                                         // now spare0.rgb =
                                         //     4*((n,h) - 0.75)
                                         // configure final combiner
                                         // A.rgb=max (0, spare0.rgb)
glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                                          // B.rgb = max (0,
                                          // spare0.rgb)
glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_SPARE0_NV,
                         GL_UNSIGNED_IDENTITY_NV, GL_RGB );
```

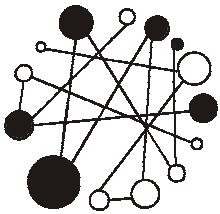
```
                                        // C = 0
    glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                                      // D = texture2
    glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_TEXTURE2_ARB,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                                       // E = 0
    glFinalCombinerInputNV ( GL_VARIABLE_E_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                                       // F = 0
    glFinalCombinerInputNV ( GL_VARIABLE_F_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                                     // G.alpa = 1
    glFinalCombinerInputNV ( GL_VARIABLE_G_NV, GL_ZERO,
                             GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                                      // so at output we'll get
                                    // RGB   = max ( 0,
                                      // max ((h, n),0) - 0.75 ) ^ 2
                                      // alpha = 1
                                     // now setup rotation
    glMatrixMode   ( GL_MODELVIEW );
    glPushMatrix   ();
    glRotatef    ( rot.x, 1, 0, 0 );
    glRotatef    ( rot.y, 0, 1, 0 );
    glRotatef    ( rot.z, 0, 0, 1 );
                                      // transform light and eye
                                     // positions with inverse
                                  // rotation matrix
    Matrix4x4   rx = Matrix4x4 :: rotateX ( -rot.x * M_PI / 180 );
    Matrix4x4   ry = Matrix4x4 :: rotateY ( -rot.y * M_PI / 180  );
    Matrix4x4   rz = Matrix4x4 :: rotateZ ( -rot.z * M_PI / 180  );
    Matrix4x4   r  = rx * ry * rz;
    r.invert ();
    torus.calcLightVectors ( light, eye, r );
    torus.draw             ();
    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 );
    light.x = 2*cos ( angle );
    light.y = 2*sin ( angle );
    light.z = 3 + 0.3 * sin ( angle / 3 );
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                            // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                            // create window
    int win = glutCreateWindow ( "OpenGL per-pixel specular lit torus
with register combiners" );
                            // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    glutMouseFunc    ( mouse   );
    glutMotionFunc   ( motion  );
    init           ();
    initExtensions ();
    printfInfo     ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_NV_register_combiners" ) )
    {
        printf ( "NV_register_combiners NOT supported" );
```

```
        return 2;
    }
    if ( !isExtensionSupported ( "GL_ARB_vertex_buffer_object" ) )
    {
        printf ( "ARB_vertex_buffer_object" );
        return 3;
    }
    bumpMap     = createNormalMapFromHeightMap ( false,
                          "../Textures/Bumpmaps/light06.tga", 1 );
    decalMap    = createTexture2D              ( true,
                          "../Textures/block.bmp" );
    normCubeMap = createNormalizationCubemap   ( 32 );
    torus.setupBuffers ();                  // create VBO's
    glutMainLoop ();
    return 0;
}
```

Глава 11



P-буфер и рендеринг в текстуру. Сопутствующие расширения

В ряде случаев возникает необходимость произвести отдельный рендеринг либо всей сцены, либо какой-то ее части для дальнейшего использования результатов (как правило, в качестве текстуры).

Простейшим примером ситуации, когда это может понадобиться, является рендеринг поверхности воды с рябью. Если мы при этом хотим, чтобы поверхность действительно отражала текущую сцену, то применение EMBM с заранее рассчитанной кубической картой отражений уже не подходит.

В то же время, если отражение в плоской воде (без ряби) можно было бы занести в текстуру, то тогда имитацию ряби можно реализовать с помощью шейдера `NV_OFFSET_TEXTURE_2D` (*см. главу 12*) или выводить поверхность воды как сетку, вершины которой подвергаются искажениям.

Существует несколько приемов, позволяющих получить результаты рендеринга в виде текстуры. Простейший — функцией `glReadPixels`, которая позволяет прочитать изображение из фреймбуфера (памяти графического ускорителя) непосредственно в память CPU. Затем это изображение можно будет загрузить в текстуру (например, при помощи `glTexImage2D`). Однако такой подход требует копирования сначала из видеопамяти в память центрального процессора, а затем — обратно.

Подобные операции копирования являются довольно дорогостоящими, поэтому этот путь малопригоден из-за больших временных затрат на копирование данных между графическим ускорителем и CPU.

Гораздо более удобно 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.

Несмотря на то, что этот подход гораздо более эффективен, чем предыдущий, он все равно требует явного копирования памяти, а также загружает тот же самый фреймбуфер, что и для построения главного изображения.

В ряде случаев для осуществления рендеринга было бы гораздо удобнее создать себе "виртуальный" (или пиксельный) буфер, обладающий собственным размером и другими атрибутами и никак не связанный с основным фрейм-буфером.

Еще лучше, если бы этот буфер можно было непосредственно использовать в качестве данных для текстуры, вообще избегая тем самым копирования памяти.

Все эти возможности и реализуются с помощью так называемых *p*-буферов (*p-buffer*). При этом на современных графических ускорителях *p*-буферы хранятся непосредственно в видеопамяти и процесс рендеринга в них аппаратно ускорен.

К сожалению, работа с *p*-буфером под Windows и Linux заметно различается. Поэтому мы сначала рассмотрим каждую из них отдельно, а затем приведем пример кроссплатформенного использования *p*-буфера.

Работа с p-буфером под Microsoft Windows

Для создания и работы с *p*-буферами под Microsoft Windows необходима поддержка сразу двух расширений: WGL_ARB_pixel_format и WGL_ARB_pbuffer. При задании *p*-буфера в качестве источника данных для текстуры необходима также 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 доступный контекст текущего устройства командой

```
HDC hDc = wglGetCurrentDC ();
```

Далее необходимо установить подходящий формат пикселов для создания *p*-буфера.

Для этого служит 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 (табл. 11.1). Каждый такой атрибут задается парой значений (`attr`, `value`), где сначала идет идентификатор атрибута, а затем — его значение. Нулевой идентификатор атрибута обозначает конец списка.

Параметр `maxFormats` сообщает, какое количество целочисленных идентификаторов формата можно записать в массив `formats`.

Последний параметр, `numFormats`, является указателем на целочисленную переменную, в которой будет возвращено количество подходящих форматов, это значение может быть меньше, чем `maxFormats`.

Таблица 11.1. *Основные типы атрибутов для функции* `wglChoosePixelFormatARB`

Атрибут	Комментарий
`WGL_COLOR_BITS_ARB`	Минимальное общее число битов под цвет
`WGL_STENCIL_BITS_ARB`	Минимальное число битов под значение в буфере трафарета
`WGL_AUX_BUFFERS_ARB`	Минимальное число дополнительных буферов
`WGL_SUPPORT_OPENGL_ARB`	Должен ли формат поддерживать вывод средствами OpenGL
`WGL_DRAW_TO_PBUFFER_ARB`	Ненулевое значение говорит о том, что вывод будет направляться в *p*-буфер
`WGL_ACCUM_BITS_ARB`	Общее число битов под значение в буфере накопления
`WGL_DOUBLE_BUFFER_ARB`	Ненулевое значение говорит о необходимости создания второго буфера
`WGL_ALPHA_BITS_ARB`	Минимальное число битов под значение в альфа-канале
`WGL_DEPTH_BITS_ARB`	То же в буфере глубины
`WGL_PIXEL_TYPE_ARB`	Тип значений пикселов; принимает одно из двух значений: `WGL_TYPE_RGBA_ARB` или `WGL_TYPE_COLORINDEX_ARB`
`WGL_RED_BITS_ARB`	Минимальное число битов под значение в красном канале
`WGL_GREEN_BITS_ARB`	То же в зеленом канале
`WGL_BLUE_BITS_ARB`	То же в синем канале

Данная функция возвращает нулевое значение в случае ошибки и ненулевое, если вызов прошел успешно. В случае если получен хотя бы один иден-

тификатор формата, то он пригоден в дальнейшем для создания *p*-буфера. Для этого служит функция

```
HPBUFFER wglCreatePbufferARB ( HDC hDc, int format, int width,
                               int height, const int * attrs );
```

Назначение параметров следующее:

- `hDc` — контекст устройства, полученный на шаге 1;.
- `format` — идентификатор формата пикселов для создаваемого *p*-буфера;
- `width` и `height` задают желаемый размер *p*-буфера в пикселах;
- в `attrs` передаются дополнительные целочисленные параметры, аналогично функции `wglChoosePixelFormatARB`.

Данная функция возвращает значение `handle` созданного *p*-буфера или `NULL` в случае ошибки.

Размер созданного *p*-буфера может отличаться от запрашиваемого, поэтому после создания 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 необходимо получить контекст устройства для него при помощи функции

```
HDC wglGetPbufferDCARB ( HPBUFFER hBuffer );
```

Заключительный шаг — создание для *p*-буфера контекста рендеринга. Это можно сделать при помощи функции `wglCreateContext`.

```
HGLRC wglCreateContext ( HDC hBufDc );
```

Выбор *p*-буфера как текущей цели для рендеринга

Для того чтобы направить вывод в заданный *p*-буфер, служит команда

```
BOOL wglMakeCurrent ( HDC hBufDc, HGLRC hBufRc );
```

Параметры `hBufDc` и `hBufRc` — это контекст устройства и рендеринга для данного *p*-буфера.

Для того чтобы снова направить вывод в окно, следует вызвать функцию `wglMakeCurrent` с параметрами, соответствующими окну, в которое надо

перенаправить вывод. В библиотеке GLUT для этой есть функция `glutSetWindow`, принимающая на вход целочисленный идентификатор окна.

Уничтожение *p*-буфера

Для уничтожения *p*-буфера (и освобождения ресурсов, связанных с ним) необходимо выполнить следующие три шага:

1. Уничтожить его контекст рендеринга.
2. Освободить его контекст устройства.
3. Уничтожить сам 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, что делает его непригодным для дальнейшего использования.

Определить, является ли данный *p*-буфер по-прежнему доступным или нет, можно с помощью следующего фрагмента кода:

```
int  flag;
wglQueryPbufferARB ( hBuffer, WGL_PBUFFER_LOST_ARB, &flag );
```

Если после его выполнения значение переменной `flag` окажется отличным от нуля, значит, *p*-буфер был поврежден и не является больше доступным. 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 функцией `glCopyTexSubImage2D` скопировать его содержимое (или только часть) в текстуру.

Однако обратите внимание, что в платформе Windows различные контексты рендеринга имеют свои наборы атрибутов, в том числе и свои наборы дисплейных списков, текстур и т. п. Поэтому, если вы хотите осуществить рендеринг текстуры в *p*-буфер, 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  hGlRcFrom,  HGLRC  hGlRcTo );
```

Здесь параметры `hGlRcFrom` и `hGlRcTo` задают исходный контекст (из которого надо перенести дисплейные списки и текстуры) и тот, в который надо осуществить перенос.

Связывание *p*-буфера с текстурой

В случае поддержки расширения WGL_ARB_render_texture можно непосредственно выбрать *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 хранит содержимое цветового буфера данного *p*-буфера, а сам он недоступен для рендеринга. Когда в *p*-буфер необходимо произвести вывод, то он сначала должен быть "освобожден" от данной текстуры.

Можно также связать *p*-буфер с кубической картой, тогда при рендеринге ему сообщается, в какую грань куба идет вывод.

Можно также применить пирамидальное фильтрование, однако для этого надо либо явно задавать все промежуточные уровни, либо воспользоваться рассмотренным ранее расширением GL_SGIS_generate_mipmap, которое обеспечивает автоматическое построение всех промежуточных уровней для заданной текстуры.

Если *p*-буфер привязан к текстуре и в это время из-за переключения видеорежима он будет поврежден, то несмотря на это содержимое текстуры попрежнему остается пригодным для использования и не разрушается.

При освобождении *p*-буфера от текстуры содержимое его цветового буфера может стать неопределенным.

Для привязки *p*-буфера к текстуре необходимо задание дополнительных атрибутов в функциях `wglChoosePixelFormatARB` и `wglCreatePbufferARB`.

Для функции `wglChoosePixelFormatARB` этими атрибутами являются `WGL_BIND_TO_TEXTURE_RGB_ARB` или `WGL_BIND_TO_TEXTURE_RGBA_ARB`. Ненулевое их значение свидетельствует о том, что данный *p*-буфер должен поддерживать привязку в RGB- (или RGBA-) текстуре.

Для функции `wglCreatePbufferARB` вводятся следующие дополнительные атрибуты: `WGL_TEXTURE_TARGET_ARB`, принимающий одно из трех значений (`WGL_TEXTURE_1D_ARB`, `WGL_TEXTURE_2D_ARB` или `WGL_TEXTURE_CUBE_MAP_ARB`), и `WGL_MIPMAP_TEXTURE_ARB`, ненулевое значение которого обеспечивает вы-

деление памяти под все промежуточные уровни пирамидального фильтрования.

Для того чтобы привязать *p*-буфер к текущей текстуре, служит функция

```
BOOL wglBindTexImageARB ( HPBUFFER hBuffer, int buf );
```

Параметр `buf` задает, какой именно из буферов данного *p*-буфера следует связать с текущей текстурой. Он принимает одно из следующих значений: `WGL_FRONT_LEFT_ARB`, `WGL_FRON_RIGHT_ARB`, `WGL_BACK_LEFT_ARB`, `WGL_BACK_RIGHT_ARB`, `WGL_AUX0_ARB`, `WGL_AUX1_ARB`, ...

После выполнения этой команды данный *p*-буфер привязывается к текущей текстуре и в него нельзя производить рендеринг.

Для того чтобы "отвязать" *p*-буфер от текстуры, служит функция

```
BOOL wglReleaseTexImageARB ( HPBUFFER hBuffer, int buf );
```

При привязывании *p*-буфера к кубической карте необходимо функцией `wglSetPbufferAttribARB` устанавливать, в какую именно грань куба идет вывод. В качестве атрибута выступает `WGL_CUBE_MAP_FACE_ARB`, который 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 выполнить команду `glFlush ()`.

Реализация *p*-буфера для платформы Windows

Для удобства работы *p*-буфер можно завернуть в класс, инкапсулирующий все операции с ним, включая его создание и уничтожение.

Листинг 11.1 содержит описание такого класса.

Листинг 11.1. Описание класса `PBuffer` для платформы Windows

```
//
// Simple p-buffer encapsulation for win32
//
```

```
#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 из значений, приведенных в табл. 11.2.

Таблица 11.2. *Флаги для класса* `PBuffer`

Константа	Комментарий
`modeAlpha`	Необходим альфа-канал
`modeDepth`	Необходим буфер глубины
`modeStencil`	Необходим буфер трафарета
`modeAccum`	Необходим буфер накопления
`modeDouble`	Необходим двойной буфер
`modeTexture1D`	Будет привязываться к одномерной текстуре
`modeTexture2D`	То же к двумерной текстуре
`modeTextureCubeMap`	То же к кубической карте
`modeTextureMipmap`	Использовать пирамидальное фильтрование

Назначение методов следующее.

- `create` и `clear` создают и уничтожают *p*-буфер. При этом для его создания используются параметры, переданные в конструктор.
- `makeCurrent` делает данный *p*-буфер текущей целью вывода. При этом запоминается, какая цель была текущей на момент этого вызова, и метод `restoreCurrent` восстанавливает цель вывода, которая была на момент вызова `makeCurrent`.
- `isLost` возвращает значение `true`, если буфер был разрушен. В этом случае для его восстановления нужно вызвать сначала `clear`, а затем `create`.
- `bindToTexture` и `unbindFromTexture` служат для привязывания и освобождения данного *p*-буфера от текущей текстуры.
- `setCubemapSide` задает, в какую именно грань кубической карты идет рендеринг. Он автоматически вызывает `glFlush`.

- `printLastError` печатает на `stderr` расшифровку последней возникшей ошибки, что может быть полезно при отладке приложения.
- `checkExtensions` проверяет поддержку необходимых для работы с *p*-буфером расширений.

При реализации данного класса удобно создать параметризованный класс `AttrList` для инкапсуляции работы с атрибутами различных типов (листинг 11.2).

Листинг 11.2. Класс `AttrList`

```
//
// AttrList class definition
//
#ifndef    __ATTR_LIST__
#define    __ATTR_LIST__
#define    MAX_ATTRIBS    20
#define    MAX_PFORMATS   20
template <typename T>
class    AttrList
{
    T      attrs [2*MAX_ATTRIBS];
    int    numAttrs;
public:
    AttrList ()
    {
        numAttrs = 0;
        clear ();
    }
    void    clear ()
    {
        for ( int i = 0; i < 2*MAX_ATTRIBS; i++ )
            attrs [i] = 0;
    }
    void    add ( int attr, T value )
    {
        attrs [2*numAttrs    ] = attr;
        attrs [2*numAttrs + 1] = value;
        numAttrs++;
    }
```

```
    int    getNumAttrs () const
    {
        return numAttrs;
    }
    const T * getAttrs () const
    {
        return attrs;
    }
};
#endif
```

В листинге 11.3 приводится полная реализация класса `PBuffer` для Microsoft Windows.

Листинг 11.3. Реализация класса `PBuffer`

```
//
// Simple p-buffer encapsulation for win32
//
#include    <stdio.h>
#include    "libExt.h"
#include    "PBuffer.h"
#include    "..\AttrList.h"
PBuffer :: PBuffer  ( int theWidth, int theHeight, int theMode,
                      bool theShareObjects )
{
    width        = theWidth;
    height       = theHeight;
    mode         = theMode;
    shareObjects = theShareObjects;
    hBuffer      = NULL;
    hdc          = NULL;
    hGlRc        = NULL;
    texture      = 0;
    saveHdc      = NULL;
    saveHglrc    = NULL;
}
PBuffer :: ~PBuffer ()
```

```
{
    clear ();
}
void    PBuffer :: clear ()
{
    if ( hBuffer != NULL )
    {
        wglDeleteContext       ( hGlRc );
        wglReleasePbufferDCARB ( hBuffer, hdc );
        wglDestroyPbufferARB   ( hBuffer );
    }
    glDeleteTextures ( 1, &texture );
}
bool    PBuffer :: create ()
{
    if ( !checkExtensions () )
        return false;
    HDC      hCurDc = wglGetCurrentDC      ();
    HGLRC    hCurRc = wglGetCurrentContext ();
    int      format;
                        // check for extension support
                        // & initialization
    if ( wglChoosePixelFormatARB == NULL ||
         wglCreatePbufferARB     == NULL ||
         wglGetPbufferDCARB      == NULL ||
         wglQueryPbufferARB      == NULL ||
         wglReleasePbufferDCARB  == NULL ||
         wglDestroyPbufferARB    == NULL )
             return false;
    if ( (mode & modeTexture1D) || (mode & modeTexture2D) ||
              (mode & modeTextureCubeMap) )
        if ( wglBindTexImageARB     == NULL ||
                    wglReleaseTexImageARB  == NULL ||
                    wglSetPbufferAttribARB == NULL )
            return false;
                        // query for pixel format
    AttrList <int>      intAttrs;
    AttrList <float>    floatAttrs;
```

```
intAttrs.add ( WGL_SUPPORT_OPENGL_ARB,  GL_TRUE );
intAttrs.add ( WGL_DRAW_TO_PBUFFER_ARB, GL_TRUE );
intAttrs.add ( WGL_PIXEL_TYPE_ARB,      WGL_TYPE_RGBA_ARB );
intAttrs.add ( WGL_RED_BITS_ARB,        8 );
intAttrs.add ( WGL_GREEN_BITS_ARB,      8 );
intAttrs.add ( WGL_BLUE_BITS_ARB,       8 );
intAttrs.add ( WGL_DOUBLE_BUFFER_ARB, (mode & modeDouble ?
                                          GL_TRUE : GL_FALSE ) );
if ( mode & modeAlpha )
    intAttrs.add ( WGL_ALPHA_BITS_ARB, 8 );

if ( mode & modeDepth )
    intAttrs.add ( WGL_DEPTH_BITS_ARB, 24 );
if ( mode & modeStencil )
    intAttrs.add ( WGL_STENCIL_BITS_ARB, 8 );
if ( mode & modeAccum )
    intAttrs.add ( WGL_ACCUM_BITS_ARB, 32 );
if ( (mode & modeTexture1D) || (mode & modeTexture2D) ||
        (mode & modeTextureCubeMap) )
    if ( mode & modeAlpha )
        intAttrs.add ( WGL_BIND_TO_TEXTURE_RGBA_ARB,
                       GL_TRUE );
    else
        intAttrs.add ( WGL_BIND_TO_TEXTURE_RGB_ARB,
                       GL_TRUE );
int        pixelFormats [MAX_PFORMATS];
unsigned   numFormats = 0;
if ( !wglChoosePixelFormatARB ( hCurDc, intAttrs.getAttrs (),
                                floatAttrs.getAttrs (),
                                MAX_PFORMATS, pixelFormats,
                                &numFormats ) )
    return false;
if ( numFormats < 1 )
    return false;
format = pixelFormats [0];
AttrList <int>    props;
if ( (mode & modeTexture1D) || (mode & modeTexture2D) )
    if ( mode & modeAlpha )
```

```
        props.add ( WGL_TEXTURE_FORMAT_ARB,
                    WGL_TEXTURE_RGBA_ARB );
    else
        props.add ( WGL_TEXTURE_FORMAT_ARB,
                    WGL_TEXTURE_RGB_ARB );
if ( mode & modeTexture1D )
    props.add ( WGL_TEXTURE_TARGET_ARB, WGL_TEXTURE_1D_ARB );
else
if ( mode & modeTexture2D )
    props.add ( WGL_TEXTURE_TARGET_ARB, WGL_TEXTURE_2D_ARB );
else
if ( mode & modeTextureCubeMap )
    props.add ( WGL_TEXTURE_TARGET_ARB,
                WGL_TEXTURE_CUBE_MAP_ARB );
if ( mode & modeTextureMipmap )
    props.add ( WGL_MIPMAP_TEXTURE_ARB, GL_TRUE );
hBuffer = wglCreatePbufferARB ( hCurDc, format, width, height,
                                props.getAttrs () );
if ( hBuffer == NULL )
    return false;
hdc = wglGetPbufferDCARB ( hBuffer );
if ( hdc == NULL )
    return false;
hGlRc = wglCreateContext ( hdc );
if ( shareObjects )
    wglShareLists ( hCurRc, hGlRc );
                    // get real size
wglQueryPbufferARB ( hBuffer, WGL_PBUFFER_WIDTH_ARB,  &width );
wglQueryPbufferARB ( hBuffer, WGL_PBUFFER_HEIGHT_ARB, &height );
                    // now create associated
                    // texture
if ( (mode & modeTexture2D) == 0 )
    return true;
glGenTextures ( 1, &texture );
glBindTexture ( GL_TEXTURE_2D, texture );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                    GL_CLAMP_TO_EDGE);
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                    GL_CLAMP_TO_EDGE);
```

```
    glPixelStorei   ( GL_UNPACK_ALIGNMENT, 1 );
    glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                      GL_LINEAR );
    glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                      GL_LINEAR  );
    if ( mode & modeTextureMipmap )
    {
            glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                              GL_LINEAR_MIPMAP_LINEAR );
            glTexParameteri ( GL_TEXTURE_2D, GL_GENERATE_MIPMAP_SGIS,
                              GL_TRUE );
    }
    glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0,
                   GL_RGBA, GL_UNSIGNED_BYTE, 0 );
    return true;
}
bool    PBuffer :: makeCurrent ()
{
    saveHdc   = wglGetCurrentDC      ();    // save previous target
    saveHglrc = wglGetCurrentContext ();
    return wglMakeCurrent ( hdc, hGlRc ) != 0;
}
bool    PBuffer :: restoreCurrent ()
{
    if ( saveHdc == NULL || saveHglrc == NULL )
        return false;
    bool    result = wglMakeCurrent ( saveHdc, saveHglrc );
    saveHdc   = NULL;
    saveHglrc = NULL;
    return result;
}
bool    PBuffer :: isLost () const
{
    int    lost = 0;
    wglQueryPbufferARB ( hBuffer, WGL_PBUFFER_LOST_ARB, &lost );
    return lost != 0;
}
bool    PBuffer :: bindToTexture ( int buf )
```

```
{
    glBindTexture ( GL_TEXTURE_2D, texture );
    return wglBindTexImageARB ( hBuffer, buf ) != GL_FALSE;
}
bool    PBuffer :: unbindFromTexture ( int buf )
{
    glBindTexture ( GL_TEXTURE_2D, texture );
    bool    res = wglReleaseTexImageARB ( hBuffer, buf ) != GL_FALSE;
    glBindTexture ( GL_TEXTURE_2D, 0 );
    return res;
}
bool    PBuffer :: setAttr ( int attr, int value )
{
    AttrList <int>    attrs;
    attrs.add ( attr, value );
    return wglSetPbufferAttribARB ( hBuffer, attrs.getAttrs () ) !=
                                                        GL_FALSE;
}
bool    PBuffer :: setCubemapSide ( int side )
{
    if ( (mode & modeTextureCubeMap) == 0 )
        return false;
    glFlush ();
    return setAttr ( WGL_CUBE_MAP_FACE_ARB, side );
}
void PBuffer :: printLastError()
{
    DWORD err = GetLastError();
    switch ( err )
    {
        case ERROR_INVALID_PIXEL_FORMAT:
            fprintf ( stderr, "Error:  ERROR_INVALID_PIXEL_FORMAT\n" );
            break;
        case ERROR_NO_SYSTEM_RESOURCES:
            fprintf ( stderr, "Error:  ERROR_NO_SYSTEM_RESOURCES\n ");
            break;
        case ERROR_INVALID_DATA:
            fprintf ( stderr, "Error:  ERROR_INVALID_DATA\n" );
            break;
```

```
        case ERROR_INVALID_WINDOW_HANDLE:
            fprintf ( stderr, "Error:  ERROR_INVALID_WINDOW_HANDLE\n" );
            break;
        case ERROR_RESOURCE_TYPE_NOT_FOUND:
            fprintf ( stderr, "Error:  ERROR_RESOURCE_TYPE_NOT_FOUND\n"
);
            break;
        case ERROR_SUCCESS:
                          // no error
            break;
    default:
            LPVOID msgBuf;
            FormatMessage ( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                            FORMAT_MESSAGE_FROM_SYSTEM     |
                            FORMAT_MESSAGE_IGNORE_INSERTS,
                    NULL, err,
                      MAKELANGID ( LANG_NEUTRAL, SUBLANG_DEFAULT ),
                     (LPTSTR) &msgBuf, 0, NULL );
           fprintf  ( stderr, "Error %d: %s\n", err, msgBuf );
            LocalFree( msgBuf );
           break;
    }
    SetLastError ( 0 );
}
bool    PBuffer :: checkExtensions ()
{
    return isExtensionSupported ( "WGL_ARB_pbuffer"         ) &&
           isExtensionSupported ( "WGL_ARB_pixel_format"    ) &&
           isExtensionSupported ( "WGL_ARB_render_texture"  ) &&
           isExtensionSupported ( "GL_SGIS_generate_mipmap" );
}
```

Работа с *p*-буфером под Linux

Для работы с *p*-буфером под платформой Linux необходима поддержка следующих 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 двух частей: указа-

теля на объект `Display` и 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:

```
GLXFBConfig * glxChooseFBConfig ( Display * display, int screen,
                                  const int * attrList,
                                  int * numConfigs );
```

Параметры `display` и `screen` задают, для чего будет выбираться подходящий формат. Параметр `attrList` является указателем на массив пар (*атрибут*, *значение*), завершенный как и для Windows нулевым значением.

В последнем параметре возвращается количество найденных форматов. В табл. 11.3 приведены основные атрибуты этой функции.

Таблица 11.3. *Основные атрибуты для функции* `glXChooseFBConfig`

Атрибут	**Комментарий**
`GLX_DRAWABLE_TYPE_SGIX`	Определяет, куда будет производиться вывод. Используется значение `GLX_PBUFFER_BIT_SGIX`
`GLX_RENDER_TYPE_SGIX`	Задает тип поверхности для вывода. Используется значение `GLX_RGBA_BIT_SGIX`
`GLX_DOUBLE_BUFFER`	Ненулевое значение обозначает использование двойной буферизации
`GLX_RED_SIZE`	Минимальное число битов под значение в красном канале
`GLX_GREEN_SIZE`	То же в зеленом канале
`GLX_BLUE_SIZE`	То же в синем канале
`GLX_ALPHA_SIZE`	То же в альфа-канале
`GLX_DEPTH_SIZE`	То же в буфере глубины
`GLX_STENCIL_SIZE`	То же в буфере трафарета
`GLX_ACCUM_RED_SIZE`	То же в красном канале буфера накопления
`GLX_ACCUM_GREEN_SIZE`	То же в зеленом канале буфера накопления
`GLX_ACCUM_BLUE_SIZE`	То же в синем канале буфера накопления
`GLX_ACCUM_ALPHA_SIZE`	То же в альфа-канале буфера накопления

После получения подходящего формата фреймбуфера можно непосредственно создать сам *p*-буфер при помощи команды

```
GLXPbuffer glXCreateGLXPbufferSGIX ( Display * display,
                                     GLXFBConfig config,
                                     unsigned int width,
                                     unsigned int height,
                                     int * attrList );
```

Параметр `display` задает объект, для которого будет создан *p*-буфер, `config` — его пискельный формат, `width` и `height` — его желаемый размер, `attrList` позволяет передать дополнительные параметры.

В отличие от Windows созданный таким образом *p*-буфер по умолчанию не разрушается при переключении режима экрана.

Получить реальные размеры *p*-буфера можно с помощью следующего фрагмента кода:

```
glXQueryGLXPbufferSGIX ( display, hBuffer, GLX_WIDTH_SGIX,
                         (unsigned *)&width  );
glXQueryGLXPbufferSGIX ( display, hBuffer, GLX_HEIGHT_SGIX,
                         (unsigned *)&height );
```

К сожалению, под Linux нет аналога расширения WGL_render_texture, поэтому необходимо явное копирование изображения из *p*-буфера в текстуру функцией `glCopyTexSubImage2D`.

Как и в случае реализации для Windows, мы завернем *p*-буфер в класс с аналогичным открытым (*public*) интерфейсом, что позволяет легко работать с *p*-буфером в кроссплатформенных приложениях. Листинг 11.4 содержит описание данного класса.

Листинг 11.4. Описание класса `PBuffer` для платформы Linux

```
//
// Simple p-buffer encapsulation for 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
```

В листинге 11.5 приводится реализация данного класса.

Листинг 11.5. Реализация класса `PBuffer` для платформы Linux

```
//
// Simple p-buffer encapsulation for Linux.
//
```

```
#include    <stdio.h>
#include    "libExt.h"
#include    "PBuffer.h"
#include    "../AttrList.h"
PBuffer :: PBuffer  ( int theWidth, int theHeight, int theMode,
                      bool theShareObjects )
{
    width             = theWidth;
    height            = theHeight;
    mode              = theMode;
    display           = 0;
    shareObjects      = theShareObjects;
    hBuffer           = 0;
    hGLContext        = NULL;
    hPreviousContext  = NULL;
    hPreviousDrawable = 0;
    texture           = 0;
}
PBuffer :: ~PBuffer ()
{
    clear ();
}
void    PBuffer :: clear ()
{
    if ( hBuffer != 0 )
    {
        if ( glXGetCurrentContext () == hGLContext )
            glXMakeCurrent ( display, hBuffer, 0 );
        glXDestroyContext ( display, hGLContext );
        glXDestroyPbuffer ( display, hBuffer );
        hBuffer = 0;
    }
    glDeleteTextures ( 1, &texture );
}
bool    PBuffer :: create ()
{
    if ( !checkExtensions () )
        return false;
```

```
display = glXGetCurrentDisplay ();
    GLXContext  context = glXGetCurrentContext ();
int         screen  = DefaultScreen ( display );
    AttrList <int>        intAttrs;
    intAttrs.add ( GLX_RENDER_TYPE_SGIX,   GLX_RGBA_BIT_SGIX );
intAttrs.add ( GLX_DRAWABLE_TYPE_SGIX, GLX_PBUFFER_BIT_SGIX );
intAttrs.add ( GLX_RED_SIZE,           8 );
intAttrs.add ( GLX_GREEN_SIZE,         8 );
intAttrs.add ( GLX_BLUE_SIZE,          8 );
if ( mode & modeAlpha )
    intAttrs.add ( GLX_ALPHA_SIZE, 8 );
else
    intAttrs.add ( GLX_ALPHA_SIZE, 0 );
if ( mode & modeStencil )
    intAttrs.add ( GLX_STENCIL_SIZE, 8 );
else
    intAttrs.add ( GLX_STENCIL_SIZE, 0 );
if ( mode & modeDepth )
    intAttrs.add ( GLX_DEPTH_SIZE, 24 );
if ( mode & modeAccum )
{
    intAttrs.add ( GLX_ACCUM_RED_SIZE,        8 );
    intAttrs.add ( GLX_ACCUM_GREEN_SIZE,      8 );
    intAttrs.add ( GLX_ACCUM_BLUE_SIZE,       8 );
    if ( mode & modeAlpha )
        intAttrs.add ( GLX_ACCUM_ALPHA_SIZE, 8 );
}
if ( mode & modeDouble )
    intAttrs.add ( GLX_DOUBLEBUFFER, GL_TRUE );
else
    intAttrs.add ( GLX_DOUBLEBUFFER, GL_FALSE );
GLXFBConfig * fbConfigs;
int           count;
fbConfigs = glXChooseFBConfig ( display, screen,
                                intAttrs.getAttrs (), &count );
    if ( fbConfigs == NULL || count < 1 )
    return false;
                    // pick first format for
                    // which we can create pbuffer
```

```
for ( int i = 0; i < count; i++ )
{
    hBuffer = glXCreateGLXPbufferSGIX ( display, fbConfigs [i],
                                        width, height, NULL );
    if ( hBuffer != 0 )
    {
        hGLContext = glXCreateContextWithConfigSGIX (
                            display, fbConfigs [i], GLX_RGBA_TYPE,
                            shareObjects ? context : NULL, GL_TRUE );
        break;
    }
}
if ( hBuffer == 0 )
    return false;
if ( hGLContext == NULL )
    return false;
glXQueryGLXPbufferSGIX ( display, hBuffer, GLX_WIDTH_SGIX,
                         (unsigned *)&width  );
glXQueryGLXPbufferSGIX ( display, hBuffer, GLX_HEIGHT_SGIX,
                         (unsigned *)&height );
if ( (mode & modeTexture2D) == 0 )    // no texture creation required
    return true;
                    // create associated texture
glGenTextures   ( 1, &texture );
glBindTexture   ( GL_TEXTURE_2D, texture );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                  GL_CLAMP_TO_EDGE );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                  GL_CLAMP_TO_EDGE );
glPixelStorei   ( GL_UNPACK_ALIGNMENT, 1 );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                  GL_LINEAR );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR );
    if ( mode & modeTextureMipmap )
{
        glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                          GL_LINEAR_MIPMAP_LINEAR );
```

```
            glTexParameteri ( GL_TEXTURE_2D, GL_GENERATE_MIPMAP_SGIS,
                              GL_TRUE );
    }
    glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0,
                   GL_RGBA, GL_UNSIGNED_BYTE, 0 );
    return true;
}
bool    PBuffer :: makeCurrent ()
{
    hPreviousContext  = glXGetCurrentContext  ();
    hPreviousDrawable = glXGetCurrentDrawable ();
    bool res = glXMakeCurrent ( display, hBuffer, hGLContext );
    glDrawBuffer ( GL_FRONT );
    glReadBuffer ( GL_FRONT );
    return res;
}
bool    PBuffer :: restoreCurrent ()
{
    if ( hPreviousContext == NULL || hPreviousDrawable == 0 )
        return false;
    glBindTexture       ( GL_TEXTURE_2D, texture );
    glCopyTexSubImage2D ( GL_TEXTURE_2D, 0, 0, 0, 0, 0, width,
                          height );
    bool    result = glXMakeCurrent ( display, hPreviousDrawable,
                                      hPreviousContext );
    hPreviousContext  = NULL;
    hPreviousDrawable = 0;
    return result;
}
bool    PBuffer :: isLost () const
{
    return false;
}
                        // bind pbuffer to previously
                        // bound texture
bool    PBuffer :: bindToTexture ()
{
    return true;
}
```

```
                        // unbind from texture
                        // (release)
bool    PBuffer :: unbindFromTexture ()
{
    return true;
}
bool    PBuffer :: checkExtensions ()
{
    return isExtensionSupported ( "GLX_SGIX_pbuffer"  ) &&
           isExtensionSupported ( "GLX_SGIX_fbconfig" ) &&
           isExtensionSupported ( "GL_SGIS_generate_mipmap" );
}
```

Примеры использования

Рассмотрим простейший пример использования *p*-буфера.

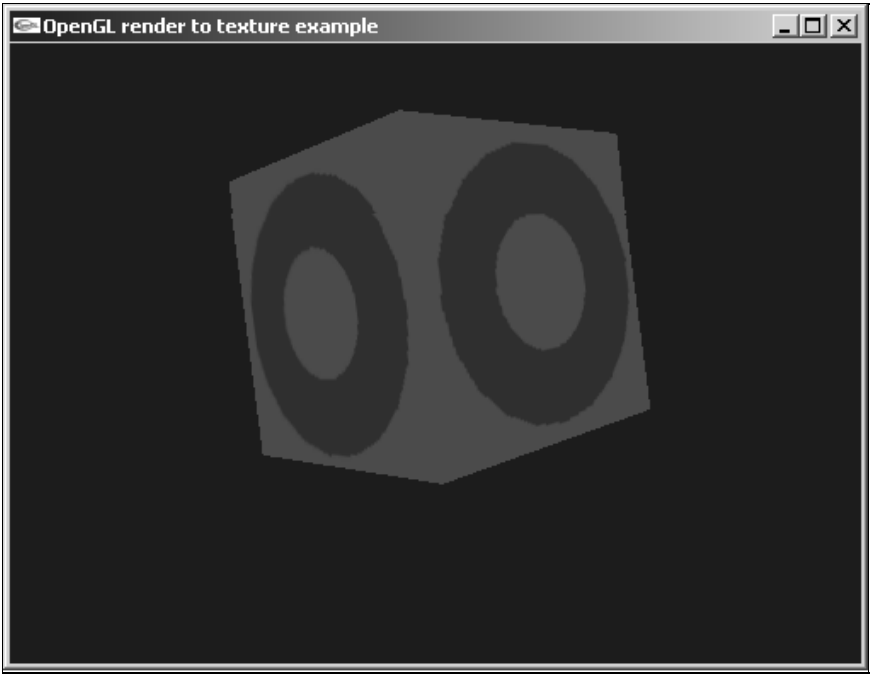


Рис. 11.1. Вращающийся куб с текстурой вращающегося чайника

Пусть строится изображение вращающегося куба, на каждую грань которого накладывается текстура, представляющая собой изображение вращающегося чайника (рис. 11.1) (выбор именно чайника объясняется тем, что это единственный из стандартных объектов GLUT, для всех вершин которого заданы текстурные координаты).

Листинг 11.6 содержит соответствующий фрагмент кода.

Листинг 11.6. Исходный код программы для рис. 11.1

```
//
// Simple rendering to texture example
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "PBuffer.h"
float        angle   = 0;
unsigned    mainTex = 0;
Vector3D    eye   ( 8, 0, 0 );            // camera position
PBuffer        pbuffer ( 128, 128, PBuffer :: modeTextureMipmap |
                         PBuffer :: modeAlpha | PBuffer :: modeDepth |
                         PBuffer :: modeTexture2D, true );
void     reshape ( int w, int h );
void    init    ();
void    initPBuffer ()
{
    static    bool    inited = false;
    if ( inited )
        return;
    inited = true;
    if ( !pbuffer.create () )
    {
        printf ( "create error\n" );
        pbuffer.printLastError ();
    }
```

```
    pbuffer.makeCurrent ();
    init ();
    reshape ( pbuffer.getWidth (), pbuffer.getHeight () );
    pbuffer.restoreCurrent ();
}
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    drawBox ( GLfloat x1, GLfloat x2, GLfloat y1, GLfloat y2,
                  GLfloat z1, GLfloat z2, unsigned tex )
{
    glBindTexture ( GL_TEXTURE_2D, tex );
    glBegin ( GL_POLYGON );         // front face
          glNormal3f   ( 0.0, 0.0, 1.0 );
          glTexCoord2f ( 0, 0 );
          glVertex3f   ( x1, y1, z2 );
          glTexCoord2f ( 1, 0 );
          glVertex3f   ( x2, y1, z2 );
          glTexCoord2f ( 1, 1 );
          glVertex3f   ( x2, y2, z2 );
          glTexCoord2f ( 0, 1 );
          glVertex3f   ( x1, y2, z2 );
        glEnd ();
        glBegin ( GL_POLYGON );         // back face
          glNormal3f ( 0.0, 0.0, -1.0 );
          glTexCoord2f ( 1, 0 );
          glVertex3f   ( x2, y1, z1 );
          glTexCoord2f ( 0, 0 );
          glVertex3f   ( x1, y1, z1 );
          glTexCoord2f ( 0, 1 );
          glVertex3f   ( x1, y2, z1 );
          glTexCoord2f ( 1, 1 );
          glVertex3f   ( x2, y2, z1 );
```

```
glEnd ();
glBegin ( GL_POLYGON );          // left face
  glNormal3f ( -1.0, 0.0, 0.0 );
  glTexCoord2f ( 0, 0 );
  glVertex3f   ( x1, y1, z1 );
  glTexCoord2f ( 0, 1 );
  glVertex3f   ( x1, y1, z2 );
  glTexCoord2f ( 1, 1 );
  glVertex3f   ( x1, y2, z2 );
  glTexCoord2f ( 1, 0 );
  glVertex3f   ( x1, y2, z1 );
glEnd ();
glBegin ( GL_POLYGON );          // right face
  glNormal3f ( 1.0, 0.0, 0.0 );
  glTexCoord2f ( 0, 1 );
  glVertex3f   ( x2, y1, z2 );
  glTexCoord2f ( 0, 0 );
  glVertex3f   ( x2, y1, z1 );
  glTexCoord2f ( 1, 0 );
  glVertex3f   ( x2, y2, z1 );
  glTexCoord2f ( 1, 1 );
  glVertex3f   ( x2, y2, z2 );
glEnd ();
glBegin ( GL_POLYGON );          // top face
  glNormal3f ( 0.0, 1.0, 0.0 );
  glTexCoord2f ( 0, 1 );
  glVertex3f   ( x1, y2, z2 );
  glTexCoord2f ( 1, 1 );
  glVertex3f   ( x2, y2, z2 );
  glTexCoord2f ( 1, 0 );
  glVertex3f   ( x2, y2, z1 );
  glTexCoord2f ( 0, 0 );
  glVertex3f   ( x1, y2, z1 );
glEnd ();
glBegin ( GL_POLYGON );          // bottom face
  glNormal3f ( 0.0, -1.0, 0.0 );
  glTexCoord2f ( 1, 1 );
  glVertex3f   ( x2, y1, z2 );
  glTexCoord2f ( 1, 0 );
```

```
          glVertex3f   ( x1, y1, z2 );
          glTexCoord2f ( 0, 0 );
          glVertex3f   ( x1, y1, z1 );
          glTexCoord2f ( 1, 0 );
          glVertex3f   ( x2, y1, z1 );
        glEnd ();
}
void    renderToBuffer ()
{
    if ( pbuffer.isLost () )
        pbuffer.create ();
    if ( !pbuffer.makeCurrent () )
        printf ( "makeCurrent failed\n" );
    glClearColor ( 0, 0.5, 0, 1 );
    glClear      ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glBindTexture  ( GL_TEXTURE_2D, mainTex );
    glPushMatrix   ();
    glRotatef      ( angle,   0, 1, 0 );
    glRotatef      ( angle/2, 0, 0, 1 );
    glutSolidTorus ( 1, 3, 20, 20 );
    glPopMatrix ();
    pbuffer.restoreCurrent ();
}
void display ()
{
    renderToBuffer ();
    glClearColor ( 0.0, 0.0, 1.0, 1.0 );
    glClear      ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glEnable     ( GL_TEXTURE_2D );
    if ( !pbuffer.bindToTexture () )
        pbuffer.printLastError ();
    glPushMatrix   ();
    glRotatef      ( angle,   1, 0, 0 );
    glRotatef      ( angle/2, 0, 1, 0 );
    drawBox ( -1.5, 1.5, -1.5, 1.5, -1.5, 1.5,
              pbuffer.getTexture () );
    glPopMatrix ();
    if ( !pbuffer.unbindFromTexture () )
        pbuffer.printLastError ();
```

```
    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 animate ()
{
    angle = 0.08 * glutGet ( GLUT_ELAPSED_TIME );
    glutPostRedisplay ();
}
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 |
                          GLUT_ALPHA );
    glutInitWindowSize  ( 500, 500 );
                            // create window
    int win = glutCreateWindow ( "OpenGL render to texture example" );
                            // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init ();
```

```
    printfInfo     ();
    initExtensions ();
    mainTex = createTexture2D ( true, "../Textures/block.bmp" );
    initPBuffer ();

    glutMainLoop ();
    return 0;
}
```

Рассмотрим еще один пример (его идея взята из работы по курсу компьютерной графики на факультете ВМиК МГУ в 2004 году). Строится изображение объекта в *p*-буфер, а затем полученная текстура используется для вывода прямоугольной сетки вершин. Сами вершины при этом подвергаются небольшим смещениям для создания эффекта "горячего воздуха над огнем" (рис. 11.2).

Рис. 11.2. Эффект "горячего воздуха"

Листинг 11.7 содержит программную реализацию этого примера.

Листинг 11.7. Эффект "горячего воздуха"

```
//
// Simple rendering to texture example
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "PBuffer.h"
float       angle   = 0;
unsigned    mainTex = 0;
Vector3D    eye     ( 4.5, 0, 0 );            // camera position
PBuffer     pbuffer ( 256, 256, PBuffer :: modeTextureMipmap |
                        PBuffer :: modeAlpha | PBuffer :: modeDepth |
                        PBuffer :: modeTexture2D, true );
Vector3D    vertexGrid  [30][30];
Vector2D    textureGrid [30][30];
void     reshape ( int w, int h );
void    init    ();
void    initPBuffer ()
{
    static    bool    inited = false;
    if ( inited )
        return;
    inited = true;
    if ( !pbuffer.create () )
    {
        printf ( "create error\n" );
        pbuffer.printLastError ();
    }
    pbuffer.makeCurrent ();
    init ();
    reshape ( pbuffer.getWidth (), pbuffer.getHeight () );
    pbuffer.restoreCurrent ();
}
```

```
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    renderToBuffer ()
{
    if ( pbuffer.isLost () )
        pbuffer.create ();
    if ( !pbuffer.makeCurrent () )
        printf ( "makeCurrent failed\n" );
    glClearColor ( 0, 0, 1, 1 );
    glClear      ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glEnable       ( GL_TEXTURE_2D );
    glBindTexture  ( GL_TEXTURE_2D, mainTex );
    glMatrixMode ( GL_TEXTURE );
    glLoadIdentity ();
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix   ();
    glRotatef      ( 820,  0, 1, 0 );
    glRotatef      ( 445,  0, 0, 1 );
    glutSolidTeapot ( 1.5 );
    glPopMatrix ();
    pbuffer.restoreCurrent ();
}
void    distortVertex ( Vector3D& v, float time )
{
    float    sine   = 0.015 * sin ( 11*v.y + 16*v.z + time / 5 );
    float    cosine = 0.015 * cos ( 7*v.y - 14*v.z + time / 5 );
    v.y += sine;
    v.z += cosine;
}
void display ()
{
    initPBuffer ();
```

```
    renderToBuffer ();
    glClearColor ( 0.0, 0.0, 1.0, 1.0 );
    glClear      ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glEnable     ( GL_TEXTURE_2D );
    if ( !pbuffer.bindToTexture () )
        pbuffer.printLastError ();
    glBindTexture ( GL_TEXTURE_2D, pbuffer.getTexture () );
    float    time = 0.08 * glutGet ( GLUT_ELAPSED_TIME );
    glBegin ( GL_TRIANGLES );
    for ( int i = 0; i < 29; i++ )
        for ( int j = 0; j < 29; j++ )
        {
            Vector3D    v00 ( vertexGrid  [i][j    ] );
            Vector3D    v01 ( vertexGrid  [i][j+1]   );
            Vector3D    v10 ( vertexGrid  [i+1][j]   );
            Vector3D    v11 ( vertexGrid  [i+1][j+1] );
            distortVertex ( v00, time );
            distortVertex ( v01, time );
            distortVertex ( v10, time );
            distortVertex ( v11, time );
            glTexCoord2fv ( textureGrid [i][j] );
            glVertex3fv   ( v00  );
            glTexCoord2fv ( textureGrid [i][j+1] );
            glVertex3fv   ( v01 );
            glTexCoord2fv ( textureGrid [i+1][j+1] );
            glVertex3fv   ( v11 );
            glTexCoord2fv ( textureGrid [i][j] );
            glVertex3fv   ( v00 );
            glTexCoord2fv ( textureGrid [i+1][j] );
            glVertex3fv   ( v10 );
            glTexCoord2fv ( textureGrid [i+1][j+1] );
            glVertex3fv   ( v11 );
        }
    glEnd ();
    if ( !pbuffer.unbindFromTexture () )
        pbuffer.printLastError ();
    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 animate ()
{
    angle = 0.08 * glutGet ( GLUT_ELAPSED_TIME );
    glutPostRedisplay ();
}
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 |
                          GLUT_ALPHA );
    glutInitWindowSize  ( 500, 500 );
                            // create window
    int win = glutCreateWindow ( "OpenGL render to texture example 2
(distort)" );
                            // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init ();
    printfInfo ();
```

```
    initExtensions ();
    initPBuffer    ();
    mainTex = createTexture2D ( false,
                          "../Textures/fieldstone.tga" );//block.bmp" );
    for ( int i = 0; i < 30; i++ )
        for ( int j = 0; j < 30; j++ )
       {
        vertexGrid  [i][j] = Vector3D(0,(i-15)*0.09,(j-15)*0.09);
        textureGrid [i][j] = Vector2D((float)i/29.0,
                                                  (float)j/29.0f);
       }
    glutMainLoop ();
    return 0;
}
```

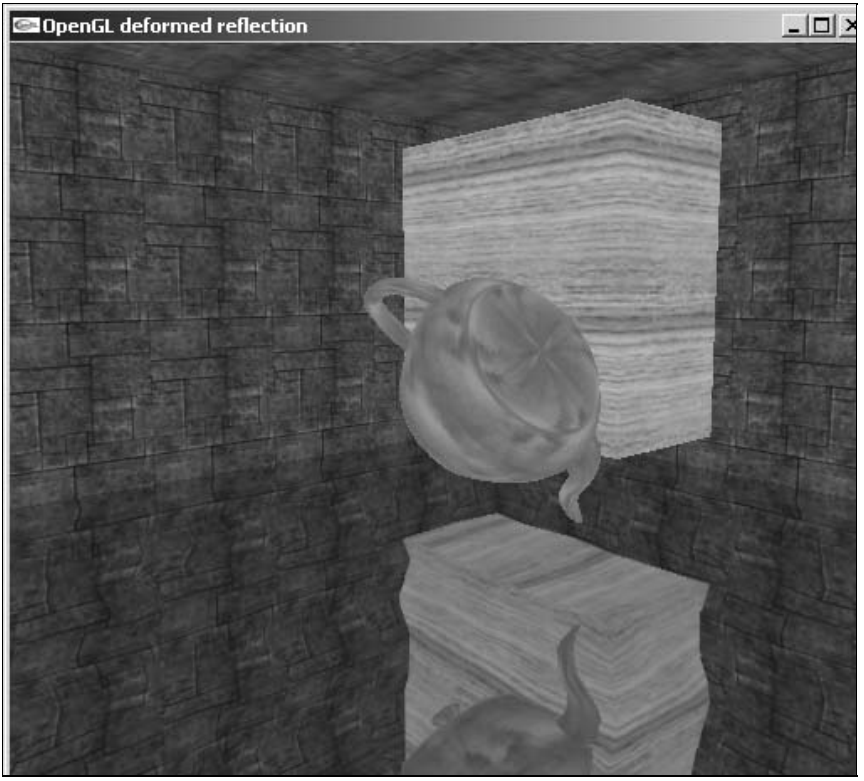


Рис. 11.3. Искаженное отражение сцены

Еще одним интересным примером является искажение отражения: строится изображение отраженной в текстуру сцены, и она накладывается на грань. При этом, как и в предыдущем примере, грань выводится как набор большого числа треугольников, текстурные координаты которых подвергаются небольшим искажениям (рис. 11.3).

Исходный текст программы приводится в листинге 11.8.

Листинг 11.8. Искажение отражения сцены

```
//
// Sample to rendering to texture - deformed reflection
//
#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    "Vector4D.h"
#include    "boxes.h"
#include    "PBuffer.h"
#define GRID_SIZE   30                  // grid subdivision
Vector3D    eye   ( -0.5, -0.5, 1.5 );  // camera position
unsigned    decalMap;                   // decal (diffuse) texture
unsigned    stoneMap;
unsigned    teapotMap;
float       angle     = 0;
float       rot       = 0;
                                  // to hold modelview matrix used for
                                  // building reflection texture
float   mv [16];
                                   // to hold projection matrix used for
                                  // building reflection texture
float   pr [16];
Vector3D    v [GRID_SIZE][GRID_SIZE];   // grid vertices for floor
Vector3D    t [GRID_SIZE][GRID_SIZE];   // texture coordinates for floor
vertices
```

```
Vector3D    floorPos  ( -5, -5, 0 );
Vector3D    floorSize ( 10, 10, 3 );
PBuffer     pbuffer ( 512, 512, PBuffer :: modeTextureMipmap |
                      PBuffer :: modeAlpha | PBuffer :: modeDepth |
                      PBuffer :: modeTexture2D, true );
void    renderToBuffer  ();
void    reshapeMirrored ( int w, int h );
void    reshape         ( int w, int h );
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 ();
    drawBoxNoBottom ( Vector3D ( -5, -5, 0 ), Vector3D ( 10, 10, 3 ),
                      stoneMap, false );
    drawBox         ( Vector3D ( 3, 2, 0.5 ), Vector3D ( 1,  2,  2 ),
                      decalMap, false );
    glBindTexture   ( GL_TEXTURE_2D, teapotMap );
    glTranslatef    ( 2, 2, 1 );
    glRotatef       ( angle * 45.3, 1, 0, 0 );
    glRotatef       ( angle * 57.2, 0, 1, 0 );
    glutSolidTeapot ( 0.5 );
    glPopMatrix ();                       // restore modelview matrix
}
//
// distort texture coordinates for internal vertices by moving them
// sligtly with the time
//
void    distortVertices ( float time )
{
    float   hx = floorSize.x / (float) (GRID_SIZE - 1);
```

```
    float   hy = floorSize.y / (float) (GRID_SIZE - 1);
    for ( int i = 1; i < GRID_SIZE - 1; i++ )
        for ( int j = 1; j < GRID_SIZE - 1; j++ )
        {
            Vector3D    p ( v [i][j] );
            float   sine   = 0.03 * sin (  11*p.y + 16*p.x + 2*time );
            float   cosine = 0.03 * cos ( -15*p.y - 14*p.x + 2*time );
            t [i][j].x = v [i][j].x + sine;
            t [i][j].y = v [i][j].y + cosine;
            t [i][j].z = v [i][j].z;
        }
}
void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    renderToBuffer ();          // render reflected image into pbuffer
    reshape ( 512, 512 );
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();            // save modelview matrix
    glRotatef    ( rot, 0, 0, 1 );
    displayBoxes ();
    if ( !pbuffer.bindToTexture () )
        pbuffer.printLastError ();
    glMatrixMode  ( GL_TEXTURE );
    glPushMatrix  ();
    glLoadIdentity ();
    glTranslatef   ( 0.5, 0.5, 0 );     // remap from [-1,1]^2 to [0,1]^2
    glScalef       ( 0.5, 0.5, 1 );
    glMultMatrixf  ( pr );
    glMultMatrixf  ( mv );
    float       x2 = floorPos.x + floorSize.x;
    float       y2 = floorPos.y + floorSize.y;
    glBindTexture ( GL_TEXTURE_2D, pbuffer.getTexture () );
                                        // draw floor
    glColor3f ( 0.8, 0.8, 0.8 );        // make it a bit darker
                                        // draw the distorted mesh
    glBegin ( GL_TRIANGLES );
    glNormal3f ( 0, 0, -1 );
```

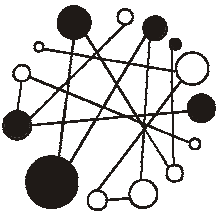
```
    for ( int i = 0; i < GRID_SIZE - 1; i++ )
        for ( int j = 0; j < GRID_SIZE - 1; j++ )
        {
            glTexCoord3fv ( t [i][j] );
            glVertex3fv   ( v [i][j] );
            glTexCoord3fv ( t [i][j+1] );
            glVertex3fv   ( v [i][j+1] );
            glTexCoord3fv ( t [i+1][j+1] );
            glVertex3fv   ( v [i+1][j+1] );
            glTexCoord3fv ( t [i][j] );
            glVertex3fv   ( v [i][j] );
            glTexCoord3fv ( t [i+1][j] );
            glVertex3fv   ( v [i+1][j] );
            glTexCoord3fv ( t [i+1][j+1] );
            glVertex3fv   ( v [i+1][j+1] );
        }
    glEnd ();
    glColor3f ( 1, 1, 1 );
    if ( !pbuffer.unbindFromTexture () )
        pbuffer.printLastError ();
    glPopMatrix ();
    glMatrixMode ( GL_MODELVIEW );
    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
                    3, 3, 1,                // center
                    0, 0, 1 );              // up
}
void reshapeMirrored ( 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 ();
   glScalef       ( 1, -1, 1 );             // change handedness of the
                                            // coordinate system
   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 );
}
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  = 0.001f * glutGet ( GLUT_ELAPSED_TIME );
    distortVertices ( angle );
    glutPostRedisplay ();
}
void    initPBuffer ()
{
    if ( !pbuffer.create () )
    {
        printf ( "create error\n" );
        pbuffer.printLastError ();
```

```
    }
    pbuffer.makeCurrent ();
    init ();
    reshapeMirrored ( pbuffer.getWidth (), pbuffer.getHeight () );
    pbuffer.restoreCurrent ();
}
void    renderToBuffer ()
{
    if ( pbuffer.isLost () )
        pbuffer.create ();
    if ( !pbuffer.makeCurrent () )
        printf ( "makeCurrent failed\n" );
    glClearColor ( 0, 0, 1, 1 );
    glClear      ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    reshapeMirrored ( pbuffer.getWidth (), pbuffer.getHeight () );
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();
    glRotatef    ( rot, 0, 0, 1 );
    displayBoxes ();
                              // get modelview and projections matrices
    glGetFloatv ( GL_MODELVIEW_MATRIX,  mv );
    glGetFloatv ( GL_PROJECTION_MATRIX, pr );
    glPopMatrix ();
    pbuffer.restoreCurrent ();
}
int main ( int argc, char * argv [] )
{
                                // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 512, 512 );
                                // create window
    int win = glutCreateWindow ( "OpenGL deformed reflection" );
                                // register handlers
    glutDisplayFunc  ( display    );
    glutReshapeFunc  ( reshape    );
    glutKeyboardFunc ( key        );
    glutSpecialFunc  ( specialKey );
    glutIdleFunc     ( animate    );
```

```
    init           ();
    initExtensions ();
    if ( !pbuffer.checkExtensions () )
    {
        printf ( "Pbuffer extensions not found\n" );
        return 1;
    }
    decalMap  = createTexture2D ( true, "../Textures/oak.bmp" );
    stoneMap  = createTexture2D ( true, "../Textures/block.bmp" );
    teapotMap = createTexture2D ( true, "../Textures/Oxidated.jpg" );
                                // init vertices
    float   hx = floorSize.x / (float) (GRID_SIZE - 1);
    float   hy = floorSize.y / (float) (GRID_SIZE - 1);
    for ( int i = 0; i < GRID_SIZE; i++ )
        for ( int j = 0; j < GRID_SIZE; j++ )
        {
            float   x = floorPos.x + i * hx;
            float   y = floorPos.y + j * hy;
            v [i][j] = Vector3D ( x, y, floorPos.z );
            t [i][j] = Vector3D ( x, y, floorPos.z );
        }
    initPBuffer  ();
    glutMainLoop ();
    return 0;
}
```

Глава 12



Расширение NV_texture_shader. EMBM и попиксельное отражение окружающей среды с учетом карты нормалей

Разработанное для графических ускорителей GeForce 3 расширение NV_texture_shader позволяет изменить механизм применения текстур в OpenGL. Вместо традиционной модели вводится понятие шагов текстурирования (*texture stage*), каждому шагу соответствует свой текстурный блок и на каждом из этих шагов работает одна из 21 предопределенных программ (шейдеров, *shaders*), задающая выходной цвет для данного текстурного блока (рис. 12.1).

Каждая из этих программ в качестве входных данных берет интерполированные текстурные координаты (*s*, *t*, *r*, *q*) для соответствующего блока и выдает два значения: RGBA-значение, с данного текстурного блока (например, при работе с блоками register combiner) и результат данного шага, который может быть использован в дальнейшем.

Отдельные программы могут приводить к различным побочным эффектам: отбрасыванию данного фрагмента или изменению его значения в буфере глубины.

Проверку поддержки расширения NV_texture_shader можно выполнить командой

```
isExtensionSupported ( "GL_NV_texture_shader" )
```

Данное расширение не вводит ни одной новой функции, но зато очень много различных констант для задания программ их параметров на разных шагах.

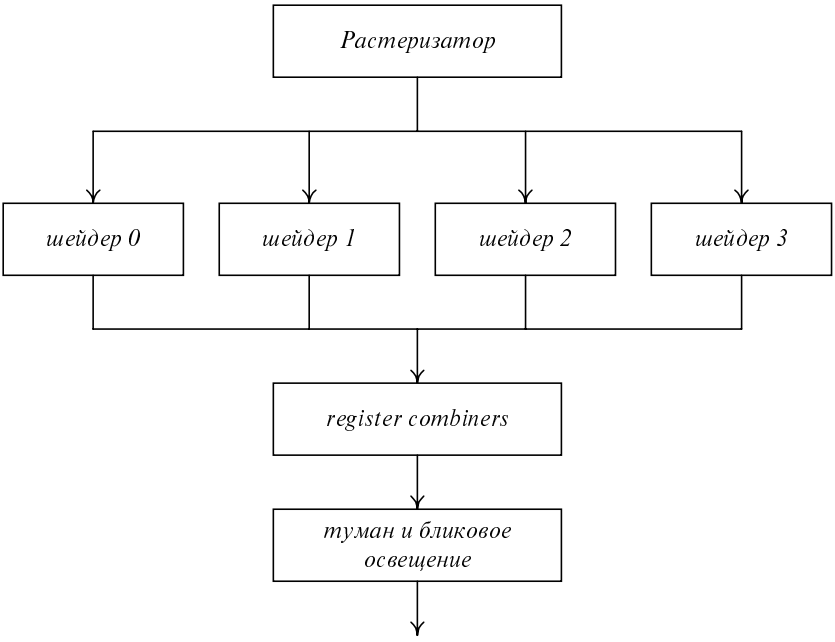


Рис. 12.1. Реализация текстурирования через расширение NV_texture_shader

Эти программы, собственно и называемые шейдерами, (*texture shaders*) можно разбить на четыре основные группы по функциональному признаку:

1. Обычные операции текстурирования.
2. Специальные режимы текстурирования.
3. Режимы, зависящие от результатов другого шага.
4. Режимы, зависящие от результатов другого шага и использующие скалярное произведение.

Для включения шейдеров служит команда

```
glEnable ( GL_TEXTURE_SHADER_NV );
```

Обратите внимание, что при включении режима текстурных шейдеров все четыре шага присутствуют автоматически. Если какие-либо шаги не нужны, то для них следует установить шейдер `GL_NONE`.

Помните, что некоторые шейдеры возвращают в качестве выходного RGBA-значения нулевой вектор (0, 0, 0, 0) независимо от текстуры, выбранной в данном блоке текстурирования.

Обычные операции текстурирования

Одномерные текстуры (Texture 1D)

Данная операция возвращает значение одномерной текстуры по параметру (*s*/*q*). Фактически это стандартное одномерное текстурирование в OpenGL (табл. 12.1).

Таблица 12.1. *Обычное одномерное текстурирование*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	(*s*, *t*, *r*, *q*)	Texture 1D	(*s*/*q*)	Любая	*tex*(*s*/*q*)

Данный режим (шейдер) устанавливается при помощи команды

```
glTexEnvi ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
              GL_TEXTURE_1D );
```

Двумерные текстуры (Texture 2D)

Данная операция возвращает значение двумерной текстуры по параметру (*s*/*q*, *t*/*q*).

Она полностью аналогична стандартному двумерному текстурированию (табл. 12.2).

Таблица 12.2. *Обычное двумерное текстурирование*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	(*s*, *t*, *r*, *q*)	Texture 2D	(*s*/*q*, *t*/*q*)	Любая	*tex*(*s*/*q*, *t*/*q*)

Данный режим (шейдер) устанавливается при помощи команды

```
glTexEnvi ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
            GL_TEXTURE_2D );
```

Прямоугольные текстуры (Texture rectange)

Это новый тип текстуры (NV_texture_rectangle) позволяющий ее размерам *width* и *height* быть произвольными целыми числами (а не только степенью двойки).

Текстурные координаты *s* и *t* для таких текстур принимают значения в области [0, *width*]×[0, *height*] вместо единичного квадрата (табл. 12.3).

Таблица 12.3. *Двумерное прямоугольное текстурирование*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	(*s*, *t*, *r*, *q*)	Texture rectangle	(*s*/*q*, *t*/*q*)	Любая прямоугольная	*tex*(*s*/*q*, *t*/*q*)

Данный режим (шейдер) устанавливается при помощи команды

```
glTexEnvi ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
            GL_TEXTURE_RECTANGLE_NV );
```

Кубические текстуры (Texture cube map)

Этот шейдер соответствует обычной кубической текстурной карте (табл. 12.4).

Таблица 12.4. *Кубические текстуры*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	(*s*, *t*, *r*)	Texture cube map	(*q*, *t*, *r*)	Любая кубическая	*cubeMap*(*s*, *t*, *r*)

Данный режим (шейдер) устанавливается при помощи команды

```
glTexEnvi ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
               GL_TEXTURE_CUBE_MAP_ARB );
```

Специальные режимы текстурирования

Пустой (None)

Данный шейдер вообще не обращается к текстуре, игнорирует переданные текстурные координаты и всегда возвращает нулевой вектор (0, 0, 0, 0).

Установить режим можно командой

```
glTexEnvi ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
            GL_NONE );
```

Стандартное применение этого шейдера заключается в его назначении на ненужные шаги текстурирования.

Прохождение (Pass-Through)

Данный шейдер также не обращается к текстуре и просто возвращает в качестве выходных RGBA-значения текстурных координат, усеченные по отрезку [0, 1] (табл. 12.5). Его работа иллюстрируется рис. 12.2.

Таблица 12.5. *Работа шейдера Pass-Through*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	(*s*, *t*, *r*, *q*)	*r* = *clamp*(*s*) *g* = *clamp*(*t*) *b* = *clamp*(*q*) *a* = *clamp*(*q*)	Нет	Любая	(*r*, *g*, *b*, *a*)

Установить режим можно командой

```
glTexEnvi ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
            GL_PASS_THROUGH_NV );
```

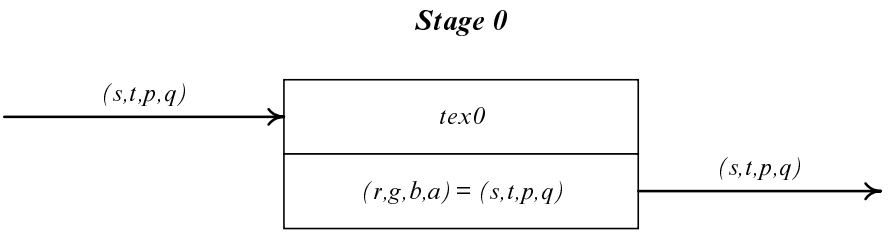


Рис. 12.2. Работа шейдера Pass-Through

Одним из возможных применений данного шейдера является непосредственная передача параметров в блоки register combiner. Дело в том, что большинство параметров для register combiner передаются в виде текстурных координат, и они не могут непосредственно участвовать в вычислениях (а только как результаты их применения к соответствующим текстурам).

Этот шейдер позволяет передать сами текстурные координаты как результат операции текстурирования. Довольно простым примером его использования является учет расстояния до источника света при попиксельном освещении.

$$k(d) = \frac{1}{a + b \cdot d + c \cdot d^2}. \qquad (12.1)$$

Стандартная формула (12.1), используемая в OpenGL, слишком сложна для попиксельной реализации (за исключением фрагментных программ и шейдерных языков высокого уровня). Ее можно заменить гораздо более простой, легко реализуемой через механизм register combiner:

$$k(d) = clamp\left(1 - \left(\frac{d}{r}\right)^2\right).$$

В этой формуле функция *clamp* осуществляет отсечение своего аргумента по отрезку [0, 1] , а параметр *r* задает "радиус влияния" источника света (вне его освещенность равна нулю).

Для реализации попиксельного освещения по этой формуле величину *l/r* можно передать в виде текстурных координат, а все остальные шаги легко реализуются при помощи register combiner. Однако для этого надо непосредственно вычислить скалярное произведение переданных текстурных координат, а не извлекать по нему значение из какой-либо текстуры.

Шейдер `GL_PASS_THROUGH_NV` как раз и предоставляет такую возможность. Мы передадим нужную нам величину в виде текстурных координат и для соответствующего текстурного блока зададим шейдер `GL_PASS_THROUGH_NV`.

В результате этого на выходе операции текстурирования для соответствующего текстурного блока мы получим величину *l/r*, отсеченную по отрезку [0, 1], которая сразу же пригодна для вычислений. Пример такого подхода иллюстрирует листинг 12.1.

Листинг 12.1. Реализация освещение с учетом расстояния до источника света

```
//
// Sample to show distance attenuation in OpenGL
// using PASS_THROUGH_NV shader
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    "libTexture.h"
#include    "TypeDefs.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
```

```
Vector3D    eye   ( 0, 0.3, -3.5 );      // camera position
Vector3D    light ( 0.3, 0.7, 0.9f );    // light position
Vector3D    vl    ( 0.4, 0.4, -0.4 );    // light velocity
float       lightRadius = 1;
unsigned    decalMap;                    // decal (diffuse) texture
struct    Vertex
{
    Vector2D    tex;
    Vector3D    pos;
};
Vertex    front [4];
Vertex    lower [4];
Vertex    upper [4];
Vertex    left  [4];
Vertex    right [4];
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    drawFace ( Vertex v [], int count )
{
    glBegin ( GL_QUADS );

    for ( int i = 0; i < count; i++ )
    {
        Vector3D    d = v [i].pos - light;
        d = Vector3D ( 0.5, 0.5, 0.5f ) + d * (0.5 / lightRadius );
        glMultiTexCoord2fv ( GL_TEXTURE0_ARB, v [i].tex );
        glMultiTexCoord3fv ( GL_TEXTURE1_ARB, d );
        glVertex3fv        ( v [i].pos );
    }
    glEnd   ();
}
```

```
void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
                        // bind decal map to
                        // texture unit 0
                        // setup texture shader
    glEnable  ( GL_TEXTURE_SHADER_NV );
                        // unit 0 -> decal
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D   );
    glBindTexture      ( GL_TEXTURE_2D, decalMap );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                         GL_TEXTURE_2D );
                        // unit 1 -> pass through
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                         GL_PASS_THROUGH_NV );
    glActiveTextureARB ( GL_TEXTURE0_ARB );
                        // setup register combiners
    glEnable               ( GL_REGISTER_COMBINERS_NV );
    glCombinerParameteriNV ( GL_NUM_GENERAL_COMBINERS_NV, 1 );
                        // combiner0 computes
                        // (tex1,tex1) -> spare0
                        // A = texture1
    glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
                        GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                        GL_RGB );
                        // B = texture1
    glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
                        GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                        GL_RGB );
                        // spare0 = tex1 + tex2
    glCombinerOutputNV ( GL_COMBINER0_NV, GL_RGB,
                         GL_SPARE0_NV,        // put (A,B) into spare0
                         GL_DISCARD_NV,       // discard CD
                         GL_DISCARD_NV,     // discard AB+CD
                         GL_NONE, GL_NONE,    // no bias or scale
                         GL_TRUE,             // use dot porduct
```

```
                          GL_FALSE,
                          GL_FALSE );
                         // configure final combiner to
                         // output clamp (1 - spare0)
                         // A = 1 - spare0
    glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_SPARE0_NV,
                             GL_UNSIGNED_INVERT_NV, GL_RGB );
                         // B = tex0 (decal)
    glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_TEXTURE0_ARB,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                         // C = 0
    glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                         // D = 0
    glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
    drawFace ( front, 4 );
    drawFace ( lower, 4 );
    drawFace ( upper, 4 );
    drawFace ( left,  4 );
    drawFace ( right, 4 );
    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 key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
```

```
            exit ( 0 );
}
void    animate ()
{
    Vector3D    delta = vl * 0.01f;
    light += delta;
    if ( light.x < 0.2f || light.x > 0.9f )
        vl.x *= -1;
    if ( light.y < -0.9f || light.y > 0.9f )
        vl.y *= -1;
    if ( light.z < -0.9f || light.z > 0.9f )
        vl.z *= -1;
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                    // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                    // create window
    int    win = glutCreateWindow ( "OpenGL per-pixel distance
    attenuation using PASS_THROUGH shader" );
                    // register handlers
glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init ();
    printfInfo ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_NV_register_combiners" ) )
    {
        printf ( "NV_register_combiners NOT supported" );
        return 2;
```

```
}
if ( !isExtensionSupported ( "GL_NV_texture_shader" ) )
{
    printf ( "NV_register_combiners NOT supported" );
    return 2;
}
initExtensions ();
decalMap   = createTexture2D      ( true, "../Textures/oak.bmp" );
front [0].tex = Vector2D ( 0, 0 );
front [0].pos = Vector3D ( -1, -1, 1 );
front [1].tex = Vector2D ( 0, 1 );
front [1].pos = Vector3D ( -1,  1, 1 );
front [2].tex = Vector2D ( 1, 1 );
front [2].pos = Vector3D (  1,  1, 1 );
front [3].tex = Vector2D ( 1, 0 );
front [3].pos = Vector3D (  1, -1, 1 );
lower [0].tex = Vector2D (  0, 0 );
lower [0].pos = Vector3D ( -1, -1, -1 );
lower [1].tex = Vector2D (  0, 1 );
lower [1].pos = Vector3D ( -1, -1,  1 );
lower [2].tex = Vector2D (  1, 1 );
lower [2].pos = Vector3D (  1, -1,  1 );
lower [3].tex = Vector2D (  1, 0 );
lower [3].pos = Vector3D (  1, -1, -1 );
upper [0].tex = Vector2D (  0, 0 );
upper [0].pos = Vector3D ( -1, 1, -1 );
upper [1].tex = Vector2D (  0, 1 );
upper [1].pos = Vector3D ( -1, 1,  1 );
upper [2].tex = Vector2D (  1, 1 );
upper [2].pos = Vector3D (  1, 1,  1 );
upper [3].tex = Vector2D (  1, 0 );
upper [3].pos = Vector3D (  1, 1, -1 );
left [0].tex = Vector2D ( 0, 0 );
left [0].pos = Vector3D ( -1, -1, -1 );
left [1].tex = Vector2D ( 0, 1 );
left [1].pos = Vector3D ( -1, -1, 1 );
left [2].tex = Vector2D ( 1, 1 );
left [2].pos = Vector3D ( -1, 1, 1 );
```

```
    left [3].tex = Vector2D ( 1, 0 );
    left [3].pos = Vector3D ( -1, 1, -1 );
    right [0].tex = Vector2D ( 0, 0 );
    right [0].pos = Vector3D ( 1, -1, -1 );
    right [1].tex = Vector2D ( 0, 1 );
    right [1].pos = Vector3D ( 1, -1, 1 );
    right [2].tex = Vector2D ( 1, 1 );
    right [2].pos = Vector3D ( 1, 1, 1 );
    right [3].tex = Vector2D ( 1, 0 );
    right [3].pos = Vector3D ( 1, 1, -1 );
    glutMainLoop ();
    return 0;
}
```

Отсечение фрагмента (Cull Fragment)

Данный шейдер позволяет отбросить фрагмент (пиксел) в зависимости от значения его текстурных координат.

Для каждой текстурной координаты ставится свое условие прохождения. Это может быть либо `GL_GEQUAL` (значение соответствующей координаты больше или равно нулю), либо `GL_LESS` (значение отрицательно).

Если для хотя бы одной из четырех текстурных координат условие прохождения не выполняется, то фрагмент отбрасывается. В противном случае фрагмент проходит, и в качестве выходного RGBA-значения выступает нулевой вектор (0, 0, 0, 0). Листинг 12.2 иллюстрирует эту возможность.

Листинг 12.2. Задание режимов для отсечения фрагмента

```
int  cullModes [4] = { GL_LESS, GL_LESS, GL_GEQUAL, GL_LESS };
glActiveTextureARB ( GL_TEXTURE0_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_CULL_FRAGMENT_NV );
glTexEnviv         ( GL_TEXTURE_SHADER_NV, GL_CULL_MODES_NV,
                     cullModes );
```

На практике данный шейдер применяется для выполнения условных операций при вычислении цвета фрагмента.

Многошаговое текстурирование

Все шейдеры из этой группы позволяют взять значение из одной текстуры и использовать его для адресации другой.

Поэтому на данном шаге всегда необходим RGBA-результат другого шага, причем эти операции не обязаны следовать друг за другом. Однако для простоты мы будем нумеровать шаги один за другим, начиная с нуля.

Двухшаговое текстурирование с использованием красной и альфа-компонет (Dependent Alpha-Red Texturing)

Данный шейдер берет красную (R) и альфа (A) компоненты с одного из шагов текстурирования (не обязательно предыдущего) и использует как текстурные координаты для доступа к другой двумерной текстуре (табл. 12.6).

Таблица 12.6. *Работа шейдера Dependent Alpha-Red Texturing*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	Зависят от приложения	Зависит от текстуры	Зависит от текстуры	Любая 2D RGBA-текстура	$R_0G_0B_0A_0$
1	Игнорируются	Dependent Alpha-Red Texturing	(R_0, A_0)	Любая 2D RGBA-текстура	$R_1G_1B_1A_1$

Работа шейдера представлена на рис. 12.3. Листинг 12.3 содержит пример программы, реализующей этот метод.

Листинг 12.3. Пример задания зависимого чтения alpha-red

```
                        // setup stage 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_TEXTURE_2D );
                        // setup stage 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DEPENDENT_AR_TEXTURE_2D_NV );
```

```
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV,
                     GL_TEXTURE0_ARB );
```

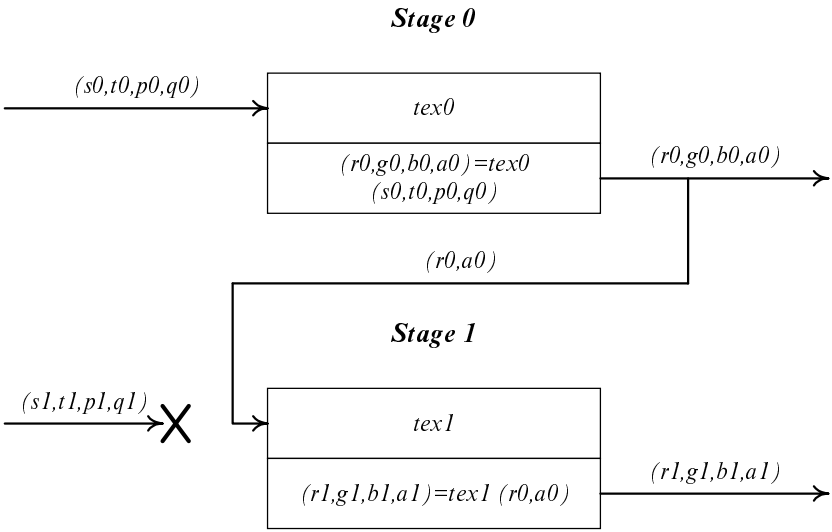


Рис. 12.3. Работа шейдера Dependent Alpha-Red Texturing

Обратите внимание на то, что для второго шага явно задается номер шага текстурирования, из которого необходимо взять выходные RGBA-значения для их использования в текстурных координатах.

Двухшаговое текстурирование с использованием зеленой и синей компонент (Dependent Green-Blue Texturing)

Данный шейдер полностью аналогичен предыдущему и отличается от него только тем, что в качестве текстурных координат для доступа к другой текстуре служат зеленая (G) и синяя (B) компоненты (табл. 12.7). Листинг 12.4 иллюстрирует пример настройки данного шейдера.

Таблица 12.7. *Работа шейдера Dependent Green-Blue Texturing*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	Зависят от приложения	Зависит от текстуры	Зависит от текстуры	Любая 2D RGBA-текстура	$R_0G_0B_0A_0$

Таблица 12.7 *(окончание)*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
1	Игнорируются	Нет	(G_0, B_0)	Любая 2D RGBA-текстура	$R_1G_1B_1A_1$

Листинг 12.4. Пример настройки для использования Dependent Green-Blue Texturing

```
                        // setup stage 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_TEXTURE_2D );
                        // setup stage 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DEPENDENT_GB_TEXTURE_2D_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
```

Двумерное текстурирование со смещением (Offset Texture 2D)

Этот шейдер позволяет по значениям из одной текстуры изменить текстурные координаты для доступа к другой. При этом полученные из текстуры два значения умножаются на матрицу 2×2 из вещественных чисел, и полученные значения добавляются к обычным текстурным координатам (табл. 12.8).

Шейдер использует новые форматы текстур: `GL_DSDT_NV` (два знаковых канала по 8 битов на канал), `GL_DSDT_MAG_NV` (три 8-битовых канала, последний из них беззнаковый) и `GL_DSDT_MAG_INTENSITY_NV` (четыре канала, последние два — беззнаковые).

Таблица 12.8. *Работа шейдера Offset Texture 2D*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	(s_0, t_0, r_0, q_0)	Texture 2D	(s_0/q_0, t_0/q_0)	DSDT	$R_0G_0B_0A_0$ (ds, dt)

Таблица 12.8 *(окончание)*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
1	(s_1, t_1)	Offset Texture 2D	$s'_1=s_1+k0\cdot ds+k2\cdot dt$ $t'_1=t_1+k1\cdot ds+k3\cdot dt$	Любая 2D. RGBA-текстура	$R_1G_1B_1A_1 =$ $= tex(s'_1, t'_1)$

Рисунок 12.4 показывает работу данного шейдера, а листинг 12.5 — пример настройки для его использования.

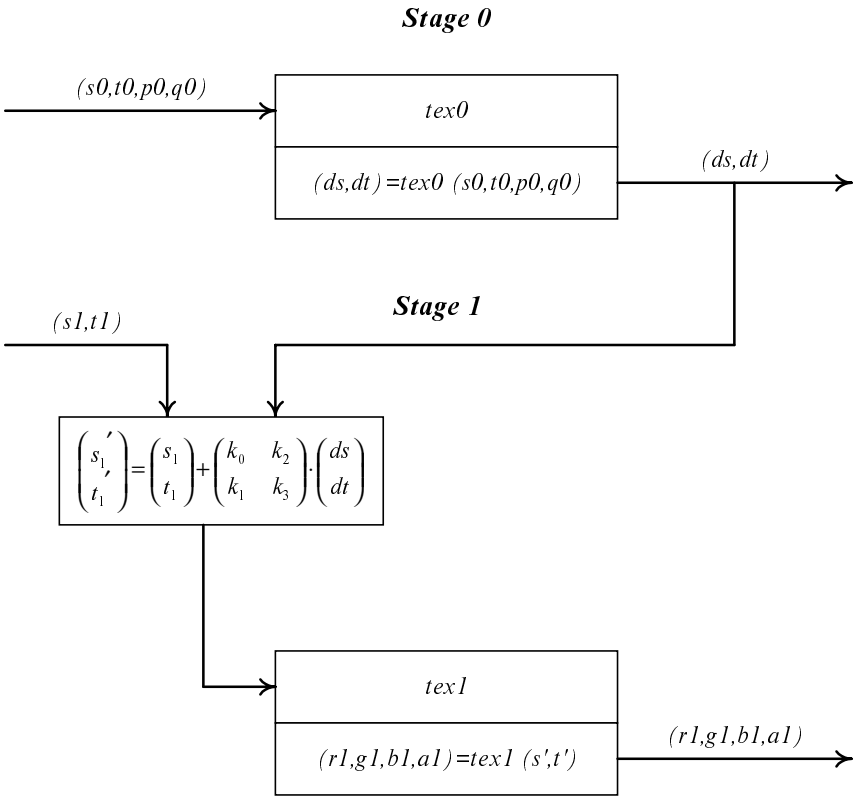


Рис. 12.4. Работа шейдера Offset Texture 2D

Листинг 12.5. Пример настройки для использования шейдера Offset Texture 2D

```
                    // setup stage 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
```

```
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_TEXTURE_2D );
                          // setup stage 1
float    k [4] = { 0.3, 0.5, 0.7, 0.5 };
glActiveTextureARB ( GL_TEXTURE1_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_OFFSET_TEXTURE_2D_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
glTexEnvfv         ( GL_TEXTURE_SHADER_NV,
                     GL_OFFSET_TEXTURE_MATRIX_NV, k );
```

В этом примере помимо явного задания предыдущего шага также задается и матрица `k` преобразования текстурных координат.

Обратите внимание, что в данном примере встречается новый формат текстуры — `GL_DSDT_NV`. Каждый ее тексел состоит из двух байтов, определяющих пару значений из отрезка [−1, 1].

Таким образом, DSDT-текстура фактически задает микрорельеф на поверхности объекта. Подобный способ имитации рельефа при отражении называется EMBM (*Environment Map Bump Mapping*). Обратите внимание, что данный способ моделирования рельефа поверхности является приближенным. Точный способ рассматривается далее в этой главе.

Двумерное текстурирование со смещением и масштабированием (Offset Texture 2D scale)

Данный шейдер аналогичен предыдущему, но здесь RGB-компоненты выходного цвета подвергаются масштабированию с коэффициентом MAG, вычисляемым при помощи задаваемых пользователем значений k_{scale} и k_{bias}. Обратите внимание, что альфа-канал масштабированию не подвергается (табл. 12.9).

В случае если текстура, задающая смещение, имеет формат `GL_DSDT_MAG_INTENSITY`, то на соответствующем текстурном блоке возвращается яркость во всех каналах (*i*, *i*, *i*, *i*), иначе возвращается значение (0, 0, 0, 0).

Листинг 12.6 содержит пример настройки шейдера.

Таблица 12.9. *Работа шейдера Offset Texture 2D Scale*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	(s_0, t_0, r_0, q_0)	Texture 2D	$(s_0/q_0, t_0/q_0)$	2D DSDT_MAG	(0, 0, 0, 0) *ds*, *dt*, *mag*
1	(s_1, t_1)	$s'_1=s_1+k0\cdot ds+$ $+k2\cdot dt$ $t'_1=t_1+k1\cdot ds+$ $+k3\cdot dt$ $M=k_{scale}\cdot mag+$ $+k_{bias}tex1(s'_1,t'_1)$	$Tex(s'_1, t'_1)$	Любая 2D. RGBA-текстура	(*R*M*, *G*M*, *B*M*, *A*)

Листинг 12.6. Пример настройки для использования шейдера Offset Texture 2D Scale

```
                        // setup stage 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_TEXTURE_2D );
                        // setup stage 1
float    k [4] = { 0.3, 0.5, 0.7, 0.5 };
glActiveTextureARB ( GL_TEXTURE1_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_OFFSET_TEXTURE_2D_SCALE_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV,
                     GL_TEXTURE0_ARB );
glTexEnvfv         ( GL_TEXTURE_SHADER_NV,
                     GL_OFFSET_TEXTURE_MATRIX_NV,   k );
glTexEnvf          ( GL_TEXTURE_SHADER_NV,
                     GL_OFFSET_TEXTURE_SCALE_NV, 0.7 );
glTexEnvf          ( GL_TEXTURE_SHADER_NV, GL_OFFSET_TEXTURE_BIAS_NV,
                     0.2 );
```

Шейдеры, использующие скалярное произведение

Эта группа шейдеров берет результат из одной текстуры, вычисляет два скалярных произведения значения из текстуры и текстурных координат и использует полученный результат для доступа к еще одной текстуре.

Скалярное произведение (dot product)

Шейдер просто вычисляет скалярное произведение RGB-значения предыдущего шага и текстурных координат для данного текстурного блока. При этом в качестве текстуры кроме обычных беззнаковых форматов `GL_RGB` и `GL_RGBA` (с компонентами, принимающими значения из отрезка [0, 1]) также допустимы их знаковые аналоги `GL_SIGNED_RGB_NV` и `GL_SIGNED_RGBA_NV` (компоненты которых принимают значения из отрезка [−1, 1]).

Одним из недостатков обычных (с 8-битовыми компонентами) текстур для вычисления скалярных произведений является их невысокая точность — всего 8 бит на компоненту.

Поэтому для повышения точности при вычислении скалярного произведения существуют два новых формата текстур: `GL_SIGNED_HILO_NV` и `GL_UNSIGNED_HILO_NV`. Для каждого из них тексел состоит из двух 16-битовых компонент.

Для формата `GL_SIGNED_HILO_NV` обе компоненты (*hi*, *lo*) считаются принимающими значение из отрезка [−1, 1] (после масштабирования, конечно). Для вычисления скалярного произведения по этим двум компонентам строится трехмерный вектор:

$$\left(hi, lo, \sqrt{1 - hi^2 - lo^2}\right).$$

Для формата `GL_UNSIGNED_HILO_NV` обе 16-битовые компоненты считаются принимающими значения из отрезка [0,1]. Для вычисления скалярного произведения по этим компонентам строится вектор:

$$(hi, lo, 1).$$

Эти форматы позволяют задавать карты нормалей с гораздо более высокой точностью.

Для беззнаковых RGB(A) текстур можно задать преобразование их в отрезок [−1, 1] или же оставить в исходном виде.

Это осуществляется командой

```
glTexEnvi ( GL_TEXTURE_SHADER_NV,
            GL_RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV, param );
```

В случае, когда `param` принимает значение `GL_EXPAND_NORMAL_NV`, происходит преобразование компонент в отрезок [−1, 1] аналогично тому, как это делается в расширении register combiners. Если значение этого параметра равно `GL_UNSIGNED_IDENTITY_NV`, компоненты в скалярном произведении не преобразуются.

Шейдер GL_DOT_PRODUCT_NV предназначен только для использования на промежуточных шагах, при этом не происходит обращения ни к одной текстуре.

Листинг 12.7 иллюстрирует пример настройки этого шейдера.

Листинг 12.7. Пример настройки для использования шейдера Dot Product

```
                        // setup stage 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_TEXTURE_2D );
                        // setup stage 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV,
                     GL_TEXTURE0_ARB );
```

Этот шейдер обычно является промежуточным для рассматриваемых далее шейдеров.

Двумерное текстурирование (Dot Product Texture 2D)

Данный шейдер вычисляет два скалярных произведения и по ним обращается к текстуре. Предыдущим шагом должно быть скалярное произведение (GL_DOT_PRODUCT_NV). Работа шейдера поясняется табл. 12.10 и рис. 12.5. Пример его настройки иллюстрирует листинг 12.8.

Таблица 12.10. *Работа шейдера Dot Product Texture 2D*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	Зависит от приложения	Зависит от текстуры	Зависит от текстуры	Любая RGB(A)	(r_0, g_0, b_0)
1	(s_1, t_1, r_1)	$u_x = ((r_0, g_0, b_0), (s_1, t_1, r_1))$	Нет	Нет	(0, 0, 0, 0)
2	(s_2, t_2, r_2)	$u_y = ((r_0, g_0, b_0), (s_2, t_2, r_2))$	$tex(u_x, u_y)$	2D RGBA	$tex2D(u_x, u_y)$

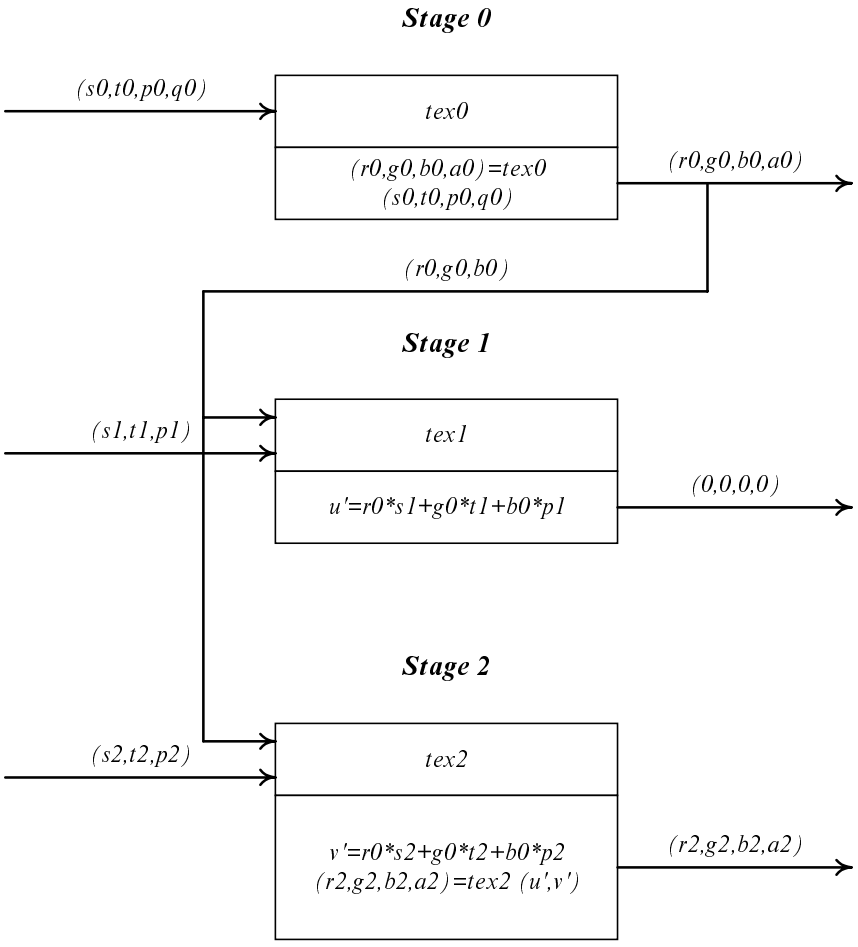


Рис. 12.5. Работа шейдера Dot Product Texture 2D

Листинг 12.8. Пример настроек для использования шейдера Dot Product Texture 2D

```
                        // setup stage 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_TEXTURE_2D );
                        // setup stage 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
```

```
glTexEnvi           ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                      GL_DOT_PRODUCT_NV );
glTexEnvi           ( GL_TEXTURE_SHADER_NV,
                      GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
                          // setup stage 2
glActiveTextureARB  ( GL_TEXTURE2_ARB );
glTexEnvi           ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                      GL_DOT_PRODUCT_TEXTURE_2D_NV );
glTexEnvi           ( GL_TEXTURE_SHADER_NV,
                      GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
```

Фактически данный шейдер вычисляет скалярное произведение текстурных координат со значением текстуры с предыдущего блока для индексации двумерной текстуры.

Двумерное прямоугольное текстурирование (Dot Product Rectange)

Полностью аналогичен предыдущему шейдеру, но на последнем шаге текстура имеет тип `NV_texture_rectange`.

Кубическое текстурирование (Dot Product Texture Cube Map)

Данный шейдер аналогичен двумерному текстурированию со скалярным произведением (Dot Product Texture 2D), но здесь вводятся два промежуточных скалярных произведения, еще одно вычисляется на текущем шаге и эти три полученных значения используются для доступа к кубической текстурной карте. Работа шейдера поясняется табл. 12.11 и рис. 12.6. Пример настроек — листингом 12.9.

Таблица 12.11. *Работа шейдера Dot Product Texture Cube Map*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	Зависит от приложения	Зависит от текстуры	Зависит от текстуры	Любая RGB(A)	(r_0, g_0, b_0)
1	(s_1, t_1, r_1)	$u_x = ((r_0, g_0, b_0), (s_1, t_1, r_1))$	Нет	Нет	(0, 0, 0, 0)

Таблица 12.11 *(окончание)*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
2	(s_2, t_2, r_2)	$u_y = ((r_0, g_0, b_0), (s_2, t_2, r_2))$	Нет	Нет	(0, 0, 0, 0)
3	(s_3, t_3, r_3)	$u_z = ((r_0, g_0, b_0), (s_3, t_3, r_3))$	$u=(u_x, u_y, u_z)$	Cube Map RGBA	$cubeMap(u_x, u_y, u_z)$

Часто в качестве текстурных координат на текущем шаге служат очередные компоненты базиса касательного пространства (t, b, n). Сначала (t_x, b_x, n_x), потом (t_y, b_y, n_y) и наконец (t_z, b_z, n_z).

Листинг 12.9. Пример настроек для использования шейдера Dot Product Cube Map

```
                    // setup stage 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_TEXTURE_2D );
                         // setup stage 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
                         // setup stage 2
glActiveTextureARB ( GL_TEXTURE2_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
                         // setup stage 3
glActiveTextureARB ( GL_TEXTURE3_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_TEXTURE_CUBE_MAP_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
```

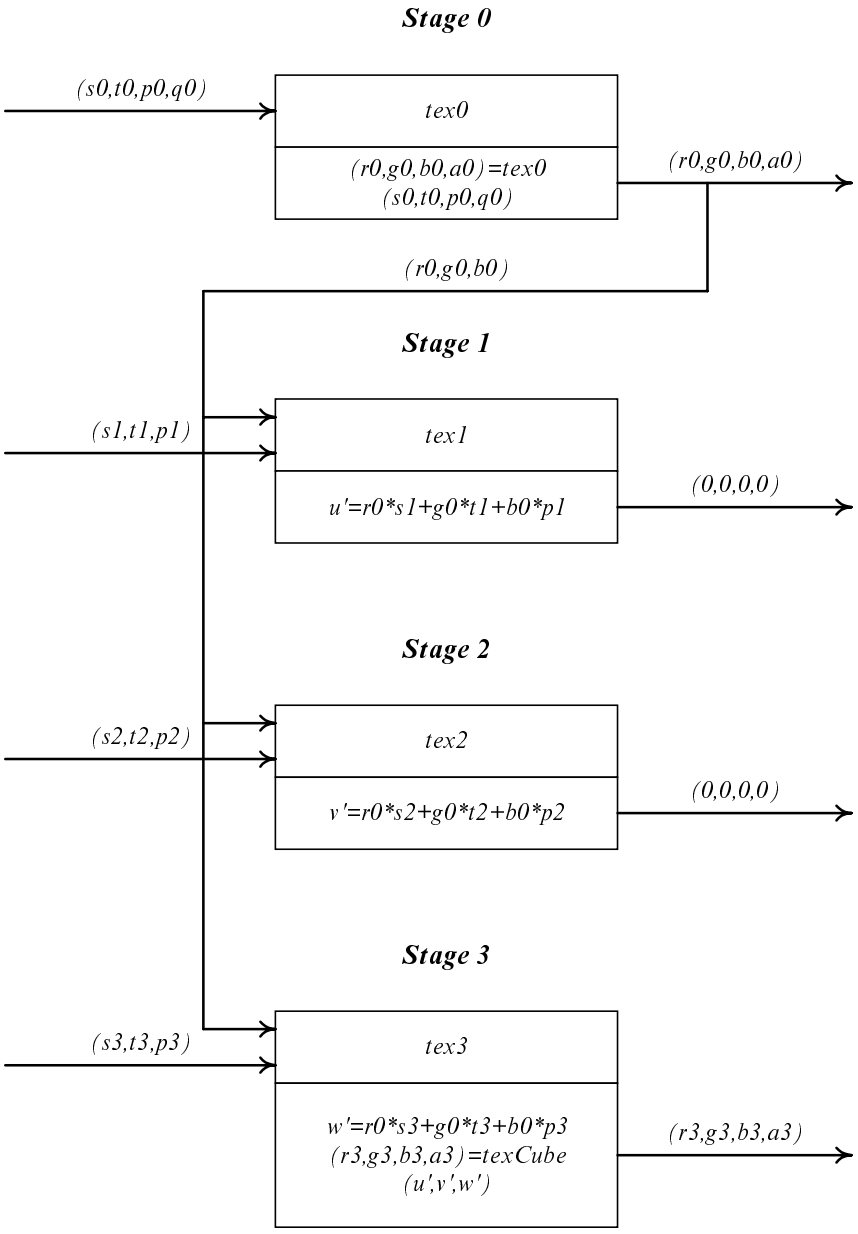


Рис. 12.6. Работа шейдера Dot Product Cube Map

Отражение применением кубической карты с постоянным положением наблюдателя (Dot Product Constant Eye Reflect Cube Map)

В предыдущих главах мы уже сталкивались с имитацией отражения окружающей среды (*environment mapping*) объектом при помощи специальной текстуры. Очень удобной для этого является кубическая карта. Однако стандартный способ моделирования заключается в вычислении отраженного вектора для каждой вершины и его интерполяции вдоль всей грани, т. е. носит повершинный характер. В случае попиксельного отражения (с картами нормалей) необходим способ вычисления отраженного вектора в каждой точке.

Рассмотрим подробнее, каким образом происходит вычисление отраженного вектора, и как это может быть реализовано попиксельно.

Пусть в точке P на поверхности объекта (рис. 12.5) задан единичный вектор v направления на наблюдателя и нормали n.

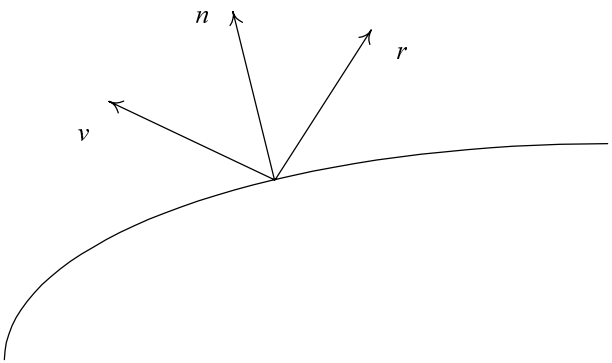


Рис. 12.7. Вычисление отраженного вектора

Тогда отраженный вектор r (как легко убедиться, он также будет единичным) будет задаваться следующей формулой:

$$r = 2\, n(e, n) - e.$$

Если мы хотим для каждого пиксела построить точный отраженный вектор с помощью карты нормалей, то и решать это уравнение необходимо для каждого пиксела объекта.

Карта нормалей задается в касательном пространстве, следовательно в него нужно перевести все векторы. Для этого, как мы уже знаем, достаточно вы-

числить три скалярных произведения. Однако мы уже рассматривали ряд шейдеров, попиксельно вычисляющих скалярные произведения для получения доступа к текстуре.

Шейдер Dot Product Constant Eye Reflect Cube Map, как и предыдущий, формирует три скалярных произведения для получения трехмерного вектора, но для доступа к кубической карте служит не сам полученный вектор u, а отраженный r, вычисляемый по следующей формуле:

$$r = \frac{2(u,\ e)}{(e,\ e)} - e.$$

Здесь вектор e — это положение наблюдателя, которое считается постоянным и задается отдельной командой. Как легко заметить, единственным отличием этой формулы от рассмотренной ранее является поддержка вектора направления на наблюдателя e, который не является единичным.

Вектор u — нормаль в касательном пространстве. Для получения правильного отражения необходимо перевести его из касательного пространства (в котором n задано в карте нормалей) в пространство кубической карты (обычно оно совпадает с пространством наблюдателя). Такой перевод осуществляется при помощи умножения его на матрицу преобразования $\boldsymbol{T}$.

$$u' = \boldsymbol{T}u.$$

В качестве $\boldsymbol{T}$ выступает произведение двух матриц: верхней левой 3×3 подматрицы матрицы *modelview*, инвертированной и транспонированной (ее мы будем далее обозначать как $\boldsymbol{M}_{3x3}^{-T}$) и матрицы, составленной из базисных векторов касательного пространства $\boldsymbol{S} = (t, b, n)$:

$$\boldsymbol{T} = \boldsymbol{M}_{3x3}^{-T} \times \boldsymbol{S}\ .$$

Строки этой матрицы являются первыми тремя компонентами текстурных координат для каждого текстурного блока:

$$(s_1,\ t_1,\ r_1)=(T_{00},\ T_{01},\ T_{02}),$$
$$(s_2,\ t_2,\ r_2)=(T_{10},\ T_{11},\ T_{12}),$$
$$(s_3,\ t_3,\ r_3)=(T_{20},\ T_{21},\ T_{22}).$$

В этом случае предыдущие три шага просто переведут вектор нормали (полученный из текстуры) из касательного пространства в пространство положения наблюдателя.

После этого по приведенной выше формуле находится отраженный вектор и далее он используется для доступа к кубической карте отражения. Работа шейдера иллюстрируется табл. 12.12, а пример настройки — листингом 12.10.

Таблица 12.12. *Работа шейдера Dot Product Constant Eye Reflect Cube Map*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	Зависит от приложения	Зависит от текстуры	Зависит от текстуры	Любая RGB(A)	(r_0, g_0, b_0)
1	(s_1, t_1, r_1)	$u_x = ((r_0, g_0, b_0), (s_1, t_1, r_1))$	Нет	Нет	(0, 0, 0, 0)
2	(s_2, t_2, r_2)	$u_y = ((r_0, g_0, b_0), (s_2, t_2, r_2))$	Нет	Нет	(0, 0, 0, 0)
3	(s_3, t_3, r_3)	$u_z = ((r_0, g_0, b_0), (s_3, t_3, r_3))$ $u = (u_x, u_y, u_z)$ $r = 2u(u, e)/(u, u) - e$	(r_x, r_y, r_z)	Cube Map RGBA	$cubeMap(r_x, r_y, r_z)$

Листинг 12.10. Пример настроек для использования шейдера Dot Product Constant Eye Reflect Cube Map

```
                        // setup stage 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                GL_TEXTURE_2D );
                        // setup stage 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
                        // setup stage 2
glActiveTextureARB ( GL_TEXTURE2_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
                        // setup stage 3
float    eye [3] = { -5, 0, 0 };
glActiveTextureARB ( GL_TEXTURE3_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                         GL_DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV );
```

```
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
glTexEnvfv         ( GL_TEXTURE_SHADER_NV, GL_CONSTANT_EYE_NV, eye );
```

Отражение с использованием кубической карты (Dot Product Reflect Cube Map)

Этот шейдер аналогичен предыдущему, но здесь координаты наблюдателя передаются в текстурной координате q. Работа шейдера поясняется табл. 12.13, а пример настроек содержится в листинге 12.11.

Таблица 12.13. *Работа шейдера Dot Product Reflect Cube Map*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	Зависит от приложения	Зависит от текстуры	Зависит от текстуры	Любая RGB(A)	(r_0, g_0, b_0)
1	(s_1, t_1, r_1, q_1)	$u_x = ((r_0, g_0, b_0), (s_1, t_1, r_1))$	Нет	Нет	(0, 0, 0, 0)
2	(s_2, t_2, r_2, q_2)	$u_y = ((r_0, g_0, b_0), (s_2, t_2, r_2))$	Нет	Нет	(0, 0, 0, 0)
3	(s_3, t_3, r_3, q_3)	$u_z = ((r_0, g_0, b_0), (s_3, t_3, r_3))$ $u = (u_x, u_y, u_z)$ $e = (q_1, q_2, q_3)$ $r = 2u(u, e)/(u, u) - e$	(r_x, r_y, r_z)	Cube Map RGBA	*CubeMap*(r_x, r_y, r_z)

Листинг 12.11. Настройка для шейдера Dot Product Reflect Cube Map

```
                        // setup stage 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_TEXTURE_2D );
                        // setup stage 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
                        // setup stage 2
```

```
glActiveTextureARB ( GL_TEXTURE2_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
                          // setup stage 3
glActiveTextureARB ( GL_TEXTURE3_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                         GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
```

Рассмотрим теперь, как можно реализовать пример тора, использующего карту нормалей для отражения окружения, задаваемого кубической текстурной картой.

Это можно сделать с помощью шейдера `GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV`. В вычислении участвуют все четыре шага (шейдера) текстурирования (*texturing stages*):

1. `GL_TEXTURE_2D`. Текстурой является карта нормалей.
2. `GL_DOT_PRODUCT_NV`.
3. `GL_DOT_PRODUCT_NV`.
4. `GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV`. Текстурой является кубическая карта отражения.

При этом вся информация, необходимая для вычисления отраженного вектора для каждого фрагмента, передается в текстурных координатах первых трех текстурных шагов (s_i, t_i, r_i, q_i).

В координатах q_i передаются координаты положения наблюдателя (*eye*):

$$eye = (q_0, q_1, q_2).$$

Мы считаем, что пространство наблюдателя (*eye-space*) совпадает с пространством, в котором задана кубическая карта отражения (*cubic environment map*), т. е. оси этой системы координат параллельны ребрам куба, на грани которого накладывается кубическая текстура (сам наблюдатель при этом находится в центре куба).

Шейдер `GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV` по преобразованной при помощи матрицы $\boldsymbol{T}$ нормали n' (обратите внимание, что преобразованная нор-

маль может уже не быть единичной) и положению наблюдателя *eye* для каждого фрагмента вычисляет отраженный вектор:

$$r = 2n'(n', eye)/(n', n') - eye.$$

Далее вектор *r* используется для доступа к кубической карте отражения.

Рассмотрим теперь практическую реализацию этого подхода — построение изображения тора, отражающего окружающую среду, на поверхность которого наложена карта нормалей.

За основу класса представления тора можно взять класс `Torus`, встречавшийся нам при вычислении попиксельного диффузного освещения (*см. главу 8*), добавив в класс `Vertex` метод, служащий для получения матрицы ***T***:

```
struct    Vertex
{
    Vector3D    pos;                    // position of vertex
    Vector2D    tex;                    // texture coordinates
    Vector3D    n;                      // unit normal
    Vector3D    t, b;                   // tangent and binormal
                                        // compute transform matrix for
                                        // reflective bump mapping
    void    buildTransformMatrix ( const Matrix3D& mInvT,
                                   Matrix3D& t ) const;
};
```

Входным параметром для метода `buildTransformMatrix` служит инвертированная и транспонированная верхняя левая подматрица `modelview` матрицы (поскольку она одинакова для всех вершин). Выходным параметром является матрица преобразования ***T***.

```
void    Vertex :: buildTransformMatrix ( const Matrix3D& mInvT,
                                         Matrix3D& tr ) const
{
    Matrix3D s ( t, b, n );
    tr = mInvT * s;
}
```

Листинг 12.12 содержит код процедуры вывода тора.

Листинг 12.12. Процедура вывода тора

```
void   Torus :: draw ( const Vector3D& eye )
{
                                        // get modelview matrix from OpenGL
```

```
float  mGl [16];
glGetFloatv ( GL_MODELVIEW_MATRIX, mGl );
                                // now extract 3*3 submatrix from it,
                                // remember that OpenGL
                                // matrices are column-ordered
Matrix3D  mv;
mv [0][0] = mGl [0];
mv [0][1] = mGl [4];
mv [0][2] = mGl [8];
mv [1][0] = mGl [1];
mv [1][1] = mGl [5];
mv [1][2] = mGl [9];
mv [2][0] = mGl [2];
mv [2][1] = mGl [6];
mv [2][2] = mGl [10];
                                // invert and transpose matrix
mv.invert    ();
mv.transpose ();
Matrix3D  t;
glBegin ( GL_TRIANGLES );
for ( int i = 0; i < numFaces; i++ )
    for ( int j = 0; j < 3; j++ )
    {
        Vertex& v = vertices [faces [i].index [j]];
        v.buildTransformMatrix ( mv, t );
        Vector4D  t1 ( t [0][0], t [0][1], t [0][2], eye.x );
        Vector4D  t2 ( t [1][0], t [1][1], t [1][2], eye.y );
        Vector4D  t3 ( t [2][0], t [2][1], t [2][2], eye.z );
        glMultiTexCoord2fv ( GL_TEXTURE0_ARB, v.tex );
        glMultiTexCoord3fv ( GL_TEXTURE1_ARB, t1    );
        glMultiTexCoord2fv ( GL_TEXTURE2_ARB, t2    );
        glMultiTexCoord3fv ( GL_TEXTURE3_ARB, t3    );
        glVertex3fv        ( v.pos                  );
    }
glEnd  ();
```

Сама программа приведена в листинге 12.13.

Листинг 12.13. Реализация точного отражения с использованием карты нормалей

```
//
// Sample to to per-pixel true reflective bumpmapping in OpenGL
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Torus.h"
Vector3D    eye   ( 8, 0, 0 );        // camera position
unsigned    envMap;                   // environment cubemap id
unsigned    bumpMap;                  // normal map
float       angle = 0;
Torus       torus ( 1, 2, 30, 30 );
Vector3D    rot ( 0, 0, 0 );
int         mouseOldX = 0;
int         mouseOldY = 0;
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",
};
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 );
    glEnable ( GL_TEXTURE_SHADER_NV );
    glEnable ( GL_REGISTER_COMBINERS_NV );
                    // setup texture stages
                    // stage 0:
                    // bind bump (normal) map to
                    // texture unit 0
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, bumpMap );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                         GL_TEXTURE_2D );
                    // stage 1:
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                         GL_DOT_PRODUCT_NV );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                         GL_PREVIOUS_TEXTURE_INPUT_NV,
                         GL_TEXTURE0_ARB );
                    // stage 2:
    glActiveTextureARB ( GL_TEXTURE2_ARB );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                         GL_DOT_PRODUCT_NV );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                         GL_PREVIOUS_TEXTURE_INPUT_NV,
                         GL_TEXTURE0_ARB );
                    // stage 3:
                    // bind environment cube map to
                    // texture unit 3
    glActiveTextureARB ( GL_TEXTURE3_ARB );
    glEnable           ( GL_TEXTURE_CUBE_MAP_ARB );
    glBindTexture      ( GL_TEXTURE_CUBE_MAP_ARB, envMap );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                         GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV );
```

```
    glTexEnvi           ( GL_TEXTURE_SHADER_NV,
                          GL_PREVIOUS_TEXTURE_INPUT_NV,
                          GL_TEXTURE0_ARB );
                    // setup register combiners
    glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_TEXTURE3_ARB,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // B.rgb = 1
    glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_ZERO,
                             GL_UNSIGNED_INVERT_NV, GL_RGB );
                    // C = 0
    glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // D = 0
    glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();
    glRotatef    ( rot.x, 1, 0, 0 );
    glRotatef    ( rot.y, 0, 1, 0 );
    glRotatef    ( rot.z, 0, 0, 1 );
    torus.draw ( eye );
    glPopMatrix     ();
    glutSwapBuffers ();
}
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 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 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  ( 500, 500 );
                        // create window
    int    win = glutCreateWindow ( "OpenGL true per-pixel reflective
bumpmapping" );
```

```
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutMouseFunc    ( mouse   );
    glutMotionFunc   ( motion  );
    init           ();
    initExtensions ();
    printfInfo     ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_NV_register_combiners" ) )
    {
        printf ( "NV_register_combiners NOT supported" );

        return 2;
    }
    if ( !isExtensionSupported ( "GL_NV_texture_shader" ) )
    {
        printf ( "GL_NV_texture_shader NOT supported" );
        return 3;
    }
    bumpMap = createNormalMap              ( false,
                      "../Textures/Bumpmaps/bump.tga" );
    envMap  = createCubeMap                ( true, faces );
    glutMainLoop ();
    return 0;
}
```

На рис. 12.8 приводится построенное этой программой изображение отражающего тора с наложенной на него картой нормалей.

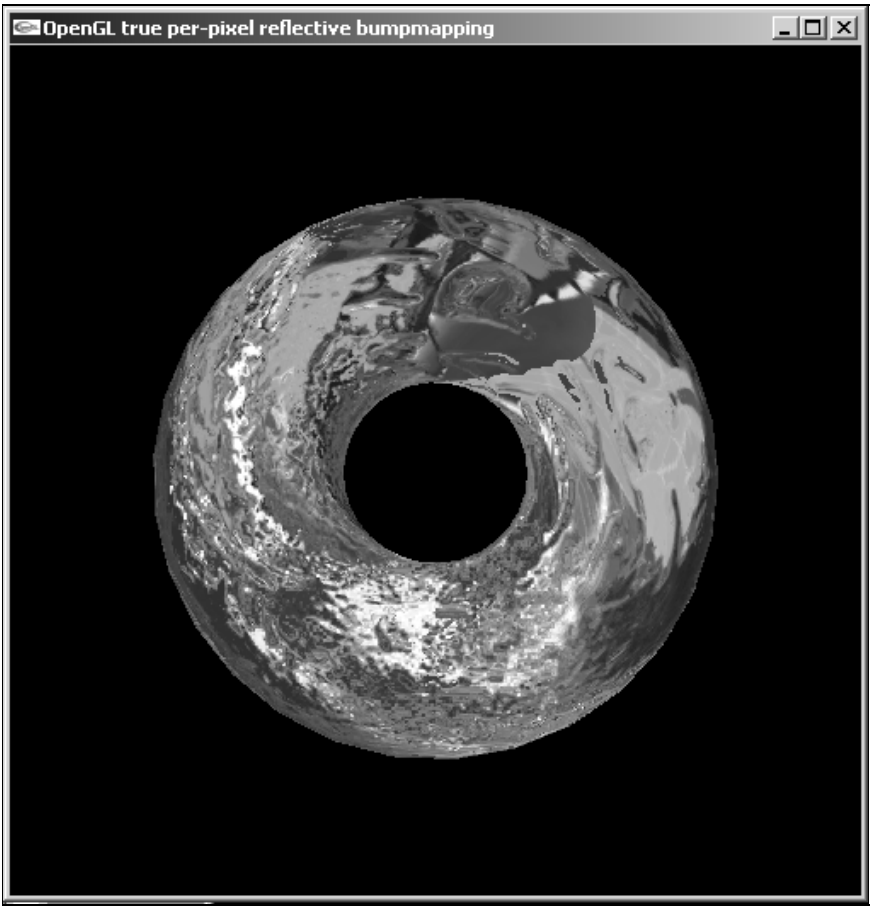


Рис. 12.8. Изображение тора с картой нормалей с environment mapping

Использование скалярного произведения для обращения сразу к двум кубическим картам (Dot Product Diffuse Cube Map)

Предыдущие расширения для получения отраженного вектора *r* вычисляют вектор *u*, который фактически является преобразованным в касательное пространство вектором нормали (возможно, не единичным).

Данный шейдер позволяет использовать этот промежуточный вектор для обращения к еще одной текстурной карте, таким образом на последнем шаге происходит обращение к кубической карте по отраженному вектору *r*, а на предпоследнем — к другой кубической карте по вектору *u*.

Если на предыдущем шаге разместить нормирующую кубическую карту, то на выходе этого шага мы получим значение единичного вектора нормали, преобразованное в касательное пространство. Этот метод иллюстрируется табл. 12.14, а пример настройки — листингом 12.14.

Таблица 12.14. *Работа шейдера Dot Product Diffuse Cube Map*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	Зависит от приложения	Зависит о т текстуры	Зависит от текстуры	Любая RGB(A)	(r_0, g_0, b_0)
1	(s_1, t_1, r_1, q_1)	$u_x = ((r_0, g_0, b_0), (s_1, t_1, r_1))$	Нет	Нет	(0, 0, 0, 0)
2	(s_2, t_2, r_2, q_2)	$u_y = ((r_0, g_0, b_0), (s_2, t_2, r_2))$ $u_z = ((r_0, g_0, b_0), (s_3, t_3, r_3))$	(u_x, u_y, u_z)	Cube Map RGBA	$CubeMap(u_x, u_y, u_z)$
3	(s_3, t_3, r_3, q_3)	$u = (u_x, u_y, u_z)$ $e = (q_1, q_2, q_3)$ $r = 2u(u, e)/(u, u) - e$	(r_x, r_y, r_z)	Cube Map RGBA	$CubeMap(r_x, r_y, r_z)$

Листинг 12.14. Настройка для использования шейдера Dot Product Diffuse Cube Map

```
                        // setup stage 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_TEXTURE_2D );
                        // setup stage 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
                        // setup stage 2
glActiveTextureARB ( GL_TEXTURE2_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
```

```
                        // setup stage 3
glActiveTextureARB ( GL_TEXTURE3_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
```

Замена глубины (Dot Product Depth Replace)

Этот шейдер позволяет выводить грань, параллельную экрану, с заданной картой глубин (*depth sprite*). Для получения наибольшей точности лучше всего выбрать беззнаковую текстуру HILO.

По текстуре на текущем шаге вычисляются два скалярных произведения, отношение которых и дает значение глубины. Если это значение лежит в пределах $[z_{near}, z_{far}]$, то оно записывается в буфер глубины. В противном случае фрагмент отбрасывается.

Предыдущим шагом должно быть скалярное произведение.

Работа шейдера иллюстрируется рис. 12.9 и табл. 12.15, а пример настроек — листингом 12.15.

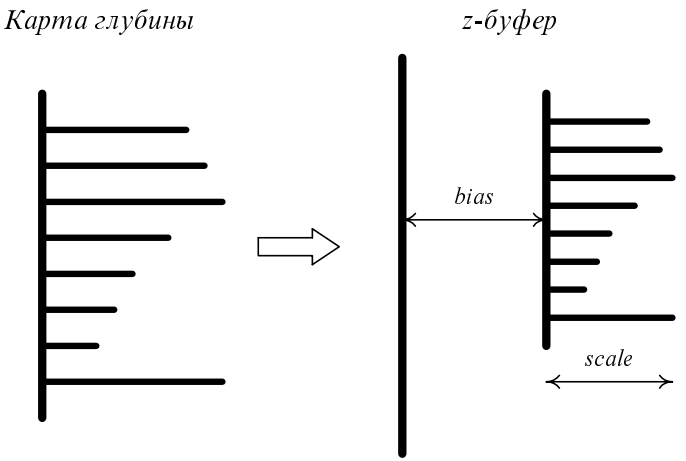


Рис. 12.9. Работа шейдера Dot Product Depth Replace

Таблица 12.15. *Работа шейдера Dot Product Depth Replace*

Шаг	Текстурные координаты	Шейдер	Обращение к текстуре	Формат текстуры	Выходное значение
0	(s_0, t_0, r_0, q_0)	Texture 2D	$(s_0/q_0, t_0/q_0)$	Беззнаковая HILO	(0, 0, 0, 0)
1	$(z_{scale}, z_{scale}/2^{16}, z_{bias})$	$Z = (z_{scale}, z_{scale}/2^{16}, z_{bias})$, $(H, L, 1)$	Нет	Нет	(0, 0, 0, 0)
2	$(w_{scale}, w_{scale}/2^{16}, w_{bias})$	$W = (w_{scale}, w_{scale}/2^{16}, w_{bias})$, $(H, L, 1)$ $z_w = Z/W$	Нет	Нет	(0, 0, 0, 0)

Листинг 12.15. Настройка для использования шейдера Dot Product Depth Replace

```
                        // setup stage 0
glActiveTextureARB ( GL_TEXTURE0_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_TEXTURE_2D );
                          // setup stage 1
glActiveTextureARB ( GL_TEXTURE1_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
                          // setup stage 2
glActiveTextureARB ( GL_TEXTURE2_ARB );
glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                     GL_DOT_PRODUCT_DEPTH_REPLACE_NV );
glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                     GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB );
```

Обратите внимание, что в ряде случаев при неправильном задании данных шаг текстурирования может находиться в ошибочном (*inconsistent*) состоянии, при этом для него устанавливается операция `GL_NONE`.

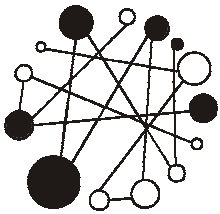
Узнать состояние шага можно при помощи следующего фрагмента кода:

```
GLint consistent;
glActiveTextureARB ( stage_to_check );
glGetTexEnviv      ( GL_TEXTURE_SHADER_NV, GL_SHADER_CONSISTENT_NV,
                     &consistent );
```

Механизм шейдеров, вводимых расширением NV_texture_shader, позволяет заметно расширить возможности графического ускорителя по программированию сложных эффектов.

Возможность организации шагов текстурирования, направляющих выход одного шага на вход другого, обеспечила в свое время качественный скачок при программировании графических ускорителей. Существуют также расширения NV_texture_shader2 и NV_texture_shader3, добавляющие новые типы шейдеров. Большие возможности открыли расширения ARB_vertex_program и ARB_fragment_program, а также язык GLSL для написания собственных шейдеров.

Глава 13



Расширения для динамического определения видимости в сложных сценах

Поскольку все современные графические ускорители поддерживают метод z-буфера удаления невидимых поверхностей (пикселов) на аппаратном уровне, то для вывода даже довольно сложной сцены достаточно передать OpenGL все грани, и в результате мы получим корректное изображение.

Однако на практике подобный подход обладает весьма серьезным недостатком. Для действительно больших и сложных сцен лишь очень немногие грани (объекты) сцены оказываются реально видимы наблюдателю. Все остальные не вносят никакого вклада в строящееся изображение, если их не выводить, то ничего не изменится.

Таким образом, все эти объекты (во много раз превышающие по количеству число реально видимых) просто впустую потребляют весьма значительные ресурсы графического ускорителя. Если бы удалось такие невидимые объекты быстро определить и сразу же отбросить (т. е. не выводить), то мы получили бы значительное увеличение быстродействия.

Отбрасывая невидимые объекты, мы выигрываем сразу на экономии двух важных типов ресурсов: вершин и пикселов. Так, один объект с очень большим числом вершин, даже если на экране он занимает всего несколько пикселов, потребует значительных затрат именно на обработку вершин, в то время как обработка пикселов окажется незначительной.

С другой стороны, одна большая грань, закрывающая практически весь экран, может потребовать больших затрат именно на обработку накрываемых ею пикселов, а обработка вершин будет весьма простой.

Для того чтобы выяснить, какие грани (объекты) следует отбрасывать сначала, надо разобраться с понятием видимости (или невидимости) отдельных объектов. Рассмотрим, за счет чего объект сцены может оказаться невидимым наблюдателю.

Возможны две различные ситуации:

1. Объект просто не попадает в область видимости наблюдателя (*viewing frustrum*) (рис. 13.1).
2. Объект, хотя и попадает в область видимости, но оказывается закрытым от наблюдателя другим (другими) объектом (объектами) (рис. 13.2).

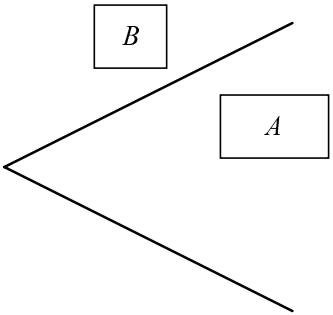


Рис. 13.1. Объект не попал в область видимости наблюдателя

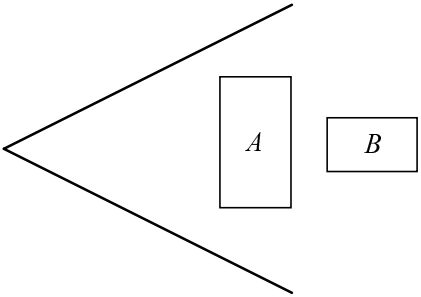


Рис. 13.2. Объект *B* закрыт от наблюдателя объектом *A*

Так, объект *B* на рис. 13.1 просто не попадает в пирамиду видимости наблюдателя и поэтому не может быть виден. На рис. 13.2 изображены два объекта (*A* и *B*), каждый из которых попадает в пирамиду видимости, но при этом объект *B* закрыт от наблюдателя объектом *A* (мы считаем все объекты непрозрачными).

На рис. 13.3 представлен более сложный случай. Объект *C* закрыт от наблюдателя сразу двумя объектами — *A* и *B*. Однако при этом ни один из них в одиночку не закрывает полностью объект *C* от наблюдателя. Это явление называется сложным затенением *occlusion fusion*.

Невидимость второго типа часто приводит к такому крайне нежелательному явлению, как перезапись (*overdraw*). Под этим термином подразумевается

многократная запись (и вычисление соответствующих цветов) в одни и те же пикселы.

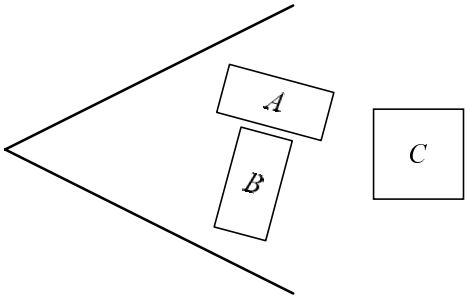


Рис. 13.3. Сложное затенение

При использовании сложных вычислительных моделей для определения цвета пиксела большой *overdraw* приводит к весьма заметному падению быстродействия. В частности, именно поэтому рекомендуется выводить объекты в порядке удаления от наблюдателя (*front-to-back*), поскольку это позволяет современным графическим ускорителям в ряде случаев сразу определять, что данный пиксел не проходит тест глубины, и избежать вычисления его цвета.

Проверка объекта на попадание внутрь области видимости алгоритмически довольно проста. Поскольку область видимости представляет собой пересечение нескольких полупространств, то достаточно проверить на попадание (хотя бы частичное) объекта в каждое из них (рис. 13.4).

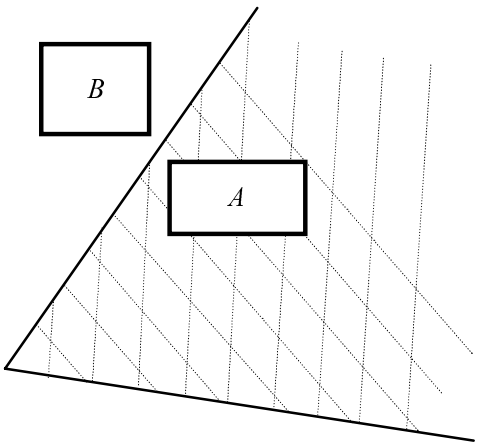


Рис. 13.4. Проверка на попадание объекта в пирамиду видимости

Однако, несмотря на свою алгоритмическую простоту, для объектов, состоящих из многих тысяч граней, проверка "в лоб" может потребовать значительных затрат ресурсов CPU. Поэтому на практике для этого используются так называемые ограничивающие тела (*bounding volume*) — достаточно простые геометрические фигуры, полностью содержащие искомый объект внутри себя (рис. 13.5).

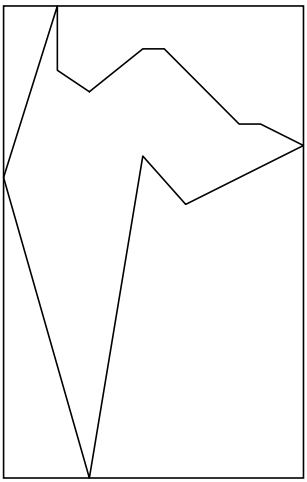


Рис. 13.5. Пример ограничивающего тела

Если ограничивающее тело не попало в какую-то область пространства, то и содержащийся в ней объект также не может попадать в эту область. Обратное в общем случае неверно (рис. 13.6).

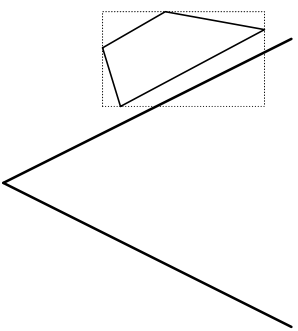


Рис. 13.6. Ограничивающее тело попадает в область видимости, а исходный объект — нет

Данное обстоятельство позволяет вместо проверки исходного сложного объекта выполнить проверку его ограничивающего тела. Если оно достаточно простое, то мы получаем возможность отбросить сложный объект путем всего лишь нескольких элементарных тестов.

В случае, когда при помощи ограничивающего тела не удается установить видимость или невидимость объекта, то можно выбрать более точные методы (требующие больших затрат) или принять, что объект виден, и вывести его (а не тратить ресурсы CPU на сложные проверки).

В качестве ограничивающих тел обычно выбираются простые геометрические фигуры — сферы, прямоугольные параллелепипеды и т. п. Очень подходящими для этого являются прямоугольные параллелепипеды с ребрами, параллельными осям координат, называемые *AABB* (*Axis Aligned Bounding Box*).

В отличие от проверки на видимость, определение затенения объекта другими объектами осуществляется гораздо сложнее и требует больших вычислительных ресурсов.

Поэтому было бы очень удобно возложить такую проверку непосредственно на графический ускоритель, где выяснить, проходит ли хотя бы один фрагмент из растрового представления объекта так называемый тест глубины (изменение хотя бы одного пиксела при попытке вывода объекта). Если таких фрагментов нет, то это значит, что данный объект полностью невидим (рис. 13.7).

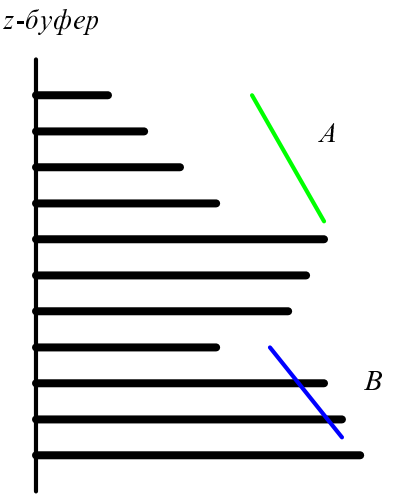


Рис. 13.7. Проверка объекта относительно буфера глубины

Именно такую возможность и предоставляет расширение HP_occlusion_test. Оно позволяет проверить, прошел хотя бы один фрагмент из растрового представления группы объектов тест глубины.

Для упрощения проверки обычно проверяется не сам исходный объект, а лицевые грани его ограничивающего тела (*AABB*). Поскольку для проверки видимости мы фактически выводим грани *AABB*, то необходимо на время тестов запретить запись во фреймбуфер, буферы глубины и трафарета.

Далее приводится фрагмент кода, иллюстрирующий работу с данным расширением.

```
glDepthMask    ( GL_FALSE );
glColorMask    ( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );
glStencilMask  ( GL_FALSE );
glEnable       ( GL_OCCLUSION_TEXT_HP );
aabb.drawFront ();
glDisable      ( GL_OCCLUSION_TEST_HP );
glDepthMask    ( GL_TRUE );
glColorMask    ( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE  );
glStencilMask  ( GL_TRUE );
glGetBooleanv  ( GL_OCCLUSION_TEST_RESULT_HP, &boxVisible );
```

Если в результате мы получим значение `boxVisible` равное `GL_TRUE`, то это означает, что хотя бы одна грань ограничивающего тела будет видна в случае его вывода. Хотя из этого и не следует видимость самого объекта, его обычно принято полагать видимым и выводить (рис. 13.8).

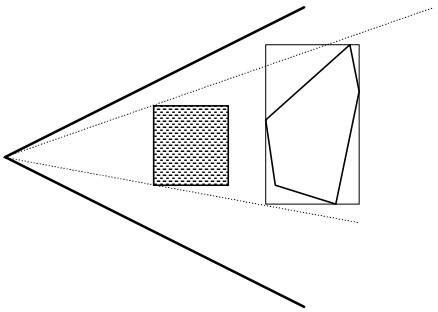


Рис. 13.8. Невидимый объект,
для которого видима грань ограничивающего тела

В противном случае мы полагаем, что интересующий нас объект невидим, и просто пропускаем его.

Однако данное расширение при всей его простоте и привлекательности имеет весьма серьезный недостаток — оно требует явной передачи данных от графического ускорителя к CPU.

Дело в том, что современные графические ускорители ориентированы в основном на передачу данных (вершин, атрибутов, текстур) только в одном направлении: от CPU к графическому ускорителю при активном распараллеливании обработки.

Случаи передачи данных в обратную сторону приводят, как правило, с одной стороны, к ожиданию CPU, пока не будет готов соответствующий результат (это явление получило название *CPU stalling*), а с другой, поскольку во время простоя CPU он не посылает данные графическому ускорителю, происходит простаивание последнего (*GPU starvation*).

Именно к таким ситуациям может привести расширение HP_occlusion_test — вызвать остановку конвейера обработки, пока результат теста не будет передан центральному процессору.

Еще одним недостатком такого метода является то, что результатом его выполнения является булева величина (прошел/не прошел). В то же время в ряде случаев важно знать, для какой части выведенных пикселов тест прошел. Если оказывается, что тест выполнен всего лишь для нескольких пикселов, то в большинстве случаев вывод соответствующего объекта также можно пропустить (или выполнить упрощенную процедуру его вывода).

Всех этих недостатков лишено предложенное компанией NVIDIA расширение GL_NV_occlusion_query. Оно позволяет параллельно выполнять сразу несколько запросов на определение видимости, получать информацию о готовности того или иного запроса и выдавать в результате число пикселов, для которого тест видимости успешно пройден. Для предоставления подобной функциональности это расширение вводит наряду с новыми константами еще и несколько новых функций.

Так как возможно одновременное выполнение сразу нескольких запросов, то для каждого из них необходимо получить уникальный идентификатор (беззнаковое целое число как и для текстур), который используется как для задания самого запроса, так и для получения информации о нем.

Для получения набора идентификаторов служит следующая функция:

```
void glGenOcclusionQueriesNV ( GLsizei n, GLuint * ids );
```

Она получает в качестве входных параметров количество `n` идентификаторов, которые необходимое вернуть, и буфер `ids`, куда они должны быть помещены.

По окончании выполнения запроса (запросов) соответствующие идентификаторы должны быть освобождены при помощи функции

```
void glDeleteOcclusionQueriesNV ( GLsizei n, const GLuint * ids );
```

После того, как идентификаторы запросов были успешно получены, можно начать выполнение самих запросов видимости.

Для этого служат следующие две функции:

```
void glBeginOcclusionQueryNV ( GLuint id );
void glEndOcclusionQueryNV   ();
```

Первая обозначает начало теста видимости с заданным идентификатором `id`. Вторая — конец задания теста видимости.

При выполнении этих тестов, так же как и для рассмотренного ранее расширения HP_occlusion_test, следует запретить запись в буферы глубины, трафарета и во фреймбуфер.

Эти функции позволяют задать сразу несколько тестов видимости и потом опрашивать их текущее состояние и результаты при помощи функции `glGetOcclusionQueryuivNV`:

```
void glGetOcclusionQueryuivNV ( GLuint id, GLenum pname,
                                GLuint * params );
```

Параметр `id` определяет, для какого запроса будет запрашиваться значение, `pname` — запрашиваемую величину, а `params` — адрес переменной, куда необходимо занести результат выполнения запроса. В табл. 13.1 приведен перечень типов запросов этой функции.

Таблица 13.1. *Типы запросов* `glGetOcclusionQueryuivNV`

Величина	Комментарий
`GL_PIXEL_COUNT_AVAILABLE_NV`	Готовы ли результаты соответствующего теста. В случае готовности возвращается значение `GL_TRUE`, в противном случае — `GL_FALSE`
`GL_PIXEL_COUNT_NV`	Возвращает количество пикселов для данного теста, которые успешно прошли тест глубины. Если на момент вызова результаты данного теста еще не готовы, то происходит их ожидание

Наличие запроса `GL_PIXEL_COUNT_AVAILABLE_NV` позволяет приложению не ждать готовности результатов данного теста, а выполнять некоторую полезную работу до тех пор, пока результаты не будут готовы (например, выводить те объекты, видимость которых уже установлена).

Возможна такая крайне нежелательная ситуация, когда выполнение запросов может потребовать много времени для ожидания окончания теста. Для решения проблемы возможны несколько вариантов.

Простейший из них заключается в выводе этих объектов (видимость которых пока еще не определена).

Более интересным и перспективным является подход на основе так называемой покадровой когерентности (*frame coherence*), т. е. том обстоятельстве, что множество видимых объектов обычно очень мало изменяется от кадра к кадру. Это позволяет нам использовать результаты "старых" (для предыдущих кадров) тестов, которые уже готовы к началу текущего кадра.

Правда в этом случае может потребоваться учет объектов, для которых видимость могла измениться по сравнению с предыдущим кадром (быстро двигающиеся объекты).

В тех случаях, когда вывод объекта осуществляется в несколько проходов, очень удобными оказываются запросы видимости. Тогда на первом проходе вместе с выводом самого объекта выполняется запрос на его видимость. Если к окончанию настройки OpenGL для очередного прохода результаты теста видимости уже получены, то в случае невидимости объекта выполнение оставшихся проходов может быть прекращено.

В 2003 г. появилось новое расширение, служащее для определения видимости, — ARB_occlusion_query. За его основу было взято NV_occlusion_query. Ниже приводится краткое описание основных функций данного расширения и способов работы с ним.

Для этого расширения также возможно одновременное выполнение сразу нескольких запросов, при этом каждый из них определяется беззнаковым целочисленным идентификатором, для получения и освобождения которого служат следующие функции:

```
void glGenQueriesARB    ( GLsizei n, GLuint *ids );
void glDeleteQueriesARB ( GLsizei n, const GLuint *ids );
```

Они полностью аналогичны рассмотренным ранее `glGenOcclusionQueriesNV` и `glDeleteOcclusionQueriesNV`.

Для проверки, является ли данное число идентификатором запроса определения видимости, служит функция

```
GLboolean glIsQueryARB ( GLuint id );
```

Для обозначения начала и окончания запроса определения видимости служат функции

```
void glBeginQueryARB ( GLenum target, GLuint id );
void glEndQueryARB   ( GLenum target );
```

В качестве параметра `target` на текущий момент допустима только константа `GL_SAMPLES_PASSED_ARB`. При этом если уже существует запрос с заданным идентификатором, то счетчик числа пройденных пикселов сбрасывается и запрос ставится на выполнение. Идентификатор, равный нулю,

является недопустимым. Учитываются только пикселы, успешно прошедшие как тест глубины, так и тест трафарета (если он разрешен).

Для получения запросов о состоянии теста и поддержке запросов имеется три функции:

```
void glGetQueryObjectivARB  ( GLuint id, GLenum pname, GLint *params);
void glGetQueryObjectuivARB ( GLuint id, GLenum pname, GLuint *params );
void glGetQueryivARB        ( GLenum target, GLenum pname,
                              GLint *params);
```

Функция `glGetQueryivARB` позволяет получать общую информацию о запросе. Параметром `target` здесь, как и ранее, является значение `GL_SAMPLES_PASSED_ARB`. При помощи параметра `pname`, равного `GL_CURRENT_QUERY_ARB`, можно получить идентификатор текущего запроса или ноль при отсутствии активного в данный момент запроса.

При помощи параметра `pname`, равного `GL_QUERY_COUNTER_BITS_ARB`, можно получить количество битов, отводимое под счетчик числа прошедших пикселов. В случае, если полученное значение отлично от нуля, то разрядности счетчика хватит, по крайней мере, для учета двух записей в каждый пиксел.

Для получения состояния готовности данного запроса служит следующий фрагмент кода:

```
glGetQueryObjectuivARB ( id, GL_QUERY_RESULT_AVAILABLE_ARB,
                         &result);
```

Если результаты запроса с заданным идентификатором `id` уже готовы, то в переменную `result` будет записано значение `GL_TRUE`.

Тогда для получения числа пикселов, успешно прошедших тест, можно воспользоваться следующим фрагментом кода:

```
glGetQueryObjectuivARB ( id, GL_QUERY_RESULT_ARB, &count);
```

Наиболее эффективным методом для проверки видимости является не простой перебор ограничивающих тел, а создание из них некоторой структуры разбиения пространства. Далее мы рассмотрим одну из простейших подобных структур — равномерное разбиение всего пространства на набор одинаковых ячеек.

В качестве модельной сцены для проверки отсечения выберем большую комнату, случайным образом заполненную различными объектами (рис. 13.9).

Каждый подобный объект мы будем представлять в виде экземпляра класса, унаследованного от класса `SceneObject`. Листинг 13.1 содержит описание классов объектов сцены.

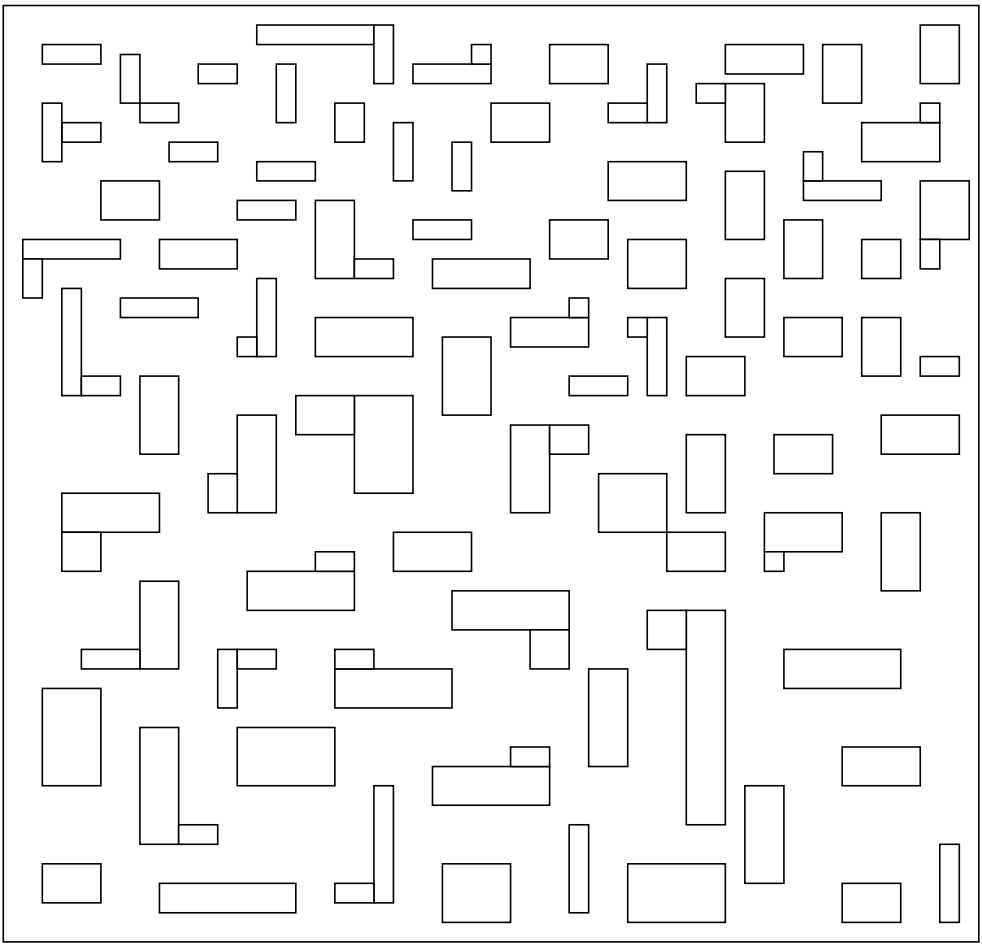


Рис. 13.9. Пример модельной сцены

Листинг 13.1. Описание классов объектов сцены

```
//
// Simple object for occlusion testing
//
#ifndef    __SCENE_OBJECT__
#define    __SCENE_OBJECT__
#include    "Vector3D.h"
#include    "BoundingBox.h"
class    SceneObject
```

```
{
protected:
    Vector3D       pos;
    Vector3D       size;
    unsigned       texture;
    BoundingBox    box;
public:
    int            frame;            // frame where object has
// been rendered
    SceneObject ( const Vector3D& thePos, const Vector3D& theSize,
                    unsigned theTexture ) : box ( thePos,
                                                  thePos + theSize )
    {
        pos     = thePos;
        size    = theSize;
        texture = theTexture;
        frame   = -1;
    }
    const BoundingBox&    getBounds () const
    {
        return box;
    }
    virtual ~SceneObject () {}
    virtual    void    draw () const = 0;
};
class    BoxObject : public SceneObject
{
protected:
    int    type;
public:
    BoxObject ( const Vector3D& thePos, const Vector3D& theSize,
                  unsigned theTexture, int theType ) :
                SceneObject ( thePos, theSize, theTexture )
    {
        type = theType;
    }
    virtual    void    draw () const;
};
```

```
class    TeapotObject : public SceneObject
{
public:
    TeapotObject ( const Vector3D& thePos, const Vector3D& theSize,
                       unsigned theTexture ) :
        SceneObject ( thePos, theSize, theTexture )
    {
    }
    virtual    void    draw () const;
};
#endif
```

Как видно, в качестве объектов здесь будут выступать различные ящики и чайники (последние особенно удобны из-за больших затрат на их вывод). С каждым из подобных объектов связывается ограничивающее тело, описанное вокруг него, используемая текстура и номер кадра, когда объект был в последний раз выведен. Реализация соответствующих классов приводится в листинге 13.2.

Листинг 13.2. Реализация объектов модельной сцены

```
//
// Simple object for occlusion testing
//
#ifdef    _WIN32
    #include    <windows.h>
#endif
#include    <GL/gl.h>
#include    <glut.h>
#include    "SceneObject.h"
#include    "boxes.h"
void    BoxObject :: draw () const
{
    switch ( type )
    {
        case 0:
        case 1:
            drawBox ( pos, size, texture );
            return;
```

```
        case 2:
            drawBox2 ( pos, size, texture );
            return;
        case 3:
            drawBox3 ( pos, size, texture );
            return;
        case 4:
            drawBox4 ( pos, size, texture );
            return;
    }
    return;
}
void    TeapotObject :: draw () const
{
    glMatrixMode    ( GL_MODELVIEW );
    glPushMatrix    ();
    glTranslatef    ( pos.x, pos.y, pos.z );
    glScalef        ( size.x, size.y, size.z );
    glBindTexture   ( GL_TEXTURE_2D, texture );
    glutSolidTeapot ( size.x );
    glPopMatrix     ();
}
```

Простейший вариант вывода подобной сцены — вывод всех объектов. Соответствующая реализация приводится в листинге 13.3.

Листинг 13.3. Простейший способ вывода сцены

```
//
// Sample to view frustum & occlusion culling
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <list>
using namespace std;
#include    "libTexture.h"
#include    "TypeDefs.h"
#include    "Vector3D.h"
```

```
#include    "Vector2D.h"
#include    "boxes.h"
#include    "SceneObject.h"
Vector3D    eye     ( 0, 0, -2 );        // camera position
Vector3D    viewDir ( 1, 0, 0 );        // viewer direction
unsigned    decalMap [3];            // decal (diffuse) texture
unsigned    stoneMap;
int         mouseOldX = 0;
int         mouseOldY = 0;
Vector3D    rot ( 0, 0, 0 );
float       step = 0.003;            // movement rate per second
bool        mousePressed = false;
float       roomSize     = 100;        // room half size
int        frames       = 0;
int        elapsedTime  = 0;
float        fps          = 0;
list <SceneObject *>    objects;        // global list of objects
void    setupCamera ();
inline    float    rnd ()
{
    return (float) rand () / (float)RAND_MAX;
}
void    initObjects ()
{
    for ( int i = 1 - roomSize;  i < roomSize - 1; i += 2 )
        for ( int j = 1 - roomSize; j < roomSize - 1; j += 2 )
        {
            float    x    = i + rnd () * 1.2;
            float    y    = j + rnd () * 1.2;
            float    sx   = 0.7 + 0.6 * rnd ();
            float    sy   = 0.7 + 0.5 * rnd ();
            int      type = 1 + rand () % 4;
            int      tex  = rand () % 3;
            if ( (rand () % 15) == 14 )
                objects.push_back ( new TeapotObject (
                          Vector3D ( x, y, -2 ),
                          Vector3D ( sx*0.5, sy*0.5, (sx + sy) * 0.25 ),
                          decalMap [tex] ) );
```

```
            else
                objects.push_front ( new BoxObject (
                        Vector3D ( x, y, -3 ),
                        Vector3D ( sx, sy, sx + sy ),
                        decalMap [tex], type ) );
        }
}
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 main hall
    drawBox  ( Vector3D ( -roomSize, -roomSize, -3 ),
               Vector3D ( 2*roomSize, 2*roomSize, 7 ),
               stoneMap, false );
                        // draw objects
    for ( list <SceneObject *> :: iterator it = objects.begin ();
            it != objects.end (); ++it )
        (*it) -> draw ();
    frames++;
    if ( (frames % 5) == 0 )        // compute every 5 frames
    {
        int        time2 = glutGet ( GLUT_ELAPSED_TIME );
        char    str [256];
        fps         = (float)frames / (0.001 *
                                       (time2 - elapsedTime));
        elapsedTime = time2;
        frames      = 0;
        sprintf ( str, "OpenGL occlusion test example, "\
                        "FPS : %5.2f", fps );
```

```
        glutSetWindowTitle ( str );
    }
    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, 0.1, 1000 );
    setupCamera    ();
}
void    setupCamera ()
{
    Vector3D    to = eye + viewDir;
    glMatrixMode   ( GL_MODELVIEW );
    glLoadIdentity ();
    gluLookAt      ( eye.x, eye.y, eye.z,    // eye
                     to.x,  to.y,  to.z,     // center
                     0.0, 0.0, 1.0 );        // up
}
void motion ( int x, int y )
{
    static    bool    mouseInited = false;
    if ( !mouseInited )
    {
        mouseInited = true;
        mouseOldX   = x;
        mouseOldY   = y;
        return;
    }
    float    scale = -M_PI / 100;
    viewDir.x = cos ( scale * x );
    viewDir.y = sin ( scale * x );
    viewDir.z = 0;
    setupCamera ();
    glutPostRedisplay ();
}
```

```
void mouse ( int button, int state, int x, int y )
{
    if ( button == GLUT_LEFT_BUTTON )
        mousePressed = (state == GLUT_DOWN);
    motion ( x, y );
}
void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
             exit ( 0 );
}
void    animate ()
{
    static    bool    timeInited = false;
    static    int    lastTime;
    if ( !timeInited )
    {
        lastTime    = glutGet ( GLUT_ELAPSED_TIME );
        timeInited = true;
    }
    if ( mousePressed )
    {
        Vector3D    delta = viewDir * (step * (
                     glutGet ( GLUT_ELAPSED_TIME ) - lastTime)*0.001f);
        Vector3D    e     = eye + delta;
        if ( e.x < roomSize && e.x > -roomSize )
            eye.x = e.x;
        if ( e.y < roomSize && e.y > -roomSize )
            eye.y = e.y;
        setupCamera ();
    }
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                        // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
```

```
    glutInitWindowSize  ( 500, 500 );
                        // create window
    int    win = glutCreateWindow ( "OpenGL occlusion test example" );
                        // register handlers
    glutDisplayFunc       ( display );
    glutReshapeFunc       ( reshape );
    glutKeyboardFunc      ( key     );
    glutMouseFunc         ( mouse   );
    glutMotionFunc        ( motion  );
    glutPassiveMotionFunc ( motion  );
    glutIdleFunc          ( animate );
    init ();
    printfInfo ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    initExtensions ();
    stoneMap     = createTexture2D ( true, "../Textures/block.bmp" );
    decalMap [0] = createTexture2D ( true, "../Textures/oak.bmp" );
    decalMap [1] = createTexture2D ( true,
                                          "../Textures/FieldStone.tga" );
    decalMap [2] = createTexture2D ( true, "../Textures/wood1.bmp" );
    initObjects  ();
    glutMainLoop ();
    return 0;
}
```

Как несложно убедиться, данный пример показывает весьма невысокое быстродействие. Причина этого лежит как в огромном объеме перезаписи (*overdraw*), так и в выводе заведомо не попадающих в область видимости объектов.

Первым шагом оптимизации этого примера будет добавление в него отсечения по области видимости.

Используемая в OpenGL пирамида видимости (рис. 13.10) представляет собой область пространства, ограниченную шестью плоскостями (ближней, дальней, левой, правой, верхней и нижней).

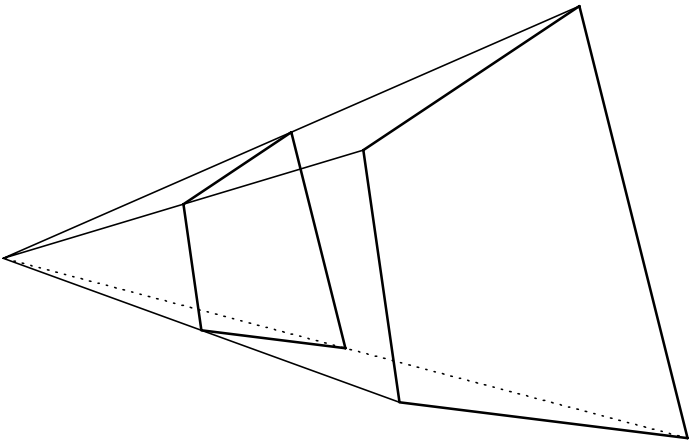


Рис. 13.10. Усеченная пирамида видимости

Для отсечения по этой области можно принять во внимание тот факт, что область в пространстве отсечения (*clip space*) преобразуется в куб с ребром длиной 2 вокруг начала координат.

Поскольку преобразование в пространство отсечения реализуется при помощи произведения матриц модельного преобразования и проектирования, то отсюда легко получить уравнения для шести отсекающих плоскостей. Удобнее всего заключить получение этих плоскостей и проверку на попадание в класс. В листинге 13.4 приводится его описание и реализация.

Листинг 13.4. 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 ();
```

```
    bool    pointInFrustum ( const Vector3D&   v    ) const;
    bool    boxInFrustum   ( const BoundingBox& box ) const;
};
#endif
```

Для простоты работы плоскости здесь представлены экземплярами класса `Plane`, заголовочный файл которого приводится в листинге 13.5.

Листинг 13.5. Описание класса `Plane`

```
//
// Class for simple plane (p,n) + dist = 0
//
#ifndef __PLANE__
#define __PLANE__
#include    "Vector3D.h"
enum
{
    IN_FRONT = 1,
    IN_BACK  = 2,
    IN_BOTH  = 3,
    IN_PLANE = 4
};
class   Plane
{
public:
    Vector3D n;                 // normal vector
    float    dist;              // signed distance along n
                                // build plane from normal
                                // and signed distance
    int      nearPointMask;     // -1 if not initialized
    int      mainAxis;          // index of main axis
    Plane () : n ( 0, 0, 1 )
    {
    dist = 0;
        computeNearPointMaskAndMainAxis ();
    }
    Plane ( const Vector3D& normal, float d ) : n (normal),
        nearPointMask ( -1 )
```

```
{
    n.normalize ();
    dist = d;
    computeNearPointMaskAndMainAxis ();
}
                            // build plane from plane equation
Plane ( float nx, float ny, float nz, float d ) : n (nx, ny, nz),
     nearPointMask ( -1 )
{
    n.normalize ();
    dist = d;
    computeNearPointMaskAndMainAxis ();
}
                            // build plane from normal and
                           // point on plane
Plane ( const Vector3D& normal, const Vector3D& point ) :
         n ( normal ), nearPointMask ( -1 )
{
    n.normalize ();
    dist = -(point & n);
    computeNearPointMaskAndMainAxis ();
}
                            // build plane from 3 points
Plane ( const Vector3D& p1, const Vector3D& p2, const Vector3D& p3 )
: nearPointMask ( -1 )
{
    n = (p2 - p1) ^ (p3 - p1 );
    n.normalize ();
    dist = -(p1 & n);
    computeNearPointMaskAndMainAxis ();
}
Plane ( const Plane& plane ) : n ( plane.n ), dist ( plane.dist )
{
    nearPointMask = plane.nearPointMask;
    mainAxis      = plane.mainAxis;
}
float    signedDistanceTo ( const Vector3D& v ) const
{
    return (v & n) + dist;
```

```
}
float    distanceTo ( const Vector3D& v ) const
{
    return (float)fabs ( signedDistanceTo ( v ) );
}
                              // get point on plane
Vector3D    point () const
{
    return (-dist) * n;
}
                          // classify point
int    classify ( const Vector3D& p ) const
{
    float   v = signedDistanceTo ( p );
    if ( v > EPS )
        return IN_FRONT;
    else
    if ( v < -EPS )
        return IN_BACK;
    return IN_PLANE;
}
void    flip ()
{
    n    = -n;
    dist = -dist;
    computeNearPointMaskAndMainAxis ();
}
float   closestPoint ( const Vector3D& p, Vector3D& res ) const
{
    float   distanceToPlane = - dist - (p & n);
    res = p + distanceToPlane * n;
    return distanceToPlane;
}
bool    intersectByRay ( const Vector3D& org, const Vector3D& dir,
                         float& t ) const
{
    float   numer = - (dist + (org & n));
    float   denom = dir & n;
```

```
        if ( fabs ( denom ) < EPS )
           return false;
        t = numer / denom;
        return true;
    }
    Vector3D    makeNearPoint ( const Vector3D& minPoint,
                                const Vector3D& maxPoint ) const
    {
        return Vector3D ( nearPointMask & 1 ? maxPoint.x : minPoint.x,
                          nearPointMask & 2 ? maxPoint.y : minPoint.y,
                          nearPointMask & 4 ? maxPoint.z : minPoint.z );
    }
    Vector3D    makeFarPoint ( const Vector3D& minPoint,
                               const Vector3D& maxPoint ) const
    {
        return Vector3D ( nearPointMask & 1 ? minPoint.x : maxPoint.x,
                          nearPointMask & 2 ? minPoint.y : maxPoint.y,
                          nearPointMask & 4 ? minPoint.z : maxPoint.z );
    }
    int getMainAxis () const
    {
        return mainAxis;
    }
protected:
    void    computeNearPointMaskAndMainAxis ();
};
#endif
```

Реализация этого класса приведена в листинге 13.6.

Листинг 13.6. Реализация класса `Plane`

```
//
// Class for simple plane (p,n) + dist = 0
//
#include    "Plane.h"
#include    "Math3D.h"
void    Plane :: computeNearPointMaskAndMainAxis ()
```

```
{
       nearPointMask = computeNearPointMask ( n );
       mainAxis      = n.getMainAxis ();
}
```

В листинге 13.7 приводится реализация класса `Frustum` с некоторыми оптимизациями.

Листинг 13.7. Реализация класса `Frustum`

```
//
// Frustum class for OpenGL
//
#ifdef    _WIN32
    #include    <windows.h>
#endif
#include    <GL/gl.h>
#include    "Frustum.h"
#include    "Math3D.h"
#include    "BoundingBox.h"
Frustum :: Frustum ()
{
    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, основной объем кода в классе `Frustum` связан с нахождением произведения модельной матрицы на матрицу проектирования и получением по элементам этого произведения коэффициентов отсекающих плоскостей.

В этом классе так же, как и в классе `Plane`, содержатся некоторые оптимизации для работы с *AABB*.

Для этих классов проверка на попадание заданного объекта (унаследованного от класса `SceneObject`) заключается в вызове метода `boxInFrustum`, параметром для которого служит *AABB* интересующего нас объекта.

Функцию `display` можно изменить, добавив отсечение по области видимости, как показано в листинге 13.8.

Листинг 13.8. Новая реализация функции `display` (из файла room-2.cpp)

```
void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    Frustum    frustum;
                        // draw main hall
    drawBox  ( Vector3D ( -roomSize, -roomSize, -3 ),
               Vector3D ( 2*roomSize, 2*roomSize, 7 ),
               stoneMap, false );
                        // draw objects
    for ( list <SceneObject *> :: iterator it = objects.begin ();
             it != objects.end (); ++it )
         if ( frustum.boxInFrustum ( (*it) -> getBounds () ) )
            (*it) -> draw ();
    frames++;
    if ( (frames % 5) == 0 )        // compute every 5 frames
    {
        int    time2 = glutGet ( GLUT_ELAPSED_TIME );
        char    str [256];
        fps         = (float)frames/(0.001*(time2 - elapsedTime));
        elapsedTime = time2;
```

```
        frames      = 0;
        sprintf ( str, "OpenGL occlusion test example"\
                       ", FPS : %5.2f", fps );
        glutSetWindowTitle ( str );
    }
    glutSwapBuffers ();
}
```

Такой прием позволяет отсечь довольно большое число объектов (в среднем около 5/6 от общего числа), что дает заметный прирост быстродействия (см. табл. 13.1).

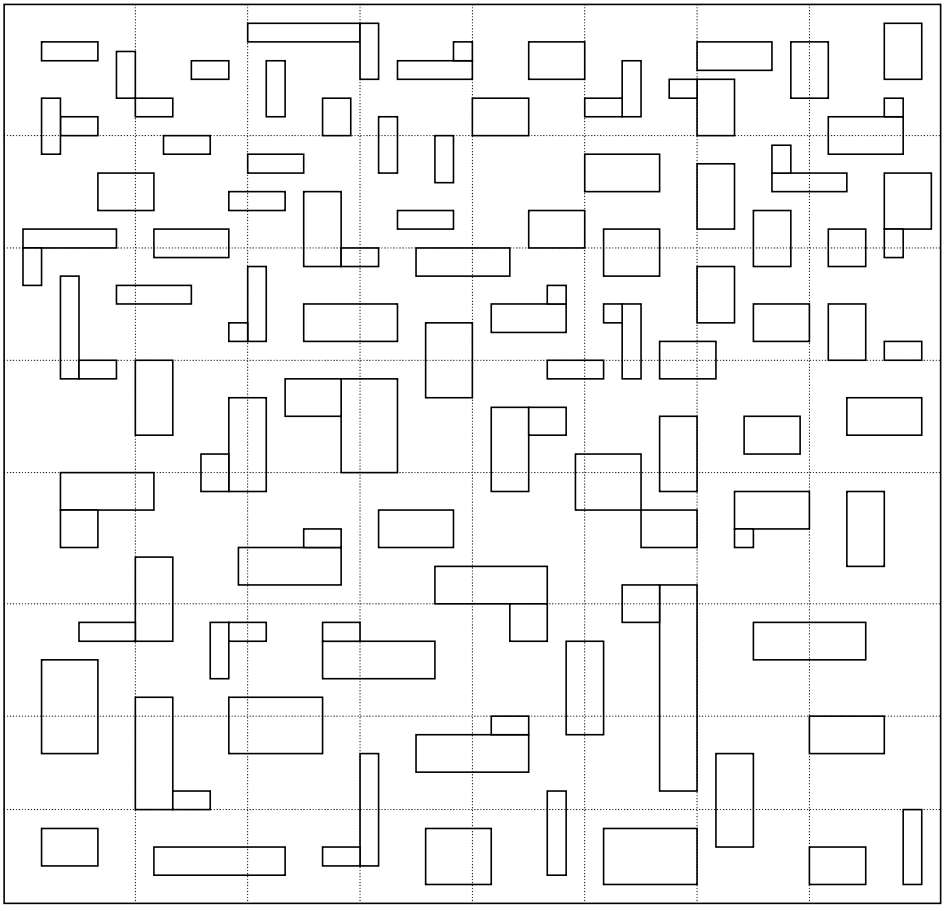


Рис. 13.11. Равномерное разбиение сцены на клетки

Однако, несмотря на то, что сразу отбрасываются заведомо не попадающие в область видимости объекты, быстродействие программы все еще остается низким. Причина этого заключается в очень большом объеме перезаписи (*overdraw*). Для многих пикселов запись производится десятки раз, что и сказывается на быстродействии программы.

Для ускорения работы программы существует расширение ARB_occlusion_query, задачей которого является "борьба" с *overdraw*.

Чтобы не выполнять отдельную проверку на видимость для каждого объекта сцены, введем следующую структуру разбиения пространства: поделим всю сцену на $N \times N$ одинаковых прямоугольных клеток (ячеек) (рис. 13.11).

С каждой такой ячейкой (она и будет выступать единицей отсечения) свяжем список содержащихся (хотя бы частично) в ней объектов и ограничивающее тело. Для работы с ячейками мы выберем класс `Cell`, описание которого содержит листинг 13.9.

Листинг 13.9. Описание класса `Cell`

```
//
// Cell class containing several objects inside
//
#ifndef    __CELL__
#define    __CELL__
#include    "BoundingBox.h"
#include    <list>
using namespace std;
class    SceneObject;
class    Cell
{
    list <SceneObject *>  objects;
    BoundingBox           box;
    mutable int           visToFrame;        // to which frame no we
                                             // assume it's visible
public:
    Cell ( const Vector3D& org, const Vector3D& size,
              const list <SceneObject *>& srcObjects );
    const BoundingBox&    getBounds () const
    {
        return box;
    }
```

```
    int    getVisToFrame () const
    {
        return visToFrame;
    }
    void    render    ( int frameNo ) const;
    void    postQuery () const;
};
#endif
```

Основными методами этого класса являются вывод (метод `render`) и запрос на видимость ограничивающего тела (метод `postQuery`).

В листинге 13.10 приводится реализация данного класса.

Листинг 13.10. Реализация класса `Cell`

```
//
// Cell class containing several objects inside
//
#include    "libExt.h"
#include    "Cell.h"
#include    "SceneObject.h"
#include    "boxes.h"
#include    <glut.h>
Cell :: Cell ( const Vector3D& org, const Vector3D& size,
               const list <SceneObject *>& srcObjects ) : box ( org,
                                                               org + size )
{
            // now add any object with not-empty intersection
            // and compute real bounding box
    BoundingBox    b;
    for ( list <SceneObject *> :: const_iterator it =
                   srcObjects.begin (); it != srcObjects.end (); ++it )
    {
        SceneObject * object = *it;
        if ( box.intersects ( object -> getBounds () ) )
        {
            objects.push_front ( object );
            b.merge ( object -> getBounds () );
        }
```

```
    }
    box        = b;        // set real bounding box
    visToFrame = -1;
}
void    Cell :: render ( int frameNo ) const
{
    for ( list <SceneObject *> :: const_iterator it =
                objects.begin (); it != objects.end (); ++it )
        if ( (*it) -> frame != frameNo )     // avoid rerendering of
                                             // already rendered
                                             // objects
        {
            (*it) -> draw ();
            (*it) -> frame = frameNo;
        }
    if ( visToFrame < frameNo )            // was invisible
        visToFrame = frameNo + 3;          // assume visible for
                                           // next 3 frames
}
void    Cell :: postQuery () const
{
    drawBox ( box.getMinPoint (), box.getSize (), 0, false );
}
```

Для повышения эффективности лучше всего организовать вывод ячеек в порядке удаления от наблюдателя (*front-to-back*). Достаточно простым способом достижения этого является вывод ячеек по вертикальным или горизонтальным блокам (в зависимости от того, к какому направлению ближе вектор взгляда).

Так, в случае, когда взгляд наблюдателя в основном направлен вдоль оси *Ox*, вывод ячеек будет осуществляться столбцами от левого до правого края области видимости (рис. 13.12). Для каждой ячейки из такого столбца выполняется запрос на видимость и происходит ожидание готовности последнего запроса (поскольку по стандарту результаты будут поступать в порядке поступления запросов).

После получения результатов проверки на видимость выводятся лишь видимые ячейки. Небольшая оптимизация, которую здесь можно с успехом при-

менить, заключается в прекращении обработки столбцов, как только мы встретим полностью невидимый (закрытый) столбец.

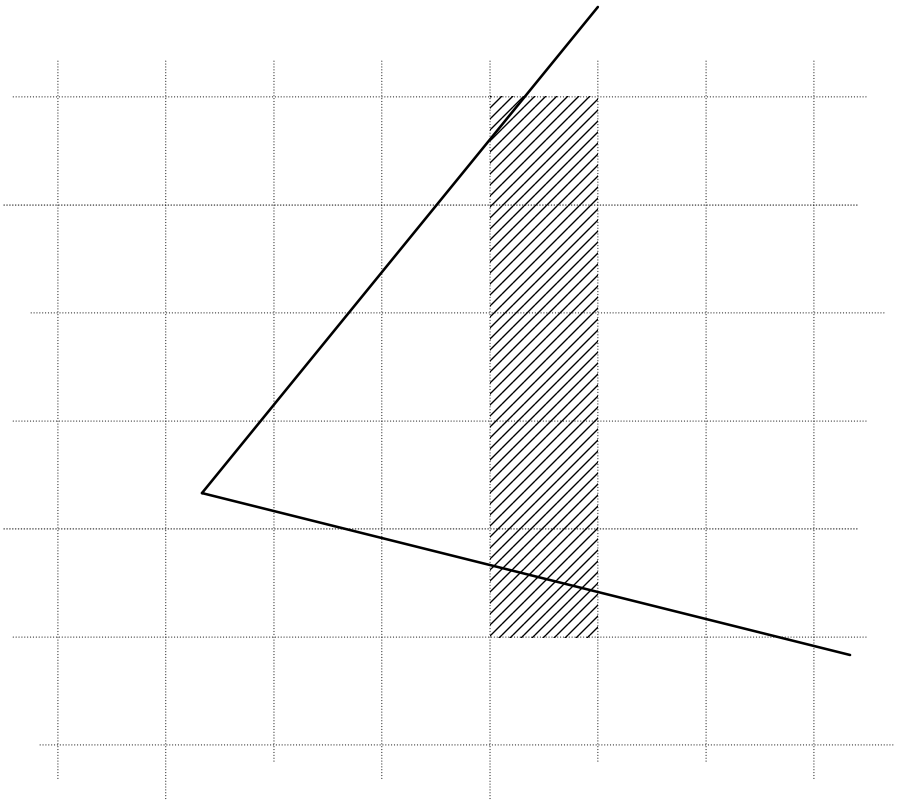


Рис. 13.12. Очередной блок ячеек для проверки

В случае, когда взгляд наблюдателя направлен в основном вдоль оси *Oy*, вывод (и проверка) ячеек осуществляется по строкам.

Важным моментом, на который следует обратить внимание, является возможность повторного вывода объекта из-за их пересечения (один и тот же объект может принадлежать сразу нескольким ячейкам). Чтобы этого избежать, проще всего для каждого объекта хранить номер кадра, в котором он был выведен в последний раз. При выводе объекта достаточно проверить, совпадает ли текущий номер с номером кадра, в котором данный объект был выведен последний раз. Если имеет место совпадение, то это значит, что в данном кадре объект уже выводился и сейчас его можно пропустить. Листинг 13.11 содержит пример вывода с проверкой на видимость.

Листинг 13.11. Вывод сцены с использованием проверки видимости

```
//
// Sample to show occlusion culling via ARB_occlusion_query extension
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    <list>
using namespace std;
#include    "libTexture.h"
#include    "TypeDefs.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "boxes.h"
#include    "SceneObject.h"
#include    "Frustum.h"
#include    "Cell.h"
#define        N    20              // size of matrix
Vector3D    eye     ( 0, 0, -2 );    // camera position
Vector3D    viewDir ( 1, 0, 0 );     // viewer direction
unsigned    decalMap [3];            // decal (diffuse) texture
unsigned    stoneMap;
int         mouseOldX = 0;
int         mouseOldY = 0;
Vector3D    rot ( 0, 0, 0 );
float       step = 0.003;             // movement rate per second
bool        mousePressed = false;
float       roomSize     = 100;        // room half size
int        frames       = 0;
int        elapsedTime  = 0;
float      fps          = 0;
list <SceneObject *>    objects;        // global list of objects
Cell                  * cells [N][N];
void    setupCamera ();
inline    float    rnd ()
```

```
{
    return (float) rand () / (float)RAND_MAX;
}
void    initObjects ()
{
    for ( int i = 1 - roomSize;  i < roomSize - 1; i += 2 )
        for ( int j = 1 - roomSize; j < roomSize - 1; j += 2 )
        {
            float    x    = i + rnd () * 1.2;
            float    y    = j + rnd () * 1.2;
            float    sx   = 0.7 + 0.6 * rnd ();
            float    sy   = 0.7 + 0.5 * rnd ();
            int      type = 1 + rand () % 4;
            int      tex  = rand () % 3;
            if ( (rand () % 15) == 14 )
                objects.push_back ( new TeapotObject (
                                Vector3D ( x, y, -2 ),
                                Vector3D ( sx*0.5, sy*0.5,(sx+sy)*0.25 ),
                                decalMap [tex] ) );
            else
                objects.push_front ( new BoxObject (
                                Vector3D ( x, y, -3 ),
                                Vector3D ( sx, sy, sx + sy ),
                                decalMap [tex], type ) );
        }
}
void    initCells ()
{
    float    step = 2*roomSize / (float) N;
    for ( int i = 0; i < N; i++ )
        for ( int j = 0; j < N; j++ )
        {
            float    x = -roomSize + i*step;
            float    y = -roomSize + j*step;
            cells [i][j] = new Cell ( Vector3D ( x, y, -3 ),
                                      Vector3D ( step, step, 5 ),
                                      objects );
        }
}
```

```
void    renderMajorX ( const Frustum& frustum, int incr )
{
    float    step = 2*roomSize / (float) N;
    int      i0   = (eye.x + roomSize) / step;
    int      j0   = (eye.y + roomSize) / step;
    int      jMin = j0;
    int      jMax = j0;
    int      i    = i0;
    int      j;
    unsigned    ids [100];
    int         numQueries;
    glGenQueriesARB ( N, ids );
    for ( ; i >= 0 && i < N; i += incr )
    {
                        // update jMin, jMax
        for ( j = jMin - 1; j >= 0; j-- )
            if ( !frustum.boxInFrustum (
                         cells [i][j] -> getBounds () ) )
                break;
        jMin = j + 1;
        for ( j = jMax + 1; j < N; j++ )
            if ( !frustum.boxInFrustum (
                          cells [i][j] -> getBounds () ) )
                break;
        jMax = j - 1;
        if ( i == i0 )    // for first row just render
        {
            for ( j = jMin; j <= jMax; j++ )
                cells [i][j] -> render ( frames );
            continue;
        }
                              // now post queries for (i,jMin)-(i,jMax)
        numQueries = jMax - jMin + 1;
                                    // disable writing
        glColorMask ( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );
        glDepthMask ( GL_FALSE );
        glDisable   ( GL_TEXTURE_2D );
        for ( j = jMin; j <= jMax; j++ )
```

```
        {
            glBeginQueryARB ( GL_SAMPLES_PASSED_ARB,
                              ids [j - jMin] );
            cells [i][j] -> postQuery ();
            glEndQueryARB ( GL_SAMPLES_PASSED_ARB );
        }
        glColorMask ( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE );
        glDepthMask ( GL_TRUE );
        glEnable    ( GL_TEXTURE_2D );
                                        // now wait for the last query
        int        available;
        unsigned    count;
        unsigned    idLast     = ids [numQueries - 1];
        bool        anyVisible = false;
        do
        {
            glGetQueryObjectivARB ( idLast,
                                    GL_QUERY_RESULT_AVAILABLE_ARB, &available );
        } while (!available);
                                       // restore all masks and options
        for ( j = jMin; j <= jMax; j++ )
        {
            glGetQueryObjectuivARB ( ids [j - jMin],
                                     GL_QUERY_RESULT_ARB, &count );
            if ( count > 0 )
            {
                cells [i][j] -> render ( frames );
                anyVisible = true;
            }
        }
        if ( !anyVisible )    // if the whole row is invisible no
            break;            // sense going further
    }
    glDeleteQueriesARB ( N, ids );
}
void    renderMajorY ( const Frustum& frustum, int incr )
{
    float    step = 2*roomSize / (float) N;
```

```
int      i0   = (eye.x + roomSize) / step;
int      j0   = (eye.y + roomSize) / step;
int      iMin = i0;
int      iMax = i0;
int      j    = j0;
int      i;
unsigned   ids [100];
int        numQueries;
glGenQueriesARB ( N, ids );
for ( ; j >= 0 && j < N; j += incr )
{
                // update iMin, iMax
    for ( i = iMin - 1; i >= 0; i-- )
        if ( !frustum.boxInFrustum (
                     cells [i][j] -> getBounds () ) )
            break;
    iMin = i + 1;
    for ( i = iMax + 1; i < N; i++ )
        if ( !frustum.boxInFrustum (
                    cells [i][j] -> getBounds () ) )
            break;
    iMax = i - 1;
    if ( j == j0 )
    {
        for ( i = iMin; i <= iMax; i++ )
            cells [i][j] -> render ( frames );
        continue;
    }
                          // now post queries for (iMin,j)-(iMax,j)
    numQueries = iMax - iMin + 1;
                         // disable writing
    glColorMask ( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );
    glDepthMask ( GL_FALSE );
    glDisable   ( GL_TEXTURE_2D );
    for ( i = iMin; i <= iMax; i++ )
    {
        glBeginQueryARB ( GL_SAMPLES_PASSED_ARB,
                          ids [i - iMin] );
```

```
            cells [i][j] -> postQuery ();
            glEndQueryARB ( GL_SAMPLES_PASSED_ARB );
        }
                                        // restore all masks and options
        glColorMask ( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE );
        glDepthMask ( GL_TRUE );
        glEnable    ( GL_TEXTURE_2D );
                                          // now wait for the last query
        int         available;
        unsigned    count;
        unsigned    idLast     = ids [numQueries - 1];
        bool        anyVisible = false;
        do
        {
            glGetQueryObjectivARB ( idLast,
                          GL_QUERY_RESULT_AVAILABLE_ARB, &available );
        } while (!available);
        for ( i = iMin; i <= iMax; i++ )
        {
            glGetQueryObjectuivARB ( ids [i - iMin],
                                     GL_QUERY_RESULT_ARB, &count );
            if ( count > 0 )
            {
                cells [i][j] -> render ( frames );
                anyVisible = true;
            }
        }
        if ( !anyVisible )    // if the whole row is invisible no
            break;        // sense going further
    }
    glDeleteQueriesARB ( N, ids );
}
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 );
    Frustum    frustum;
                    // draw main hall
    drawBox  ( Vector3D ( -roomSize, -roomSize, -3 ),
               Vector3D ( 2*roomSize, 2*roomSize, 5 ),
               stoneMap, false );
                    // draw objects
    if ( fabs ( viewDir.x ) >= fabs ( viewDir.y ) )
        renderMajorX ( frustum, viewDir.x > 0 ? 1 : -1 );
    else
        renderMajorY ( frustum, viewDir.y > 0 ? 1 : -1 );
    frames++;
    if ( (frames % 5) == 0 )    // compute every 5 frames
    {
        int    time2 = glutGet ( GLUT_ELAPSED_TIME );
        char    str [256];
        fps         = (float)frames/(0.001*(time2 - elapsedTime));
        elapsedTime = time2;
        frames = 0;
        sprintf ( str, "OpenGL occlusion test example,"\
                       " FPS : %5.2f", fps );
        glutSetWindowTitle ( str );
    }
    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, 0.1, 1000 );
    setupCamera    ();
}
```

```
void    setupCamera ()
{
    Vector3D    to = eye + viewDir;
    glMatrixMode   ( GL_MODELVIEW );
    glLoadIdentity ();
    gluLookAt      ( eye.x, eye.y, eye.z,    // eye
                     to.x,  to.y,  to.z,     // center
                     0.0, 0.0, 1.0 );        // up
}
void motion ( int x, int y )
{
    static    bool    mouseInited = false;
    if ( !mouseInited )
    {
        mouseInited = true;
        mouseOldX   = x;
        mouseOldY   = y;
        return;
    }
    float    scale = -M_PI / 100;
    viewDir.x = cos ( scale * x );
    viewDir.y = sin ( scale * x );
    viewDir.z = 0;
    setupCamera ();
    glutPostRedisplay ();
}
void mouse ( int button, int state, int x, int y )
{
    if ( button == GLUT_LEFT_BUTTON )
        mousePressed = (state == GLUT_DOWN);
    motion ( x, y );
}
void key ( unsigned char key, int x, int y )
{
    if ( key == 27 || key == 'q' || key == 'Q' )    // quit requested
                exit ( 0 );
}
void    animate ()
```

```
{
    static    bool   timeInited = false;
    static    int    lastTime;
    if ( !timeInited )
    {
        lastTime   = glutGet ( GLUT_ELAPSED_TIME );
        timeInited = true;
    }
    if ( mousePressed )
    {
        Vector3D    delta = viewDir * (step * (
                      glutGet ( GLUT_ELAPSED_TIME ) - lastTime)*0.001f);
        Vector3D    e     = eye + delta;
        if ( e.x < roomSize && e.x > -roomSize )
            eye.x = e.x;
        if ( e.y < roomSize && e.y > -roomSize )
            eye.y = e.y;
        setupCamera ();
    }
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                        // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                           // create window
    int    win = glutCreateWindow ( "OpenGL occlusion test example" );
                        // register handlers
    glutDisplayFunc       ( display );
    glutReshapeFunc       ( reshape );
    glutKeyboardFunc      ( key     );
    glutMouseFunc         ( mouse   );
    glutMotionFunc        ( motion  );
    glutPassiveMotionFunc ( motion  );
    glutIdleFunc          ( animate );
    init ();
```

```
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_ARB_occlusion_query" ) )
    {
        printf ( "ARB_occlusion_query NOT supported" );
        return 2;
    }
    initExtensions ();
    printfInfo ();
    stoneMap     = createTexture2D ( true, "../Textures/block.bmp" );
    decalMap [0] = createTexture2D ( true, "../Textures/oak.bmp"   );
    decalMap [1] = createTexture2D ( true,
                                        "../Textures/FieldStone.tga" );
    decalMap [2] = createTexture2D ( true, "../Textures/wood1.bmp" );
    initObjects ();
    initCells   ();
    glutMainLoop ();
    return 0;
}
```

Таблица 13.2 содержит результаты расчетов быстродействия (в кадрах в секунду) для трех версий (без отсечения, с отсечением по области видимости и с отсечением и по области видимости, и по закрыванию, листинги 13.3, 13.8 и 13.11 соответственно). Данные получены на компьютере Celeron 2,4 GHz GeForceFX 5600XT.

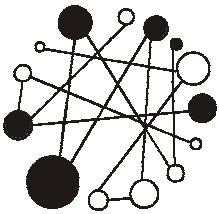
Таблица 13.2. *Быстродействие примеров*

Положения наблюдателя	room (fps)	room-2 (fps)	room-3 (fps)
1	0,6	3,2	28,4
2	0,6	3,5	83,0
3	0,6	3,5	16,8

По результатам этих тестов видно, что для неоптимизированного варианта быстродействие практически не зависит от начального положения и ориентации наблюдателя. Отсечение по пирамиде видимости дает увеличение быстродействия в среднем в 6 раз, что и должно было иметь место. Отсечение закрытых ячеек дополнительно увеличивает быстродействие практически на порядок. Это объясняется тем, что наблюдатель в случае 3 стоит прямо перед стеной, закрывающей от него всю сцену.

Последнее обстоятельство показывает, что механизм отбрасывания невидимых объектов обеспечивает быстродействие, определяемое сложностью только видимой части сцены.

Глава 14



Сжатые текстуры и работа с ними

Одной из характерных особенностей игр и других графических приложений последнего поколения является очень высокий уровень детализации. Помимо сложности геометрии и моделей освещения, это проявляется в существенном увеличении разрешения текстур.

Фактически в современных играх большое число текстур имеют разрешения 512×512, 1024×1024 и выше. Подобные текстуры требуют большого объема видеопамяти для своего хранения (так, RGB-текстура размером 1024×1024 текселов занимает 3 Мбайт памяти).

Таких текстур обычно довольно много, и вместе они занимают очень большой объем видеопамяти. Размещение их в памяти CPU крайне невыгодно из-за больших временных затрат на передачу графическому ускорителю (скорость передачи данных по шине AGP8 заметно уступает скорости передачи данных внутри самого графического ускорителя).

Таким образом, приходится размещать эти текстуры непосредственно в памяти графического ускорителя, объем которой ограничен, ведь кроме текстур в ней приходится размещать различные буферы (кадра, глубины, трафарета, *p*-буфера, и т. п.).

Кроме того, из-за таких текстур, занимающих большой объем памяти, снижается эффективность кэширования при доступе к памяти графического ускорителя, что приводит к уменьшению быстродействия приложения.

Довольно простым выходом из данной ситуации является использование сжатых текстур (этот подход уже давно реализован, например, для передачи изображений в Интернете).

Применение сжатых текстур в графических ускорителях имеет свои особенности. В отличие от изображений в формате JPEG, когда оно сначала распаковывается целиком и уже в таком виде выводится, в памяти графического ускорителя текстура должна храниться в сжатом виде и распаковываться лишь частями по мере необходимости, а не вся целиком.

Это накладывает определенные ограничения на алгоритмы сжатия текстур. Во-первых, они должны быть достаточно просты, чтобы не снижалось быстродействие, а во-вторых, — позволять распаковывать произвольные фрагменты текстуры.

На данный момент распространение получили два подобных способа сжатия текстур: S3TC компании S3 и FXT1 компании 3dfx. Каждый из них обеспечивает сжатие текстур в 4—8 раз. Оба относятся к классу сжатия с потерями, при сжатии изображения происходит потеря информации, за счет чего распакованное изображение будет слегка отличаться от оригинала (хорошо известный алгоритм сжатия JPEG действует так же, алгоритмы GIF и PNG обеспечивают сжатие без потерь). В настоящее время алгоритм S3TC поддерживается всеми современными графическими ускорителями, и далее мы подробно рассмотрим его.

Поддержка сжатых текстур в OpenGL реализуется через механизм расширений. Начиная с версии 1.3 OpenGL поддерживает сжатие текстур через расширение ARB_texture_compression, принятое в марте 2000 г. Оно вводит ряд необходимых функций и констант для работы со сжатыми текстурами, а также несколько внутренних форматов для их использования.

Обратите внимание, что данное расширение само не вводит поддержку каких-либо реальных форматов сжатых текстур (например, S3TC или FXT1). Поддержка конкретных форматов должна осуществляться посредством других расширений, одно из них — EXT_texture_compression_s3tc — мы рассмотрим далее.

Расширение ARB_texture_compression вводит следующие внутренние форматы для сжатых текстур:

- `GL_COMPRESSED_ALPHA_ARB`;
- `GL_COMPRESSED_LUMINANCE_ARB`;
- `GL_COMPRESSED_LUMINANCE_ALPHA_ARB`;
- `GL_COMPRESSED_INTENSITY_ARB`;
- `GL_COMPRESSED_RGB_ARB` и `GL_COMPRESSED_RGBA_ARB`.

Для загрузки сжатых текстур в память графического ускорителя расширение ARB_texture_compression вводит 3 функции.

```
void glCompressedTexImage1DARB ( GLenum target, GLint level,
                                 GLenum internalFormat, GLsizei width,
                                 GLint border, GLsizei imageSize,
                                 const void *data );
void glCompressedTexImage2DARB ( GLenum target, GLint level,
                                 GLenum internalFormat, GLsizei width,
```

```
                                GLsizei height, GLint border,
                                GLsizei imageSize, const void *data );
void glCompressedTexImage3DARB ( GLenum target, GLint level,
                                GLenum internalFormat, GLsizei width,
                                GLsizei height, GLsizei depth,
                                GLint border, GLsizei imageSize,
                                const void *data );
```

Они позволяют загружать одно-, дву- и трехмерные текстуры. Параметры `target`, `level`, `internalFormat`, `width`, `height`, `depth` и `border` имеют тот же смысл, что и в функциях `glTexImage1D`, `glTexImage2D` и `glTexImage3DEXT`. Параметр `data` указывает на область памяти, в которой размещена сжатая текстура в формате, задаваемом параметром `internalFormat`.

При помощи следующих трех функций можно загрузить прямоугольную область текстуры.

```
void glCompressedTexSubImage1DARB ( GLenum target, GLint level,
                                   GLint xOffset, GLsizei width,
                                   GLenum format, GLsizei imageSize,
                                   const void *data );
void glCompressedTexSubImage2DARB ( GLenum target, GLint level,
                                   GLint xOffset, GLint yOffset,
                                   GLsizei width, GLsizei height,
                                   GLenum format, GLsizei imageSize,
                                   const void *data );
void glCompressedTexSubImage3DARB  ( GLenum target, GLint level,
                                    GLint xOffset, GLint yOffset,
                                    GLint zOffset, GLsizei width,
                                    GLsizei height, GLsizei depth,
                                    GLenum format,  GLsizei imageSize,
                                     const void *data );
```

Еще одной вводимой этим расширением функцией является `glGetCompressedTexImageARB`.

```
void glGetCompressedTexImageARB ( GLenum target, GLint lod, void * img );
```

Она позволяет получить в оперативной памяти текстуру в сжатом виде. Параметры `target`, `lod` и `img` имеют тот же смысл, что и в функции `glGetTexImage`.

Размер памяти, занимаемый сжатой текстурой, можно получить при помощи следующего запроса:

```
int imageSize;
glGetTexLevelParameteriv ( target, level,
                           GL_TEXTURE_COMPRESSED_IMAGE_SIZE_ARB,
                           &imageSize );
```

Здесь параметр `target` задает тип текстуры, а `level` — уровень в пирамиде фильтрования (0 соответствует исходной текстуре).

Вводится также механизм, позволяющий задавать предпочтения при работе со сжатыми текстурами.

Команда

```
glHint  ( GL_TEXTURE_COMPRESSION_HINT_ARB, GL_NICEST );
```

задает режим, обеспечивающий наибольшее качество получаемого изображения, а

```
glHint  ( GL_TEXTURE_COMPRESSION_HINT_ARB, GL_FASTEST ); —
```

наибольшую скорость.

Для поддержки сжатых текстур в библиотеке libTexture кроме обычного класса `Texture` введен дополнительный — `CompressedTexture`, предназначенный для работы со сжатыми текстурами.

В листинге 14.1 приводится заголовочный файл для этого класса.

Листинг 14.1. Описание класса `CompressedTexture`

```
//
// Compressed texture class
//
#ifndef    __COMPRESSED_TEXTURE__
#define    __COMPRESSED_TEXTURE__
#include    "Texture.h"
class    CompressedTexture : public Texture
{
public:
    CompressedTexture ( int theWidth, int theHeight,
                        int theNumComponents, int theFormat,
                        int theLevels, int theSize );
    virtual    bool    upload ( int target, bool mipmap = true );
};
#endif
```

В листинге 14.2 приведена реализация класса `CompressedTexture`.

Листинг 14.2. Реализация класса `CompressedTexture`

```
//
// Compressed texture class
//
#ifdef    _WIN32
    #include    <windows.h>
#endif
#include    <malloc.h>
#include    <memory.h>
#include    "libExt.h"
#include    "CompressedTexture.h"
CompressedTexture :: CompressedTexture ( int theWidth, int theHeight,
                                         int theNumComponents,
                                         int theFormat,
                                         int theLevels, int theSize ) :
                                                 Texture ()
{
    width         = theWidth;
    height        = theHeight;
    numComponents = theNumComponents;
    format        = theFormat;
    data          = (byte *) malloc ( theSize );
    levels        = theLevels;
    compressed    = true;
}
bool    CompressedTexture :: upload ( int target, bool mipmap )
{
    int    w    = width;
    int    h    = height;
    int    offs = 0;     int    blockSize;
    int    size;
    if( format == GL_COMPRESSED_RGBA_S3TC_DXT1_EXT )
            blockSize = 8;
    else
        blockSize = 16;
    for ( int i = 0; i < levels; i++ )
```

```
    {
            if ( w  == 0 )
                w  = 1;
            if ( h == 0 )
                 h = 1;
        size = ((w+3)/4) * ((h+3)/4) * blockSize;
        glCompressedTexImage2DARB ( target, i, format, w, h, 0, size,
                                    data + offs );
        offs += size;
        w = w / 2;
        h = h / 2;
    }
    return true;
}
```

Как видно из листинга, данный класс поддерживает загрузку двумерных сжатых текстур вместе с набором уровней пирамидального фильтрования, что понадобится нам далее при работе с DDS-файлами.

Сжатие методом S3TC

На практике вместе с расширением ARB_texture_compression применяется EXT_texture_compression_s3tc, принятое в ноябре 2001 г. Оно позволяет работать с текстурами, сжатым алгоритмом S3TC.

Данное расширение вводит ряд внутренних форматов текстур, соответствующих методу сжатия S3TC, различающихся областью применимости. Все они приведены в табл. 14.1.

Таблица 14.1. *Внутренние форматы текстур, вводимые расширением EXT_texture_compression_s3tc*

Формат	Поддерживается ли α-канал	Степень сжатия
GL_COMPRESS_RGB_S3TC_DXT1_EXT	Нет	6:1
GL_COMPRESS_RGBA_S3TC_DXT1_EXT	Да	8:1
GL_COMPRESS_RGBA_S3TC_DXT3_EXT	Да	4:1
GL_COMPRESS_RGBA_S3TC_DXT5_EXT	Да	4:1

Рассмотрим теперь подробно реализацию каждого из этих форматов.

Формат *GL_COMPRESS_RGB_S3TC_DXT1_EXT*

Для сжатия изображения в этом формате оно разбивается на блоки 4×4 текселя (при необходимости изображение дополняется пикселами). Каждый такой блок текселов кодируется восьмью байтами (рис. 14.1).

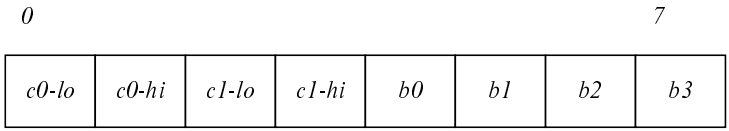


Рис. 14.1. Закодированный блок 4×4 текселя

Первые два байта задают цвет (в дальнейшем обозначаемый как C_0) в формате 5–6–5, т. е. под красную и синюю компоненты отводится по 5 битов, под зеленую — 6. Подобное предпочтение зеленому цвету объясняется гораздо большей чувствительностью к нему человеческого глаза.

Байты 2 и 3 задают еще один цвет в формате 5–6–5, обозначаемый в дальнейшем как C_1.

Последние четыре байта блока задают 32-битовое целое беззнаковое число (байт b_0 соответствует младшим его битам, байт b_3 — старшим). В этом числе под каждый из 16 текселов отведено по два бита, причем биты, соответствующие текселу (x,y), начинаются с позиции $2 \cdot (4 \cdot y + x)$.

По двум битам для каждого текселя по табл. 14.2 и 14.3 определяются коэффициенты k_0 и k_1. Результирующий цвет для текселя задается по формуле:

$$C(x, y) = C_0 \cdot k_0 + C_1 \cdot k_1. \qquad (14.1)$$

Таблица 14.2. *Коэффициенты смешения для случая* $C_0 > C_1$

Биты	Коэффициент k_0	Коэффициент k_1
00	1	0
01	0	1
10	2/3	1/3
11	1/3	2/3

Таблица 14.3. *Коэффициенты смешения для случая* $C_0 \le C_1$

Биты	Коэффициент k_0	Коэффициент k_1
00	1	0
01	0	1
10	$^1/_2$	$^1/_2$
11	0	0

Таким образом, каждый блок из 16 текселов формата RGB, занимающий 16×3=48 байт, заменяется на 8 байт, т. е. коэффициент сжатия равен 6. При этом каждый такой блок характеризуется двумя цветами, а все остальные цвета получаются из этих двух путем линейной интерполяции по формуле (14.1).

Формат *GL_COMPRESS_RGBA_S3TC_DXT1_EXT*

Поддерживает задание α-канала, но при этом считается, что для этого достаточно одного бита на тексел (что имеет место, например, при задании различных масок).

Тогда сжимаемая текстура, как и ранее, разбивается на блоки 4×4 тексела. Блок данных представляет собой RGB-части текселов блока (в формате, описанном ранее).

α-значение для очередного тексела определяется по следующему правилу. Тексел у сопоставляется значение $\alpha = 0$, если $C_0 \le C_1$ и биты для данного тексела равны 11 (в этом случае RGB-значение тексела определяется по табл. 14.3, которая дает черный цвет). Во всех остальных случаях считается $\alpha = 1$.

Таким образом, 16 RGBA текселов (занимающих 64 байта) сжимаются в 8 байтов, что дает коэффициент сжатия 8. Однако за это приходится расплачиваться крайне низкой точностью сжатия α-канала.

Формат *GL_COMPRESS_RGBA_S3TC_DXT3_EXT*

В этом формате, как и ранее, изображение кодируется блоками 4×4 тексела. При этом в сжатых данных сначала идут 8 байтов, представляющих сжатые α-значения для всех текселов (по 4 бита на тексел), а далее 8 байтов, представляющих сжатые по алгоритму `GL_COMPRESS_RGB_S3TC_DXT1_EXT` RGB-

части, причем коэффициенты интерполяции всегда задаются по табл. 14.2 (рис. 14.2).

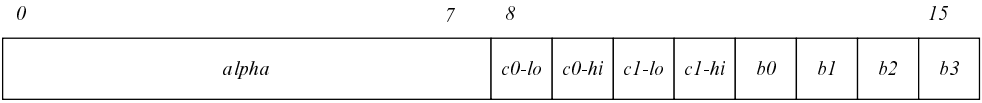


Рис. 14.2. Блок 4×4 тексела, закодированный способом GL_COMPRESS_RGBA_S3TC_DXT1_EXT

При этом байты, служащие для кодирования α-частей, можно трактовать как 64-битовое беззнаковое целое (считая, что первый байт блока соответствует наименее значимым битам), собранное из старших 4-х битов α-частей каждого тексела. Самый первый байт блока содержит α-части для текселов (0, 0) и (0, 1), следующий — α-части текселов (0, 2) и (0, 3) и т. д.

Как легко убедиться, данный способ обеспечивает сжатие исходного изображения в 4 раза (блок из 16 текселов кодируется в 16 байтов).

Формат *GL_COMPRESS_RGBA_S3TC_DXT5_EXT*

Как и ранее, текстура разбивается на блоки 4×4 тексела, которые кодируются независимо друг от друга. Каждый такой блок представлен 16-ю байтами (рис. 14.3).

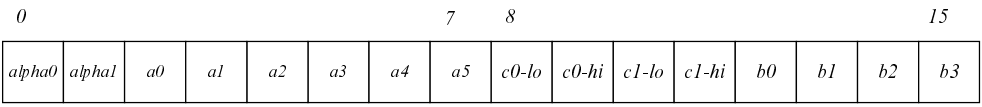


Рис. 14.3. Блок 4×4 тексела, закодированный способом GL_COMPRESS_RGBA_S3TC_DXT1_EXT

Величины *alpha0* и *alpha1* являются 8-битовыми α-значениями, которые будут использоваться для получения α-значений для всех текселов блока путем линейной интерполяции (как это делается с RGB-частями).

Оставшиеся 6 байтов — a_0, a_1, a_2, a_3, a_4, a_5 — задают 48-битовое число (байт a_0 соответствует наименее значимым битам), в котором под каждый тексел отводится по 3 бита. Биты, соответствующие текселу (x, y), начинаются с позиции $3 \cdot (4 \cdot y + x)$.

По этим трем битам согласно табл. 14.4 и 14.5 определяются коэффициенты интерполяции для получения α-частей всех текселов блока.

Таблица 14.4. *Коэффициенты интерполяции для случая alpha0 > alpha1*

Биты	Коэффициент k_0	Коэффициент k_1
000	1	0
001	0	1
010	6/7	1/7
011	5/7	2/7
100	4/7	3/7
101	3/7	4/7
110	2/7	5/7
111	1/7	6/7

Таблица 14.5. *Коэффициенты интерполяции для случая alpha0 ≤ alpha1*

Биты	Коэффициент k_0	Коэффициент k_1
000	1	0
001	0	1
010	4/5	1/5
011	3/5	2/5
100	2/5	3/5
101	1/5	4/5

В случае *alpha0* ≤ *alpha1* битам 110 всегда соответствует $\alpha = 0$, а 111 — $\alpha = 1$. Коэффициент сжатия для этого алгоритма, как и для предыдущего, равен 4.

Некоторые соображения о выборе формата сжатия текстуры

Проще всего вопрос о выборе наиболее подходящего способа сжатия решается для RGB-текстур. Из рассмотренных выше способов единственным подходящим (и к тому же дающим хороший коэффициент сжатия) является `GL_COMPRESS_RGB_S3TC_DXT1_EXT`.

Однако обратите внимание на следующие два момента (свойственные всем рассмотренным алгоритмам). Во-первых, для представления цветов внутри каждого блока используются всего два базовых цвета. Все остальные 14 цветов получаются путем линейной интерполяции базовых. Как известно из теории цвета, таким путем можно получить лишь ограниченное количество возможных цветов из некоторого множества.

Из этого следует, что при формировании пестрой исходной текстуры, когда практически в каждый блок 4×4 тексела попадает несколько сильно отличающихся цветов (например, красный, зеленый и синий) при двух исходных базовых цветах возникнут значительные погрешности. Поэтому подобные текстуры вообще плохо подходят для сжатия алгоритмами семейства S3TC.

Во-вторых, заранее жестко задано множество коэффициентов интерполяции, выбираемых при восстановлении цветов. В случае, если в исходной текстуре есть плавные переходы цветов, это приведет к их разрушению и появлению полос (соответствующих определенной паре коэффициентов интерполяции). Поэтому подобные текстуры тоже плохо подходят для сжатия алгоритмами S3TC.

Для сжатия RGBA-изображений, в которых α-канал служит в качестве маски, наиболее удачным выбором будет способ `GL_COMPRESS_RGBA_S3TC_DXT1_EXT`.

В общем случае выбор алгоритма для сжатия RGBA-текстуры ограничен методами `GL_COMPRESS_RGBA_S3TC_DXT3_EXT` и `GL_COMPRESS_RGBA_S3TC_DXT5_EXT`. Скорее всего, если для α-значений текстуры характерны сильные изменения ("скачки"), то более удачным будет формат `GL_COMPRESS_RGBA_S3TC_DXT3_EXT`, в противном случае, скорее всего, более подходящим будет выбор формата `GL_COMPRESS_RGBA_S3TC_DXT5_EXT`.

Для оценки формата можно попробовать сжать текстуру сразу несколькими способами, сравнить результаты и выбрать наиболее качественное изображение.

Практическая работа с DDS-файлами

Для работы со сжатыми методом S3TC-текстурами чаще всего применяются файлы формата DDS (Direct Draw Surface). Он был впервые введен Microsoft в DirectX 7.0 и расширен впоследствии.

Файлы этого формата могут содержать как обычные (несжатые), так и сжатые методом S3TC-текстуры. Текстуры могут быть одно-, дву- и трехмерными, допустимы также кубические текстурные карты.

Еще одним удобным свойством DDS-файлов является то, что в них может храниться не только исходная текстура, но и все ее промежуточные *mipmap*-уровни.

Рассмотрим работу с этими файлами в OpenGL.

На рис. 14.4 приводится структура DDS-файла.

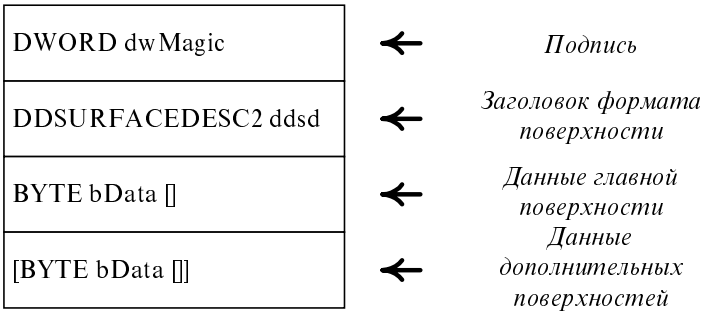


Рис. 14.4. Структура DDS-файла

Первые четыре байта служат для идентификации формата файла, в них содержится определенная подпись, по которой можно установить принадлежность к тому или иному типу файла.

После подписи идет структура, описывающая содержащиеся в файле поверхности. Далее идет (возможно сжатая) главная поверхность (текстура), после чего может идти набор дополнительных.

Для создания текстуры по данному файлу в библиотеке libTexture содержится специальный класс `DdsLoader` (это упрощенная реализация, поддерживающая только двумерные текстуры). Листинг 14.3 содержит описание этого класса.

Листинг 14.3. Описание класса `DdsLoader`

```
//
// Class to load images from DDS files
//
#ifndef    __DDS_LOADER__
#define    __DDS_LOADER__
#include    "TypeDefs.h"
class    Data;
class    Texture;
class    DdsLoader
```

```
{
public:
    DdsLoader () {}
    Texture * load ( Data * data );
};
#endif
```

Как видно из листинга, основной задачей этого класса является создание текстуры по переданному блоку данных. Рассмотрим подробнее реализацию данного класса, приведенную в листинге 14.4.

Листинг 14.4. Реализация класса `DdsLoader`

```
//
// Class to load images from DDS files,
//
#include    <stdio.h>
#ifdef    _WIN32
    #include    <windows.h>
#endif

#include    <string.h>
#include    <GL/gl.h>
#include    "../glext.h"
#include    "Data.h"
#include    "DdsLoader.h"
#include    "CompressedTexture.h"
#pragma pack (push, 1)
struct DDS_PIXELFORMAT
{
        unsigned long dwSize;
        unsigned long dwFlags;
        unsigned long dwFourCC;
        unsigned long dwRGBBitCount;
        unsigned long dwRBitMask;
        unsigned long dwGBitMask;
        unsigned long dwBBitMask;
        unsigned long dwABitMask;
};
```

```
struct DDS_HEADER
{
        unsigned long dwSize;
        unsigned long dwFlags;
        unsigned long dwHeight;
        unsigned long dwWidth;
        unsigned long dwPitchOrLinearSize;
        unsigned long dwDepth;
        unsigned long dwMipMapCount;
        unsigned long dwReserved1[11];
        DDS_PIXELFORMAT ddspf;
        unsigned long dwCaps1;
        unsigned long dwCaps2;
        unsigned long dwReserved2[3];
};
#pragma    pack (pop)
#define DDS_ALPHA_PIXELS        0x00000001L
#define DDS_ALPHA               0x00000002L
#define DDS_FOURCC              0x00000004L
const unsigned long FOURCC_DXT1 = 0x31545844;
const unsigned long FOURCC_DXT3 = 0x33545844;
const unsigned long FOURCC_DXT5 = 0x35545844;
Texture * DdsLoader :: load ( Data * data )
{
    DDS_HEADER   ddsd;
    char         filecode [4];
    int          factor;
    int          bufferSize;
    int          format;
                        // Verify the file is a true .dds file
    data -> seekAbs  ( 0 );
    data -> getBytes ( filecode, 4 );
    if( strncmp( filecode, "DDS ", 4 ) != 0 )
        return NULL;
                        // Get the surface descriptor
    data -> getBytes ( &ddsd, sizeof ( ddsd ) );
    if ( (ddsd.ddspf.dwFlags & DDS_FOURCC) == 0 )// not compressed
    {
        int    numComponents = 0;
```

```
    if ( ddsd.ddspf.dwFlags & DDS_ALPHA_PIXELS )    // ARGB
        numComponents = 4;
    else                                             // RGB
        numComponents = 3;
    int    bytesPerLine = ddsd.dwWidth * numComponents;
    if ( (bytesPerLine & 3) != 0 )    // do dword alignment
        bytesPerLine += 4 - (bytesPerLine & 3);
    byte    * buf     = new byte [bytesPerLine];
    Texture * texture = new Texture ( ddsd.dwWidth,
                                      ddsd.dwHeight,
                                      numComponents );
    for ( int i = 0; i < ddsd.dwHeight; i++ )
    {
        data -> getBytes ( buf, bytesPerLine );
                                        // rearrange components
        byte * dest = texture -> getData () +
                                  i * ddsd.dwWidth * numComponents;
        byte * src  = buf;
        for ( register int j = 0; j < ddsd.dwWidth; j++ )
        {
            dest [0] = src [2];            // red
            dest [1] = src [1];            // green
            dest [2] = src [0];            // blue
            if ( numComponents == 4 )
                dest [3] = src [3];        // alpha
            dest += numComponents;
            src  += numComponents;
        }
    }
    return texture;
}
//
// This .dds loader supports the loading of compressed formats DXT1,
// DXT3 and DXT5.
//
switch ( ddsd.ddspf.dwFourCC )
{
    case FOURCC_DXT1:
                        // DXT1's compression ratio is 8:1
```

```
            format = GL_COMPRESSED_RGBA_S3TC_DXT1_EXT;
            factor = 2;
            break;
        case FOURCC_DXT3:
                            // DXT3's compression ratio is 4:1
            format = GL_COMPRESSED_RGBA_S3TC_DXT3_EXT;
            factor = 4;
            break;
        case FOURCC_DXT5:
                            // DXT5's compression ratio is 4:1
            format = GL_COMPRESSED_RGBA_S3TC_DXT5_EXT;
            factor = 4;
            break;
        default:
            return NULL;
    }
    //
    // How big will the buffer need to be to load all of the pixel data
    // including mip-maps?
    //
    if( ddsd.dwPitchOrLinearSize == 0 )
        return NULL;
    int    numComponents = 4;
    if( ddsd.ddspf.dwFourCC == FOURCC_DXT1 )
        numComponents = 3;
    if( ddsd.dwMipMapCount > 1 )
        bufferSize = ddsd.dwPitchOrLinearSize * factor;
    else
        bufferSize = ddsd.dwPitchOrLinearSize;
    CompressedTexture * texture = new CompressedTexture (
                                        ddsd.dwWidth,
                                        ddsd.dwHeight,
                                        numComponents, format,
                                        ddsd.dwMipMapCount,
                                        bufferSize );
    data -> getBytes ( texture -> getData (), bufferSize );
    return texture;
}
```

Главную информацию о данных внутри файла несет в себе его заголовок (структура `DDS_HEADER`). Основные поля этой структуры приведены в табл. 14.6.

Таблица 14.6. *Основные поля заголовка DDS-файла*

Поле	Описание
`dwSize`	Размер заголовка, должен быть равен 124
`dwFlags`	Флаги, обозначающие, какие поля содержат осмысленные значения. Обязательно включает в себя `DDSD_CAPS`, `DDSD_PIXELFORMAT`, `DDSD_WIDTH`, `DDSD_HEIGHT`
`dwHeight`	Высота главной текстуры в пикселах
`dwWidth`	Ширина главной текстуры в пикселах
`dwDepth`	Для трехмерных текстур глубина текстуры. При этом поле `dwFlags` должно содержать в себе бит `DDSD_DEPTH`
`dwPitchOrLinearSize`	Для несжатых текстур это количество байтов на строку (выровненное по размеру двойного слова) Для сжатых — полное количество байтов под главное изображение. В этом случае поле `dwFlags` должно содержать бит `DDSD_LINEARSIZE`
`dwMipMapCount`	Для изображения с *mipmap*-уровнями, данное поле содержит полное число уровней для главного изображения. Поле `dwFlags` должно содержать бит `DDSD_MIPMAPCOUNT`
`ddspf`	Структура, задающая используемый формат пикселов в изображении
`dwCaps1`	Поле, содержащее дополнительную информацию. Обязательно должно содержать бит `DDSCAPS_TEXTURE`. Если в файле есть *mipmap*-уровни, то должен содержаться бит `DDSCAPS_MIPMAP`. Если в файле содержится более одного главного изображения (например, для кубических текстурных карт и трехмерных текстур), то должен присутствовать бит `DDSCAPS_COMPLEX`
`dwCaps2`	Поле, содержащее дополнительную информацию. Для кубических текстурных карт должен присутствовать бит `DDSCAPS2_CUBEMAP`, вместе с битами для каждой из сторон: `DDSCAPS2_CUBEMAP_POSITIVEX`, `DDSCAPS2_CUBEMAP_NEGATIVEX`, `DDSCAPS2_CUBEMAP_POSITIVEY`, `DDSCAPS2_CUBEMAP_NEGATIVEY`, `DDSCAPS2_CUBEMAP_POSITIVEZ` и `DDSCAPS2_CUBEMAP_NEGATIVEX`. Для трехмерных текстур должен присутствовать бит `DDSCAPS2_VOLUME`

Вся информация о внутренней структуре текстуры (включая сжатие) хранится в специальной структуре DDS_PIXELFORMAT, описанной в табл. 14.7.

Таблица 14.7. *Описание структуры* DDS_PIXELFORMAT

Поле	Описание
dwSize	Размер данной структуры, должен быть равен 32
dwFlags	Флаги, обозначающие, какие поля содержат информацию. Для несжатых текстур обычно устанавливается флаг DDPF_RGB, для сжатых — DDPF_FOURCC
dwFourCC	Задает формат сжатых текстур. Принимает одно из значений 'DXT1', 'DXT2', 'DXT3', 'DXT4' и 'DXT5'. В OpenGL форматы 'DXT2' и 'DXT4' не используются
dwRGBBitCount	Для RGB-форматов — число битов на пиксел; поле dwFlags должно включать бит DDPF_RGB
dwRBitMask	Для RGB-форматов это битовая маска для красной компоненты
dwGBitMask	То же для зеленой компоненты
dwBBitMask	То же для синей компоненты
dwABitMask	Для RGBA-форматов это битовая маска альфа-канала. В случае присутствия альфа-канала в поле dwFlags должен быть выставлен бит DDPF_ALPHAPIXELS

Средства для работы со сжатыми текстурами

В комплекте NVSDK содержится ряд средств для создания и преобразования текстур в сжатые S3TC-форматы.

Наиболее удобным является плагин для Adobe Photoshop, позволяющий сохранять изображения в файлы DDS с различными режимами сжатия, включая создание всех необходимых *mipmap*-уровней. Для установки данного плагина достаточно просто скопировать файл dds.8bi в подкаталог Plug-Ins каталога, где у вас установлен Adobe Photoshop.

В ряде случаев более удобными оказываются утилиты, запускаемые из командной строки. В состав NVSDK входят две подобные утилиты: nvdxt.exe, позволяющая преобразовывать изображения из форматов TGA, BMP, GIF, JPG, PPM, TIF и CEL в формат DDS с заданием ряда параметров сохранения, и readdxt.exe — для преобразования файлов из формата DDS в TGA.

Утилита nvdxt.exe запускается командой

```
nvdxt -file имя-файла [опции]
```

При помощи параметра `-file` задается исходный файл, набор опций управляет созданием DDS-файла.

В табл. 14.8 приведены некоторые дополнительные опции этой команды.

Таблица 14.8. *Основные опции команды* `nvdxt`

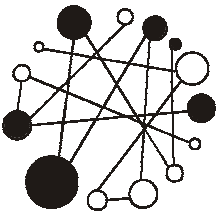
Опция	**Описание**
`-flip`	Перевернуть изображение сверху вниз
`-list <filename>`	Файл, содержащий список файлов для преобразования
`-cubemap <filename>`	Создать кубическую карту с именем `filename`. Далее должна идти опция `list`, задающая файл со списком текстур для граней куба в следующем порядке: положительное и отрицательное направления *X*, то же для *Y*, то же для *Z*
`-all`	Преобразовать все изображения в текущем каталоге
`-outdir <directory>`	Задать каталог для помещения преобразованных файлов
`-binaryalpha`	Считать, что α принимает лишь значения 0 и 1
`-dxt1c`	Использовать сжатие RGB DXT1
`-dxt1a`	То же RGBA DXT1
`-dxt3`	То же RGBA DXT3
`-dxt5`	То же RGBA DXT5
`-u1555`	Сохранить в несжатый формат 1:5:5:5
`-u4444`	То же 4:4:4:4
`-u565`	То же 5:6:5
`-u8888`	То же 8:8:8:8
`-u888`	То же 8:8:8
`-u555`	То же 5:5:5
`-p8`	Сохранить в несжатый палитровый формат

Формат вызова утилиты readdxt.exe очень прост:

```
readdxt имяфайла
```

Результатом является преобразование файла DDS, заданного параметром `имяфайла`, в TGA.

Глава 15



Вершинные программы и работа с ними через расширения ARB_vertex_program

Ранее рассматриваемые расширения позволяли задавать довольно сложные законы рендеринга примитивов, но при этом сами эти законы задавались фактически путем комбинации небольшого числа предопределенных шагов (операций), и выйти за их пределы было очень затруднительно.

Совершенно другой подход предлагают так называемые вершинные (*vertex*) и фрагментные (*fragment*) программы. Они представляют собой полноценные программы, написанные на специальном ассемблероподобном языке, которые выполняются непосредственно на GPU. С помощью таких программ можно выполнять очень сложные вычисления, практически невозможные для других способов.

Отличие между двумя основными типами таких программ (иногда называемых шейдерами) — вершинными (*vertex program* или *vertex shader*) и фрагментными (пиксельными) (*fragment program* или *fragment shader*) — заключается в виде обрабатываемых ими элементов.

Вершинные программы применяются для обработки данных, заданных на уровне отдельных вершин, в то время как фрагментные работают на уровне отдельных пикселов.

В данной главе мы рассмотрим использование вершинных программ через расширение ARB_vertex_program.

В стандартном OpenGL имеется набор заранее определенных операций над данными, задаваемыми в вершинах, например, преобразование вершин, вычисление и преобразование текстурных координат, освещение и т. п. Большинство из этих операций могут конфигурироваться программистом.

Фактически все эти операции представляют собой вершинную программу, однако в ней выполняемые операции и их порядок заранее заданы и зависят от текущего состояния OpenGL, а не явно задаются программистом. Расши-

рение ARB_vertex_program предоставляет пользователю возможность явно задавать закон преобразования данных в вершинах при помощи последовательности элементарных операций (команд).

Подобные вершинные программы работают с набором регистров (4-мерных вещественных векторов), и каждая команда осуществляет какую-либо операцию над этими регистрами. Сама программа представляет собой последовательность действий над 4-мерными векторами и задает закон определения выходных значений для каждой вершины на основе параметров (задаваемых вне блока `glBegin/glEnd`) и набора атрибутов в вершине.

Стандартный конвейер (*pipeline*) OpenGL представлен на рис. 15.1.

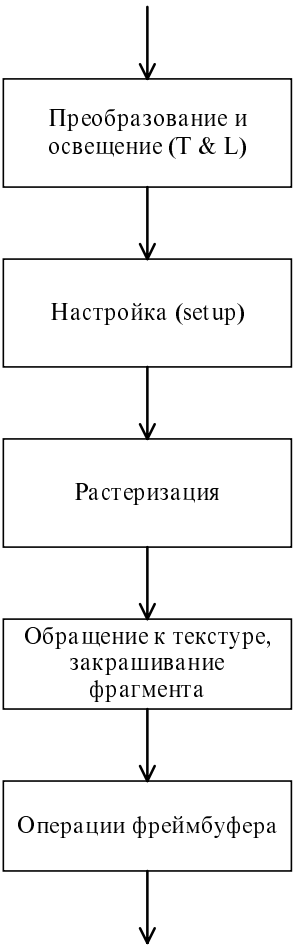


Рис. 15.1. Стандартный конвейер OpenGL

Применение вершинных программ позволяет заменить первый блок (преобразование и освещение) на операции, явно задаваемые пользователем. Эти программы дают полный контроль над операциями преобразования, освещения, вычисления текстурных координат, а также дают возможность выполнять другие типы вычислений в вершинах. При этом все вершинные программы полностью выполняются на GPU, не занимая времени центрального процессора.

Входными данными для вершинной программы являются атрибуты в вершинах (пользователь может задавать новые типы атрибутов), локальные переменные (регистры) и переменные окружения.

На выходе вершинной программы должны быть вычислены однородные координаты вершины, цвет, степень затуманивания, текстурные координаты и т. п.

При этом вершинная программа не может ни создавать новых вершин, ни уничтожать существующие. Также вершинная программа не обладает доступом к какой-либо топологической информацией (ребра, грани и т. п.).

Обычно вершинная программа заменяет собой следующие функции OpenGL:

- преобразование вершин при помощи модельной (*modelview*) матрицы и матрицы проектирования (*projection*);
- смешение (*weighting/blending*) вершин;
- преобразование нормали;
- вершинное освещение;
- употребление цветных материалов;
- вычисление текстурных координат и их преобразование;
- вычисление степени затуманивания вершины;
- выполнение операций, задаваемых пользователем.

Следующие функции не могут быть подменены вершинной программой:

- вычисления (*evaluators*);
- отсечение по области видимости (*view frustrum*);
- перспективное деление;
- преобразование в окно (*viewport transformation*);
- преобразование диапазона глубины;
- выбор лицевой и нелицевой граней (для двусторонних материалов);
- отсечение первичного и вторичного цветов по отрезку [0, 1];
- работа с примитивами, растеризация, смешение цветов.

На рис. 15.2 изображена вершинная программа и различные регистры, с которыми она работает.

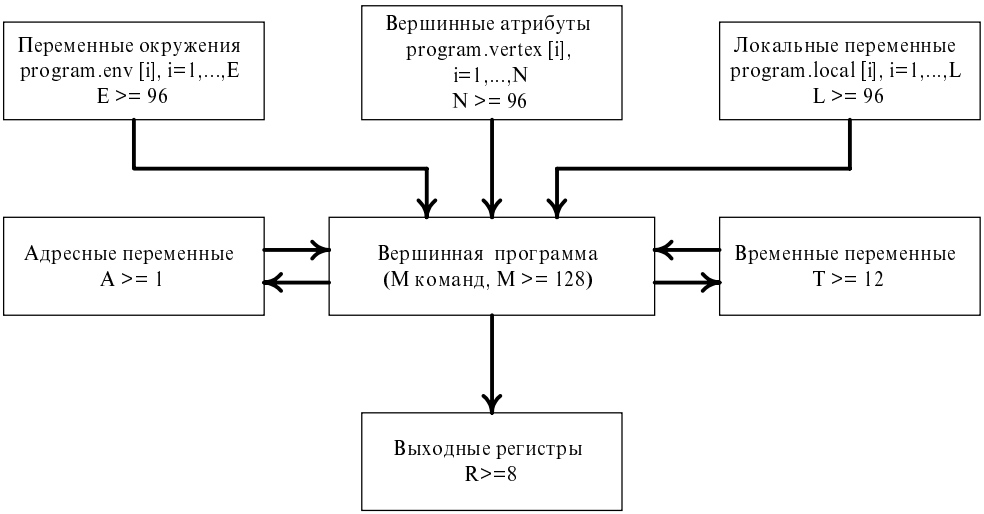


Рис. 15.2. Вершинная программа и используемые ею регистры

Создание вершинной программы

Каждая вершинная программа представляется набором байтов (ASCII-строкой). Работа с ней сильно напоминает работу с текстурами, вводится понятие объекта-программы (*program object*), идентифицируемого беззнаковым целым числом (`GLuint`). Этот объект является внутренним объектом OpenGL, инкапсулирующим внутри себя как программу, так и параметры ее состояния.

Непосредственное создание вершинной программы состоит из следующих шагов:

1. Получение идентификатора для программы при помощи функции `glGenProgramsARB`.
2. Выбор полученного идентификатора как текущего при помощи функции `glBindProgramARB`.
3. Загрузка текста программы.

Эти шаги продемонстрированы листингом 15.1.

Листинг 15.1. Пример использования вершинной программы

```
const char * vpText =
    "!!ARBvp1.0\
     ATTRIB pos     = vertex.position;\
     PARAM  mat [4] = { state.matrix.mvp };\
     # transform by concatenation of modelview and projection matrices\
     DP4 result.position.x, mat [0], pos;\
     DP4 result.position.y, mat [1], pos;\
     DP4 result.position.z, mat [2], pos;\
     DP4 result.position.w, mat [3], pos;\
     # copy primary color\
     MOV result.color, vertex.color;\
     END";
        GLuint       progId;
    glGenProgramsARB   ( 1, &progId );
    glBindProgramARB   ( GL_VERTEX_PROGRAM_ARB, progId );
    glProgramStringARB ( GL_VERTEX_PROGRAM_ARB,
                         GL_PROGRAM_FORMAT_ASCII_ARB,
                         strlen ( vpText ), vpText );
    if ( glGetLastError () == GL_INVALID_OPERATION )
    {
        GLint     errorPos;
        GLubyte * errorStr;
        glGetIntegerv ( GL_PROGRAM_ERROR_POSITION_ARB, &errorPos );
        errorString = glGetString ( GL_PROGRAM_ERROR_STRING_ARB );
        fprintf ( stderr, "Error at position %d.\Line: \"%s\"",
                  errorPos, errorStr );
    }
```

После того как вершинная программа больше не нужна, она может быть уничтожена при помощи функции `glDeleteProgramsARB`.

Рассмотрим подробнее использованные функции.

Для получения идентификаторов вершинных программ служит функция

```
void glGenProgramsARB ( GLsizei n, GLuint * ids );
```

Она резервирует `n` идентификаторов в качестве идентификаторов вершинных программ и помещает их в массив `ids`. При этом, хотя возвращенные

идентификаторы и помечаются как использованные, соответствующего объекта-программы еще не создается, он будет создан при первом выборе идентификатора как текущей программы при помощи `glBindProgramARB`.

Для уничтожения вершинных программ (и освобождения соответствующих идентификаторов) служит функция

```
void glDeleteProgramsARB ( GLsizei n, GLuint * ids );
```

Для выбора данной вершинной программы как текущей (и создания объекта-программы при первом обращении) служит функция

```
void glBindProgramARB ( GLenum target, GLuint id );
```

При задании вершинной программы параметр `target` должен принимать значения `GL_VERTEX_PROGRAM_ARB`, а `id` — идентификатор программы, полученный от `glGenProgramsARB`.

Проверить, является ли заданное беззнаковое целое число идентификатором программы, можно при помощи функции

```
GLboolean glIsProgramARB ( GLuint progId );
```

Узнать идентификатор вершинной программы, которая сейчас является активной, можно при помощи следующего фрагмента кода:

```
glGetProgramivARB ( target, GL_PROGRAM_BINDING_ARB, (int *)&id );
```

Для задания текста вершинной программы служит функция

```
void glProgramStringARB ( GLenum target, GLenum format, GLsizei length,
                          const void * string );
```

Параметр `target` для задания текста вершинной программы должен принимать значение `GL_VERTEX_PROGRAM_ARB`, `format` может принимать пока только одно значение `GL_PROGRAM_FORMAT_ASCII_STRING_ARB`. Параметр `length` задает длину строки (в байтах) `string`, которая задает текст программы. Длина считается без нулевого терминатора, которого вообще может не быть.

В соответствии со стандартом можно загружать программу, даже если она уже была загружена и выбрана как текущая (таким образом можно прямо "на ходу" загрузить новую версию программы).

Одна и та же вершинная программа может разделяться сразу несколькими контекстами точно так же, как и дисплейные списки и текстуры, т. е. при помощи команды `wglShareLists (hRcTo, hRcFrom)` (аналогично тому, как это рассматривалось в *главе 11*).

Задание параметров

Все параметры, которые могут быть переданы вершинной программе, делятся на три типа:

- Вершинные атрибуты (*vertex attributes*).
- Локальные параметры программы (*local parameters*).
- Параметры окружения (*environment parameters*).

Вершинные атрибуты

Это параметры, задаваемые в каждой вершине и представляющие собой 4-мерные вещественные векторы. Всего может быть до N таких параметров ($N \geq 16$), точно максимальное число вершинных атрибутов можно получить при помощи функции `glGetProgramivARB` по значению константы `GL_MAX_VERTEX_ATTRIBS_ARB`:

```
int   maxAttribs;
glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB, GL_MAX_VERTEX_ATTRIBS_ARB,
                    &maxAttribs );
```

Для задания вершинных атрибутов в виде вещественных (*float*) 3- и 4-мерных векторов служат следующие функции:

```
void glVertexAttrib4fARB  ( GLuint index, float x, float y, float z,
                            float w );
void glVertexAttrib4fvARB ( GLuint index, const float * vector );
void glVertexAttrib3fARB  ( GLuint index, float x, float y, float z );
void glVertexAttrib3fvARB ( GLuint index, const float * vector );
```

Существуют версии этой команды и для меньшего числа аргументов, и для других типов передаваемых аргументов по аналогии с командами `glVertex` и `glColor`.

Также есть возможность передачи вершинных атрибутов в виде массивов при помощи функции

```
void glVertexAttribPointerARB ( GLuint index, GLint size,
                                GLenum type,  GLboolean normalized,
                                GLsizei stride, const void * pointer );
```

Параметр `normalized` сообщает, нужно ли проводить нормализацию переданного значения.

При помощи функций

```
void glEnableVertexAttribArrayARB  ( GLuint index );
void glDisableVertexAttribArrayARB ( GLuint index );
```

можно включать и отключать поддержку отдельных вершинных массивов.

Запись вершинного атрибута с индексом 0 вызывает выполнение вершинной программы для обработки данной вершины.

Из самой программы можно обращаться к вершинным атрибутам как по индексу, так и по именам стандартных атрибутов. Соответствие между индексами и именами приводится в табл. 15.1.

Таблица 15.1. *Соответствие между индексами и именами вершинных атрибутов*

Имя	Индекс
`vertex.position`	`vertex.attrib [0]`
`vertex.weight`	`verter.attrib [1]`
`vertex.weight [0]`	`verter.attrib [1]`
`vertex.normal`	`verter.attrib [2]`
`vertex.color`	`verter.attrib [3]`
`vertex.color.primary`	`verter.attrib [3]`
`vertex.color.secondary`	`verter.attrib [4]`
`vertex.fogcoord`	`vertex.attrib [5]`
`vertex.texcoord`	`verter.attrib [8]`
`vertex.texcoord [0]`	`verter.attrib [8]`
`vertex.texcoord [n]`	`verter.attrib [8+n]`

Обратите внимание на то, что по стандарту приведенные пары содержат значения, которые не могут встречаться одновременно, например, если в программе есть обращение сразу к `vertex.position` и `vertex.attrib [0]`, то это приведет к ошибке при загрузке данной программы.

Задание вершинного атрибута с индексом 0 (включая и задание посредством команды `glVertex*`) вызывает выполнение программы для данной вершины.

Локальные параметры

Каждая программа в качестве локальных параметров получает в свое распоряжение массив из L 4-мерных вещественных векторов. Точно узнать максимальное число локальных параметров (оно должно быть не более 96) можно при помощи следующего фрагмента кода:

```
int   maxLocalParams;

    glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_PROGRAM_LOCAL_PARAMETERS_ARB,
                              &maxLocalParams );
```

При этом локальные параметры сохраняют свои значения не только между вызовами вершинной программы, но даже при ее перезагрузке.

Для задания локальных параметров служат следующие функции:

```
void glProgramLocalParameter4fARB  ( GLenum target, GLuint index,
                                     float x, float y,
                                     float z, float w );
void glProgramLocalParameter4fvARB ( GLenum target, GLuint index,
                                     const float * vector );
```

В качестве параметра `target` во всех этих функциях выступает `GL_VERTEX_PROGRAM_ARB`.

Из самой вершинной программы локальные программы доступны по имени `program.local[index]`.

Узнать значение локального параметра можно при помощи функции

```
void glGetProgramLocalParameterfvARB ( GLenum target, GLuint index,
                                       float * params );
```

В качестве параметра `target` по-прежнему выступает константа `GL_VERTEX_PROGRAM_ARB`, параметр `params` указывает на массив с достаточным местом для размещения четырех вещественных чисел.

Параметры окружения

Кроме вершинных атрибутов, задаваемых в каждой вершине, и локальных параметров, уникальных для каждой вершинной программы, вершинная программа также получает с свое распоряжение массив из E 4-мерных вещественных векторов, так называемых параметров окружения. Этот набор параметров является общим для всех вершинных программ.

Максимальное число параметров окружения (не менее 96) можно узнать при помощи следующего фрагмента кода:

```
int   maxEnvParams;
glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                    GL_MAX_PROGRAM_ENV_PARAMETERS_ARB,
                    &maxEnvParams );
```

Для задания значений параметров окружения служат следующие функции:

```
void glProgramEnvParameter4fARB  ( GLenum target, GLuint index,
                                   float x, float y,
                                   float z, float w );
void glProgramEnvParameter4fvARB ( GLenum target, GLuint index,
                                   const float * vector );
```

Из самой вершинной программы локальные программы доступны по имени `program.env[index]`.

Узнать значение параметра окружения с заданным индексом можно при помощи функции

```
void glGetProgramEnvParameterfvARB ( GLenum target, GLuint index,
                                     float * params );
```

Параметры здесь аналогичны `void glGetProgramLocalParameterfvARB`.

Параметры состояния

Из вершинной программы доступны также все основные параметры состояния OpenGL. В табл. 15.2—15.8 приводятся основные параметры и имена для доступа к ним.

В табл. 15.2 приводятся все доступные вершинной программе свойства материала, включая тип получаемого значения и способ доступа.

Таблица 15.2. *Свойства материала*

Имя	Компоненты	Комментарий
`state.material.ambient`	(*r*, *g*, *b*, *a*)	Фоновый цвет материала для лицевой грани
`state.material.diffuse`	(*r*, *g*, *b*, *a*)	Диффузный цвет материала для лицевой грани
`state.material.specular`	(*r*, *g*, *b*, *a*)	Бликовый цвет материала для лицевой грани
`state.material.emission`	(*r*, *g*, *b*, *a*)	Цвет свечения материала для лицевой грани
`state.material.shininess`	(*s*, 0, 0, 1)	Степень Фонга материала для лицевой грани
`state.material.front.ambient`	(*r*, *g*, *b*, *a*)	Фоновый цвет материала для лицевой грани
`state.material.front.diffuse`	(*r*, *g*, *b*, *a*)	Диффузный цвет материала для лицевой грани
`state.material.front.specular`	(*r*, *g*, *b*, *a*)	Бликовый цвет материала для лицевой грани
`state.material.front.emission`	(*r*, *g*, *b*, *a*)	Цвет свечения материала для лицевой грани
`state.material.front.shininess`	(*s*, 0, 0, 1)	Степень Фонга материала для лицевой грани

Таблица 15.2 *(окончание)*

Имя	Компоненты	Комментарий
`state.material.back.ambient`	(*r*, *g*, *b*, *a*)	Фоновый цвет материала для нелицевой грани
`state.material.back.diffuse`	(*r*, *g*, *b*, *a*)	Диффузный цвет материала для нелицевой грани
`state.material.back.specular`	(*r*, *g*, *b*, *a*)	Бликовый цвет материала для нелицевой грани
`state.material.back.emission`	(*r*, *g*, *b*, *a*)	Цвет свечения материала для нелицевой грани
`state.material.back.shininess`	(*s*, 0, 0, 1)	Степень Фонга материала для нелицевой грани

В табл. 15.3 приведены все доступные вершинной программе параметры источников света.

Таблица 15.3. *Параметры источников света*

Имя	Компоненты	Комментарий
`state.light[n].ambient`	(*r*, *g*, *b*, *a*)	Фоновый цвет источника света *n*
`state.light[n].diffuse`	(*r*, *g*, *b*, *a*)	Диффузный цвет источника света *n*
`state.light[n].specular`	(*r*, *g*, *b*, *a*)	Бликовый цвет источника света *n*
`state.light[n].position`	(*x*, *y*, *z*, *w*)	Координаты источника света *n*
`state.light[n].attenuation`	(*a*, *b*, *c*, *e*)	Коэффициент затухания в зависимости от расстояния и показатель для конического источника света *n*
`state.light[n].spot.direction`	(*x*, *y*, *z*, *c*)	Направление для конического источника света и косинус его угла (для источника света *n*)
`state.light[n].half`	(*x*, *y*, *z*, 1)	Нормализованный серединный угол между направлением от глаза на источник света и вектором (0, 0, 1)
`state.lightmodel.ambient`	(*r*, *g*, *b*, *a*)	Фоновый цвет модели освещения
`state.lightmodel.scenecolor`	(*r*, *g*, *b*, *a*)	Цвет сцены в модели освещения для лицевых граней

Таблица 15.3 *(окончание)*

Имя	Компоненты	Комментарий
`state.lightmodel.front.scenecolor`	(*r*, *g*, *b*, *a*)	То же
`state.lightmodel.back.scenecolor`	(*r*, *g*, *b*, *a*)	Цвет сцены в модели освещения для нелицевых граней
`state.lightprod[n].ambient`	(*r*, *g*, *b*, *a*)	Произведение фонового цвета источника света и материала
`state.lightprod[n].diffuse`	(*r*, *g*, *b*, *a*)	Произведение диффузного цвета источника света и материала
`state.lightprod[n].specular`	(*r*, *g*, *b*, *a*)	Произведение бликового цвета источника света и материала
`state.lightprod[n].front.ambient`	(*r*, *g*, *b*, *a*)	Произведение фонового цвета источника света и материала для лицевой грани
`state.lightprod[n].front.diffuse`	(*r*, *g*, *b*, *a*)	Произведение диффузного цвета источника света и материала для лицевой грани
`state.lightprod[n].front.specular`	(*r*, *g*, *b*, *a*)	Произведение бликового цвета источника света и материала для лицевой грани
`state.lightprod[n].back.ambient`	(*r*, *g*, *b*, *a*)	Произведение фонового цвета источника света и материала для нелицевой грани
`state.lightprod[n].back.diffuse`	(*r*, *g*, *b*, *a*)	Произведение диффузного цвета источника света и материала для нелицевой грани
`state.lightprod[n].back.specular`	(*r*, *g*, *b*, *a*)	Произведение бликового цвета источника света и материала для нелицевой грани

В табл. 15.4 приведены параметры для генерации текстурных координат, в табл. 15.5 — для вычисления затуманивания.

Таблица 15.4. *Параметры генерации текстурных координат*

Имя	Компоненты	Комментарий
`state.texgen[n].eye.s`	(*a*, *b*, *c*, *d*)	Коэффициенты для получения *s*-координаты для текстурного блока *n* через координаты в системе координат наблюдателя
`state.texgen[n].eye.t`	(*a*, *b*, *c*, *d*)	То же для *t*-координаты
`state.texgen[n].eye.r`	(*a*, *b*, *c*, *d*)	То же для *r*-координаты

Таблица 15.4 *(окончание)*

Имя	Компоненты	Комментарий
`state.texgen[n].eye.q`	(*a*, *b*, *c*, *d*)	То же для *q*-координаты
`state.texgen[n].object.s`	(*a*, *b*, *c*, *d*)	Коэффициенты для получения *s*-координаты для текстурного блока *n* через координаты в системе координат объекта
`state.texgen[n].object.t`	(*a*, *b*, *c*, *d*)	То же для *t*-координаты
`state.texgen[n].object.r`	(*a*, *b*, *c*, *d*)	То же для *r*-координаты
`state.texgen[n].object.q`	(*a*, *b*, *c*, *d*)	То же для *q*-координаты

Таблица 15.5. *Параметры вычисления затуманивания*

Имя	Компоненты	Комментарий
`state.fog.color`	(*r*, *g*, *b*, *a*)	RGB-цвет тумана
`state.fog.params`	(*d*, *s*, *e*, *r*)	Плотность тумана (*d*), линейное начало и конец (*s*, *e*) и *1/*(*end-start*)

В табл.15.6 приведены параметры плоскостей отсечения.

Таблица 15.6. *Параметры плоскостей отсечения*

Имя	Компоненты	Комментарий
`state.clip[n].plane`	(*a*, *b*, *c*, *d*)	Коэффициенты плоскости отсечения *n*. Плоскость задается уравнением $a \cdot x + b \cdot y + c \cdot z + d = 0$

В табл. 15.7 приведены свойства точки, задаваемые при помощи расширения ARB_point_parameters.

Таблица 15.7. *Параметры точки*

Имя	Компоненты	Комментарий
`state.point.size`	(*s*, *n*, *x*, *f*)	Параметры точки: размер, минимальный и максимальный размер, граница ослабевания (*fade threshold*)

Таблица 15.7 *(окончание)*

Имя	Компоненты	Комментарий
`state.point.attenuation`	(*a*, *b*, *c*, 1)	Коэффициенты ослабевания точки

Матрицы, доступные программе, перечислены в табл. 15.8.

Таблица 15.8. *Доступные матрицы*

Имя	Комментарий
`state.matrix.modelview[n]`	*n*-я матрица модельного преобразования. Параметр *n* необязателен
`state.matrix.projection`	Матрица проектирования
`state.matrix.mvp`	Произведение матрицы модельного преобразования и проектирования: MVP = P×M[0]
`state.matrix.texture[n]`	*n*-я матрица преобразования текстурных координат. Параметр *n* необязателен
`state.matrix.palette[n]`	Палитровая матрица *n*-го модельного преобразования
`state.matrix.program[n]`	*n*-я матрица программы

К имени матрицы из табл. 15.8 может быть добавлено `.inverse`, `.transpose` и `.invtrans` для обозначения того, что указанную матрицу следует обратить, транспонировать или обратить и транспонировать.

Также к имени может быть добавлена запись `.row [n]` для получения заданного столбца матрицы в виде 4-мерного вектора.

Запись `state.matrix.program[n]` используется для доступа к дополнительным матрицам. Из основной программы способ задания этих матриц аналогичен заданию матрицы модельного преобразования, проектирования и преобразования текстурных координат: вызывается функция `glMatrixMode` с параметром, равным `GL_MATRIXn_ARB`, где числом `n` является номер матрицы.

Для получения максимально возможного числа таких матриц можно выполнить следующий фрагмент кода

```
int   maxMatrices;
glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                    GL_MAX_PROGRAM_MATRICES_ARB,
                    &maxMatrices );
```

Результаты выполнения вершинная программа заносит в выходные регистры, приведенные в табл. 15.9.

Таблица 15.9. *Регистры результата*

Имя	Компоненты	Комментарий
`result.position`	(*x*, *y*, *z*, *w*)	Координаты точки в пространстве отсечения
`result.color`	(*r*, *g*, *b*, *a*)	Главный (*primary*) цвет лицевой грани
`result.color.primary`	(*r*, *g*, *b*, *a*)	То же
`result.color.secondary`	(*r*, *g*, *b*, *a*)	Дополнительный (*secondary*) цвет лицевой грани
`result.color.front`	(*r*, *g*, *b*, *a*)	Главный (*primary*) цвет лицевой грани
`result.color.front.primary`	(*r*, *g*, *b*, *a*)	То же
`result.color.front.secondary`	(*r*, *g*, *b*, *a*)	Дополнительный (*secondary*) цвет лицевой грани
`result.color.back`	(*r*, *g*, *b*, *a*)	Главный (*primary*) цвет нелицевой грани
`result.color.back.primary`	(*r*, *g*, *b*, *a*)	То же
`result.color.back.secondary`	(*r*, *g*, *b*, *a*)	Дополнительный (*secondary*) цвет нелицевой грани
`result.fogcoord`	(*f*, *, *, *)	Затуманивание (*for coordinate*). Компоненты, обозначенные '*', не используются
`result.pointsize`	(*s*, *, *, *)	Размер точки. Компоненты, обозначенные '*', не используются
`result.texcoord`	(*s*, *t*, *r*, *q*)	Текстурная координата для нулевого текстурного блока
`result.texcoord [n]`	(*s*, *t*, *r*, *q*)	Текстурная координата для *n*-го текстурного блока

В листинге 15.2 приводится программа, печатающая информацию о константах, ограничивающих количество регистров, параметров, временных переменных и т. п.

Листинг 15.2. Определение поддержки вершинных программ (программа vp-info)

```
//
// Sample to check for vertex 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
int    win = glutCreateWindow ( "OpenGL VP 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_vertex_program" ) )
{
    printf("GL_ARB_vertex_program extension NOT supported.\n");
    return 1;
}
    printf ( "ARB_vertex_program extension is supported !\n" );
    int        maxVertexAttribs;
    int        maxLocalParams;
    int        maxEnvParams;
    int        maxMatrices;
    int        maxTemporaries;
    int        maxParams;
    int        maxAddressRegs;
    glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_VERTEX_ATTRIBS_ARB,
                        &maxVertexAttribs );
    glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_PROGRAM_LOCAL_PARAMETERS_ARB,
                        &maxLocalParams   );
    glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_PROGRAM_ENV_PARAMETERS_ARB,
                        &maxEnvParams     );
    glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
```

```
                        GL_MAX_PROGRAM_MATRICES_ARB,
                        &maxMatrices      );
        glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_PROGRAM_TEMPORARIES_ARB,
                        &maxTemporaries   );
        glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_PROGRAM_PARAMETERS_ARB,
                        &maxParams        );
        glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_PROGRAM_ADDRESS_REGISTERS_ARB,
                        &maxAddressRegs   );
    printf ( "\nVertex program limits:\n"                        );
    printf ( "\tmax vertex attributes  : %d\n", maxVertexAttribs );
    printf ( "\tmax local parameters   : %d\n", maxLocalParams   );
    printf ( "\tmax env. parameters    : %d\n", maxEnvParams     );
    printf ( "\tmax program matrices   : %d\n", maxMatrices      );
    printf ( "\tmax program temporaries: %d\n", maxTemporaries   );
    printf ( "\tmax parameters         : %d\n", maxParams        );
    printf ( "\tmax address registers  : %d\n", maxAddressRegs   );
    return 0;
}
```

Структура вершинной программы

Каждая вершинная программа представляет собой набор строк, причем каждой строке соответствует какая-то команда (строка может быть пустой или содержать комментарий). В листинге 15.3 приводится пример вершинной программы вычисляющей фоновое, диффузное и бликовое освещение от одного бесконечно удаленного источника света.

Листинг 15.3. Пример вершинной программы

```
!!ARBvp1.0
ATTRIB   iPos        = vertex.position;
ATTRIB   iNormal     = vertex.normal;
PARAM    mvinv[4]    = { state.matrix.modelview.invtrans };
PARAM    mvp[4]      = { state.matrix.mvp };
PARAM    lightDir    = state.light[0].position;
PARAM    halfDir     = state.light[0].half;
```

```
PARAM    specExp     = state.material.shininess;
PARAM    ambientCol  = state.lightprod[0].ambient;
PARAM    diffuseCol  = state.lightprod[0].diffuse;
PARAM    specularCol = state.lightprod[0].specular;
OUTPUT   oPos        = result.position;
OUTPUT   oColor      = result.color;
TEMP     xfNormal, temp, dots;
              # Transform the vertex to clip coordinates.
DP4 oPos.x, mvp [0], iPos;
DP4 oPos.y, mvp [1], iPos;
DP4 oPos.z, mvp [2], iPos;
DP4 oPos.w, mvp [3], iPos;
              # Transform the normal to eye coordinates.
DP3 xfNormal.x, mvinv [0], iNormal;
DP3 xfNormal.y, mvinv [1], iNormal;
DP3 xfNormal.z, mvinv [2], iNormal;
              # Compute diffuse and specular dot products and
              # use LIT to compute lighting coefficients.
DP3 dots.x, xfNormal, lightDir;
DP3 dots.y, xfNormal, halfDir;
MOV dots.w, specExp.x;
LIT dots, dots;
              # Accumulate color contributions.
MAD temp, dots.y, diffuseCol, ambientCol;
MAD oColor.xyz, dots.z, specularCol, temp;
MOV oColor.w, diffuseCol.w;
END
```

Первая строка программы `!!ARBvp1.0` сообщает, что далее идет вершинная программа, соответствующая версии 1.0.

Каждая команда (кроме команды `END`) должна завершаться точкой с запятой.

Завершается вершинная программа командой `END`.

Весь текст, идущий после символа `#` и до конца строки, является комментарием и игнорируется.

Обратите внимание на то, что все имена команд, ключевые слова, используемые при объявлении переменных, а также `vertex`, `state`, `program` и `result`, являются зарезервированными.

В начале вершинной программы при помощи команды OPTION могут задаваться различные опции. В версии 1.0 единственной поддерживаемой опцией является ARB_position_invariant.

Ее наличие связано с тем, что при многопроходном рендеринге объектов на разных стадиях могут вызываться различные вершинные программы, выполняющиеся с различной точностью. В результате этого координаты вершин на разных проходах могут слегка отличаться, что крайне нежелательно.

Наличие опции ARB_position_invariant гарантирует, что на всех проходах будут получаться одни и те же координаты вершин.

Идентификаторы

В качестве идентификаторов в вершинной программе служат произвольные последовательности латинских букв, цифр и символа подчеркивания _ и знака доллара $, начинающиеся не с цифры (например, Mu$t_Die является совершенно законным идентификатором).

Обратите внимание, что идентификаторы в вершинных программах чувствительны к регистрам букв, т. е. идентификаторы a1 и A1 различаются.

Каждый идентификатор переменной до его использования должен быть объявлен, само объявление может находиться в любом месте программы (но до первого упоминания данного идентификатора).

Временные переменные

Для хранения промежуточных значений в вершинных программах предусмотрены временные переменные — 4-мерные вещественные векторы, которые заранее должны быть объявлены командой TEMP.

```
TEMP flag;
TEMP a,b,c;
```

Все временные переменные доступны вершинной программе как для чтения, так и для записи. Максимально возможное их число можно узнать при помощи следующего фрагмента кода:

```
int        maxTemporaries;
glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                    GL_MAX_PROGRAM_TEMPORARIES_ARB,
                    &maxTemporaries );
```

Начальные значения временных переменных (как и выходных) не определены и зависят от реализации драйвера.

Параметры

В качестве параметров могут выступать 4-мерные векторы и их массивы.

Параметры могут задаваться как явно (через команду `PARAM`), так и неявно (путем непосредственной подстановки в текст). При этом они представляют собой постоянные (на время выполнения вершинной программы) величины, т. е. доступны только для чтения. Пример задания иллюстрирует листинг 15.4.

Листинг 15.4. Пример задания параметров

```
PARAM a = { 1, 2, 3, 4 };                  # vector (1, 2, 3, 4)
PARAM b = { 3 };                           # vector (3, 0, 0, 1)
PARAM c = { 3, 4 };                        # vector (3, 4, 0, 1)
PARAM e = 3;                               # vector (3, 3, 3, 3)
                                           # array of two vectors
PARAM arr [2] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 } };
ADD a, b, { 1, 2, 3, 4 };
ADD a, c, 3;
```

Матрица задается как массив из четырех векторов-столбцов. Рассмотрим это подробнее.

Пусть исходная матрица задавалась следующим фрагментом кода:

```
GLfloat matrix [16] =
{
   1, 5, 9,  13,
   2, 6, 10, 14,
   3, 7, 11, 15,
   4, 8, 12, 16
};
glMatrixMode  ( GL_MATRIX0_ARB );
glLoadMatrixf ( matrix         );
```

Также пусть в вершинной программе присутствуют следующие объявления:

```
PARAM mat1 [4] = { state.matrix.program [0] };
PARAM mat2 [4] = { state.matrix.program [0].transpose };
```

В OpenGL принято расположение матриц по столбцам, следовательно `mat1 [0]` принимает значение (1, 2, 3, 4), `mat1 [3]` — (13, 14, 15, 16), `mat2 [0]` — (1, 5, 9, 13), и `mat2 [3]` — (4, 8, 12, 16).

Вот еще несколько примеров параметров, основанных на матрицах:

```
PARAM m0    =   state.matrix.modelview [1].row [0];
PARAM m1    =   state.matrix.projection.transpose.row [3];
PARAM m2 [] = { state.matrix.program [0].row [1..2] };
PARAM m3 [] = { state.matrix.program [0].transpose };
```

Максимально возможное общее число всех параметров (явных и неявных) можно получить из следующего фрагмента кода:

```
int        maxParams;
glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                    GL_MAX_PROGRAM_PARAMETERS_ARB,
                    &maxParams );
```

Можно связать параметр с каким-либо локальным параметром или параметром окружения:

```
PARAM c       = program.local [0];
PARAM mat [4] = program.env   [0..3];
PARAM ambient = state.material.ambient;
```

Адресные переменные

Адресные переменные выступают фактически в качестве индексов при обращении к массивам векторов, представляя собой однокомпонентный целочисленный вектор.

Для объявления адресных переменных служит команда `ADDRESS`.

Листинг 15.5 иллюстрирует пример использования адресных переменных.

Листинг 15.5. Пример задания и использования адресных переменных

```
ADDRESS     addr;
PARAM     list [] = {
  { 1, 0, 0, 1 },
  { 2, 1, 0, 1 },
  { 3, 4, 5, 1 }
};
. . .
ARL addr.x, index.x;     # load into the address variable floor (index.x)
. . .
SUB    v, a, list [addr.x];
. . .
```

Для адресных переменных можно задавать смещение, лежащее в диапазоне от −64 до +63.

Атрибуты

При помощи команды ATTRIB можно явно описать атрибут, дав ему другое имя.

```
ATTRIB    pos      = vertex.position;
ATTRIB    normal   = vertex.normal;
ATTRIB    tangent  = vertex.texcoord [1];
ATTRIB    binormal = vertex.texcoord [2];
```

Подобная запись позволяет повысить читаемость и сразу показать, какими атрибутами пользуется данная программа.

Выходные значения

Выходные параметры доступны только для записи и задаются командой OUTPUT:

```
OUTPUT oCol = result.color.primary;
```

Фактически эта команда просто создает ссылку на соответствующий выходной регистр аналогично ATTRIB.

Можно создать второе имя для уже существующего параметра (alias):

```
ALIAS b = a;
```

Система команд

Всего поддерживается 27 различных инструкций, работающих с 4-мерными вещественными векторами. Каждая такая команда имеет следующий вид:

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

Здесь opCode — это 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. Далее представлены простейшие примеры команд.

Пример 1. Простейшие команды

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

Необязательный знак "минус" указывает, что источником является не сам регистр, а 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 (пример 2).

Пример 2. Перестановка компонент

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

Для выходного регистра можно задать маску, защищающую отдельные компоненты регистра от изменения (пример 3).

Пример 3. Маскирование

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

Обратите внимание, что команды в примере 4 эквивалентны.

Пример 4. Сокращенная запись операнда

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

Рассмотрим систему команд, приведенную в табл. 15.10.

Таблица 15.10. *Система команд вершинной программы*

Название	Синтаксис	Комментарий
ABS Вычисление модуля	`ABS dest, src;`	Данная команда осуществляет покомпонентное вычисление модуля `dest=fabs(src)`
ADD Сложение	`ADD dest, src0, src1;`	Данная команда складывает два параметра и записывает сумму в результат `dest=src0+src1`
ARL Загрузка значения в адресный регистр	`ARL dest.C1, src.C2;`	Загружает в адресный регистр значение целой части скалярного операнда
DP3 Вычисление скалярного произведения по первым трем компонентам	`DP3 dest, src0, src1;`	Вычисляет скалярное произведение источников как трехмерных векторов `dest=src0.x·src1.x+src0.y·src1.y+src0.z·src1.z`

Таблица 15.10 *(продолжение)*

Название	Синтаксис	Комментарий
DP4 Вычисление скалярного произведения	`DP4 dest, src0, src1;`	Вычисляет скалярное произведение четырехмерных векторов и записывает результат во все компоненты `dest` `dest=src0.x·src1.x+src0.y·src1.y+src0.z·src1.z+src0.w·src1.w`
DPH Вычисление однородного скалярного произведения	`DPH dest, src0, src1;`	Вычисляет однородное скалярное произведение `dest=src0.x·src1.x + src0.y·src1.y · src0.z·src1.z · src1.w`

DST
Вычисление расстояния
	`DST dest, src0.C1, src1.C2;`	Эффективно вычисляет вектор `dest=(1,d,d^2,1/d)` по двум скалярным значениям
		`src0.C1=d^2`
		`src1.C2=1/d`

EX2 Приближенное вычисление 2 в степени	`EX2 dest, src.C;`	Для заданного скалярного операнда `src.C` вычисляет приближенное значение `2^src.C` и записывает во все компоненты регистра `dest`

EXP
Вычисление 2 в степени
	`EXP dest, src.C;`	Служит для более точного вычисления `2^src.C`
		`dest.x=2^floor(src.C)`
		`dest.y=src.C - floor (src.C)`
		`dest.z=roughAppr2ToX(floor(scr.C))`
		`dest.w=1`
		Функция `roughAppr2ToX` вычисляет приближенное значение с точностью до 2^{-11}

FLR Вычисление `floor`	`FLR dest, src;`	Осуществляет покомпонентное вычисление целой части (*floor*) (наибольшего целого, не превосходящего аргумента) от компонентов источника
FRC Вычисление дробной части	`FRC dest, src;`	Осуществляет покомпонентное вычисление дробной части. Дробная часть `frac(x)=x-floor(x)`
LG2 Приближенное вычисление логарифма по основанию 2	`LG2 dest, src.C;`	Вычисляет приближенное значение логарифма по основанию 2 от скалярного аргумента и записывает во все компоненты источника

Таблица 15.10 *(продолжение)*

Название	Синтаксис	Комментарий

LIT
Вычисление коэффициентов освещения
	`LIT dest, src;`	Служит для ускорения вычисления диффузной и бликовой освещенности в вершинах
		`src.x=(n,l)`
		`src.y=(n,h)src.w = p`
		`p` приводится к отрезку [–128, 128]
		На выходе
		`dest.x=1`
		`dest.y=max(0,(n,l))`
		`dest.z=(n.l)>0?roughAppr2ToX(max(0,(n,h))^p):0`
		`dest.w=1`

LOG
Вычисление логарифма по основанию 2
	`LOG dest, src.C;`	Вычисляет приближенное значение 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;`	Вычисляет `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 Покомпонентное умножение	`MUL dest, src0, src1;`	Вычисляет покомпонентное произведение `dest=src0·src1`
POW Возведение в степень	`POW dest, src0.C1, src1.C2;`	Вычисляет приближенное значение первого скалярного операнда, возведенного в степень второго скалярного операнда и записывает результат во все компоненты регистра `dest` `apprPow(a,b)=apprExp2(b·apprLog2·a))`
RCP Вычисление обратного	`RCP dest, src.C;`	Вычисляет приближенное значение обратного к скалярному операнду и записывает во все компоненты `dest` `dest=1/src.C`

Таблица 15.10 *(окончание)*

Название	Синтаксис	Комментарий
RSQ Вычисление значения операнда в степени $-1/2$	`RSQ dest, src.C;`	`dest=1/sqrt(fabs(scr.C))`

SGE
Установка по "больше или равно"
	`SGE dest, src0, src1;`	`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`

SLT
Установка по "меньше чем"
	`SLT dest, src0, src1;`	`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 Вычисление разности	`SUB dest, src0, src1;`	Покомпонентное вычисление разности операндов `dest=src0 - src1`

SWZ
"перемешивание" компонент
	`SWZ dest, src, <extSwizzle>`	Позволяет произвольным образом перемешать компоненты (или их часть); источником для компоненты может быть любая компонента исходного регистра (со знаком или без) или же значения 0 и 1.
		Здесь
		`<extSwizzle> ::= <comp>, <comp>, <comp>, <comp>`
		`<comp> ::= [-] (0 | 1 | x | y | z | w)`

XPD Вычисление векторного произведения	`XPD dest, src0, src1;`	Осуществляет вычисление векторного произведения первых трех компонент первого операнда на первые три компоненты второго операнда и записывает в первые три компоненты `dest`

Примеры

Иллюстрация работы команды

```
MUL R1.xyz, R2, R3;
```

представлена на рис. 15.3.

На рис. 15.4 представлена иллюстрация работы команды

```
RCP R1, R2.w;
```

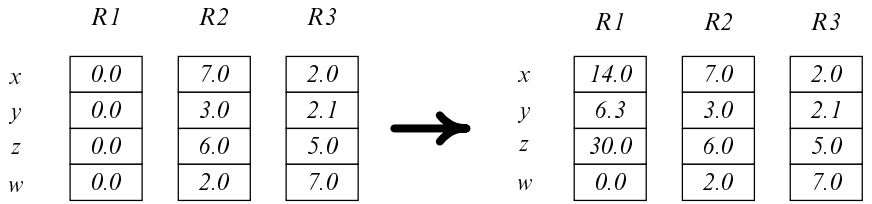


Рис. 15.3. Работа команды `MUL R1.xyz, R2, R3;`

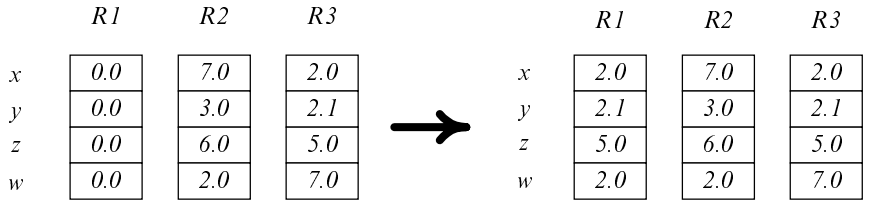


Рис. 15.4. Работа команды `RCP`

Аналогично, рис. 15.5 иллюстрирует работу команды `MIN`.

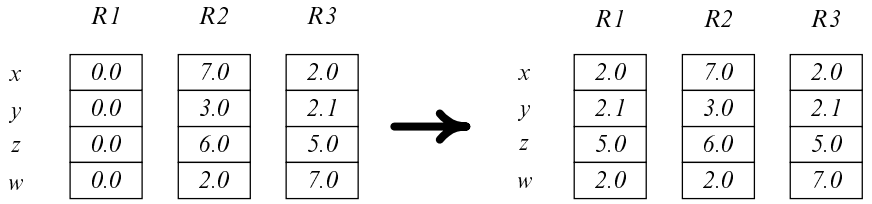


Рис. 15.5. Работа команды `MIN R1, R2, R3;`

Рисунок 15.6 иллюстрирует работу команды `MAD R1, R2, R3, R4;`

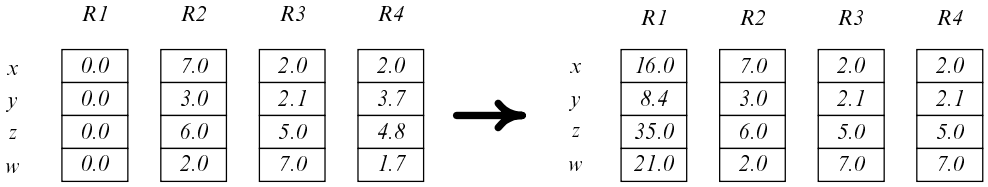


Рис. 15.6. Работа команды `MAD R1, R2, R3, R4;`

На рис. 15.7 изображена работа команды `DP3`.

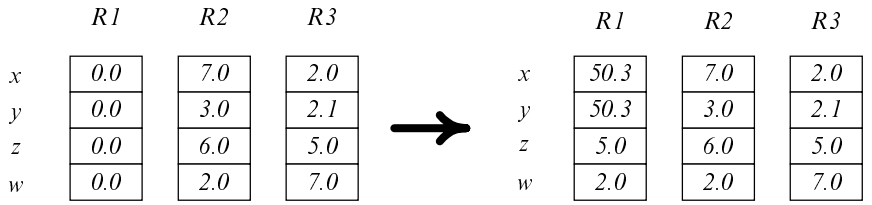


Рис. 15.7. Работа команды `DP3 R1.xy, R2, R3;`

Рассмотрим теперь реализацию нескольких простых вычислений при помощи вершинных программ.

Нормирование трехмерного вектора

Пусть у нас есть трехмерный вектор, который необходимо нормировать. При помощи вершинной программы это можно реализовать всего за три шага.

Сначала командой `DP3` определяется скалярный квадрат вектора (квадрат его длины). Потом команда `RSQ` вычисляет величину, обратную длине вектора (нормирующий множитель). Затем результаты перемножаются. Далее приведен соответствующий фрагмент кода.

```
DP3 result.w, vector, vector;       # result.w = nx^2+ny^2+nz^2
RSQ result.w, result.w;             # result.w = 1/sqrt(nx^2+ny^2+nz^2)
MUL result.xyz, result.w, vector;   # normalize by multiplying by 1/length
```

Примеры операций

Рассмотрим, как при помощи вершинной программы можно найти определитель матрицы 3×3.

Будем считать, что матрица задана первыми тремя компонентами трех векторов `vec0`, `vec1` и `vec2`. Далее представлены примеры, иллюстрирующие различные матричные операции.

Пример 5. Вычисление определителя матрицы

```
#
# Determinant of
# | vec0.x vec0.y vec0.z |
```

```
                    # | vec1.x vec1.y vec1.z |
                    # | vec2.x vec2.y vec2.z |
                    #
MUL result, vec1.zxyw, vec2.yzxw;
MAD result, vec1.yzxw, vec2.zxyw, -result;
DP3 result, vec0, result;
```

Пример 6. Линейная интерполяция двух векторов (lerp, mix)

```
                    # result = a * vec0 + (1-a) * vec1
                    # = vec1 + a * (vec0 - vec1)
SUB result, vec0, vec1;
MAD result, a, result, vec1;
```

Пример 7. Приведение (отсечение) величины в заданный отрезок [arg1, arg2]

```
            # result = min ( max ( var, arg1 ), arg2 )
MAX temp, var, arg1;
MIN result, temp, arg2;
```

Вот пример приведения угла (в радианах) в диапазон $[0, 2\pi]$.

Пример 8. Приведение угла к отрезку [0, 2*PI]

```
              # result = 2*PI * fraction(in/(2*PI))
              # piVec = (1/(2*PI), 2*PI, 0, 0)
PARAM piVec = { 0.159154943, 6.283185307, 0, 0 };
MUL   result, in, piVec.x;
EXP   result, result.x;
MUL   result, result.y, piVec.y;
```

Преобразование в пространство отсечения

Поскольку вершинная программа заменяет собой стандартный блок преобразования и освещения (T&L), то она обязательно должна выполнять преобразование координат в пространство отсечения (*clip space*).

Для этого необходимо умножить преобразуемый вектор на произведение матрицы модельного преобразования и проектирования. Удобно воспользо-

ваться тем, что это произведение доступно программе и его не надо явно вычислять.

Вычисление компонентов произведения матрицы на вектор легко реализуется через четырехмерное скалярное произведение `DP4` (листинг 15.6).

Листинг 15.6. Преобразование в пространство отсечения

```
!!ARBvp1.0
ATTRIB pos      = vertex.position;
PARAM  mat [4] = { state.matrix.mvp };
                    # transform by concatenation of modelview
                    # and projection matrices
DP4 result.position.x, mat [0], pos;
DP4 result.position.y, mat [1], pos;
DP4 result.position.z, mat [2], pos;
DP4 result.position.w, mat [3], pos;
```

Примеры вершинных программ

Вычисление необходимых параметров для попиксельного диффузного освещения

Когда мы рассматривали попиксельное диффузное освещение и механизм *register combiner*, то возникала необходимость вычисления вектора направления на источник света (l) для каждой вершины выводимого объекта и перевод его в систему координат касательного пространства.

Это как раз пример работы, которую легко может выполнить вершинная программа. Получив координаты источника света (в одном из локальных параметров), она может по координатам вершины и базису касательного пространства (векторам t, b и n) вычислить вектор на источник света (l) и сразу же перевести его в касательное пространство. При этом все вычисления будут производиться полностью на GPU; единственное, что должен делать центральный процессор, — это поставлять необходимые данные для каждой вершины.

Листинг 15.7 содержит исходный текст соответствующей вершинной программы.

Листинг 15.7. Вершинная программа для попиксельного диффузного освещения

```
!!ARBvp1.0
#
# simple vertex shader to setup data for per-pixel diffuse lighting
#
# on entry:
#      vertex.position
#      vertex.normal       - normal vector (n) of TBN basic
#      vertex.texcoord [0] - normal texture coordinates
#      vertex.texcoord [1] - tangent vector  (t)
#      vertex.texcoord [2] - binormal vector (b)
#
#      program.local [0] - eye position
#      program.local [1] - light position
#
# on exit:
#     result.texcoord [0] - texture coordinates
#     result.texcoord [1] - vector l
#
PARAM    eye     = program.local [0];
PARAM    light   = program.local [1];
PARAM    mvp [4] = { state.matrix.mvp };
TEMP     l, lt;
            # compute l (vector to light)
ADD    l, -vertex.position, light;
            # transform it into tangent space
DP3    lt.x, l, vertex.texcoord [1];
DP3    lt.y, l, vertex.texcoord [2];
DP3    lt.z, l, vertex.normal;
MOV    lt.w, l.w;
            # store it into texcoord [1]
MOV    result.texcoord [1], lt;
            # store 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, vertex.position, mvp [0];
DP4    result.position.y, vertex.position, mvp [1];
DP4    result.position.z, vertex.position, mvp [2];
DP4    result.position.w, vertex.position, mvp [3];
            # we're done
END
```

Как видно из листинга, сначала определяется вектор на источник света (как разность положений источника света и текущей вершины). При помощи трех трехмерных скалярных произведений получившийся вектор переводится в касательное пространство. Полученный вектор заносится в текстурные координаты для первого текстурного блока:

```
            # compute l (vector to light)
ADD    l, -vertex.position, light;
            # transform it into tangent space
DP3    lt.x, l, vertex.texcoord [1];
DP3    lt.y, l, vertex.texcoord [2];
DP3    lt.z, l, vertex.normal;
MOV    lt.w, l.w;
            # store it into texcoord [1]
MOV    result.texcoord [1], lt;
```

Программа должна также записать в `result.texcoord [0]` настоящие текстурные координаты (используемые для обращения к текстуре), значения основного (`result.color`) и дополнительного (`result.color.secondary`) цветов, а также координаты преобразованной вершины (`result.position`).

Листинг 15.8 содержит исходный код примера, иллюстрирующего работу данной вершинной программы.

Листинг 15.8. Вычисление попиксельного диффузного освещения при помощи вершинной программы

```
//
// Sample to to show diffuse lighting usinf Register Combiners and
// vertex programs support in OpenGL
//
#include    "libExt.h"
#include    <glut.h>
```

```
#include    <stdio.h>
#include    <stdlib.h>
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Torus.h"
#include    "Data.h"
Vector3D    eye   ( 7, 5, 7 );        // camera position
Vector3D    light ( 5, 0, 4 );        // light position
unsigned    normCubeMap;              // normalization cubemap id
unsigned    bumpMap;                  // normal map
unsigned    decalMap;                 // decal (diffuse) texture
unsigned    program;                  // vertex program id
float       angle = 0;
Torus       torus ( 1, 3, 30, 30 );
Vector3D    rot ( 0, 0, 0 );
int         mouseOldX = 0;
int         mouseOldY = 0;
////////////////////////////// methods ////////////////////////////
unsigned    loadProgram ( const char * fileName )
{
    Data    data ( fileName );
    if ( !data.isOk () || data.isEmpty () )
        return 0;
    unsigned    id;
    glGenProgramsARB ( 1, &id );
    glBindProgramARB ( GL_VERTEX_PROGRAM_ARB, id );
    glProgramStringARB ( GL_VERTEX_PROGRAM_ARB,
                         GL_PROGRAM_FORMAT_ASCII_ARB,
                         data.getLength (), data.getPtr () );
    if ( glGetError () == GL_INVALID_OPERATION )
    {
        int          errorCode;
        const char * errorString;
        glGetIntegerv ( GL_PROGRAM_ERROR_POSITION_ARB, &errorCode );
        errorString = (const char *)
                         glGetString ( GL_PROGRAM_ERROR_STRING_ARB );
        printf ( "Error loading program: \n%s\n", errorString );
        return 0;
```

```
    }
    return id;
}
void display ()
{
    float    lightColor   [4] = { 1, 1, 0, 1 };       // yellow light
    float    ambientColor [4] = { 0, 0, 0.5f, 1 };    // dark blue
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glMatrixMode   ( GL_MATRIX0_ARB );
    glLoadIdentity ();
    glRotatef    ( rot.x, 1, 0, 0 );
    glRotatef    ( rot.y, 0, 1, 0 );
    glRotatef    ( rot.z, 0, 0, 1 );
    glMatrixMode   ( GL_MODELVIEW );
    glPushMatrix   ();
    glRotatef    ( rot.x, 1, 0, 0 );
    glRotatef    ( rot.y, 0, 1, 0 );
    glRotatef    ( rot.z, 0, 0, 1 );
                   // setup texture units
                   // bind bump (normal) map
                   // to texture unit 0
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, bumpMap );
                   // bind normalization cube map
                   // to texture unit 1
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnable           ( GL_TEXTURE_CUBE_MAP_ARB );
    glBindTexture      ( GL_TEXTURE_CUBE_MAP_ARB, normCubeMap );
                   // setup register combiners
    glEnable                ( GL_REGISTER_COMBINERS_NV );
    glCombinerParameteriNV  ( GL_NUM_GENERAL_COMBINERS_NV, 1 );
    glCombinerParameterfvNV ( GL_CONSTANT_COLOR0_NV, lightColor );
    glCombinerParameterfvNV ( GL_CONSTANT_COLOR1_NV, ambientColor );
                   // configure A = expand (tex0)
    glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
                        GL_TEXTURE0_ARB, GL_EXPAND_NORMAL_NV,
                        GL_RGB );
```

```
                // configure B = expand (tex1)
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                    GL_RGB );
                // configure C = 0
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_C_NV,
                    GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                // configure D = 0
glCombinerInputNV ( GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_D_NV,
                    GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                // setup output of (l,n)
glCombinerOutputNV ( GL_COMBINER0_NV, GL_RGB,
                     GL_SPARE0_NV,         // AB output
                     GL_DISCARD_NV,        // CD output
                     GL_DISCARD_NV,        // sum output
                     GL_NONE,              // no scale
                     GL_NONE,              // no bias
                     GL_TRUE,              // AB = A dot B
                     GL_FALSE, GL_FALSE );
                // configure A.alpha = 1
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_A_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                // configure B.alpha = lz
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_B_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                    GL_BLUE );
                // configure C.alpha = 1
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_C_NV,
                    GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                // configure D.alpha = lz
glCombinerInputNV ( GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_D_NV,
                    GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV,
                    GL_BLUE );
glCombinerOutputNV ( GL_COMBINER0_NV, GL_ALPHA,
                     GL_DISCARD_NV,        // AB
                     GL_DISCARD_NV,        // CD
                     GL_SPARE0_NV,         // AB+CD
                     GL_SCALE_BY_FOUR_NV,
```

```
                        GL_NONE,              // bias
                        GL_FALSE, GL_FALSE, GL_FALSE );
                    // now spare0.rgb contains (l,n)
                    //     spare0.alpha contains 8*lz
                    // configure final combiner
                    // A.rgb = max ( spare0.rgb, 0 )
    glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_SPARE0_NV,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // B.rgb = EF
    glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_E_TIMES_F_NV,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // C = 0
    glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // D = constant_color1.rgb
    glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_CONSTANT_COLOR1_NV,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // E = constant_color_0
    glFinalCombinerInputNV ( GL_VARIABLE_E_NV, GL_CONSTANT_COLOR0_NV,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // F = selfShadow = 8*lz
    glFinalCombinerInputNV ( GL_VARIABLE_F_NV, GL_SPARE0_NV,
                             GL_UNSIGNED_IDENTITY_NV, GL_ALPHA );
                    // G.alpa = 1
    glFinalCombinerInputNV ( GL_VARIABLE_G_NV, GL_ZERO,
                             GL_UNSIGNED_INVERT_NV, GL_ALPHA );
                    // so at output we'll get
                    // RGB   = c1 + c0 * max ( (l, n), 0 )
                    // alpha = 1
                    // now draw torus
    glEnable         ( GL_VERTEX_PROGRAM_ARB );
    glBindProgramARB ( GL_VERTEX_PROGRAM_ARB, program );
    torus.draw ();
    glDisable   ( GL_VERTEX_PROGRAM_ARB );
    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, 1 );        // 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 );
    light.x = 2*cos ( angle );
    light.y = 2*sin ( angle );
    light.z = 3 + 0.3 * sin ( angle / 3 );
                        // setup data
    float    light4D [4];
    light4D [0] = light.x;
    light4D [1] = light.y;
    light4D [2] = light.z;
    light4D [3] = 1;
    glProgramLocalParameter4fvARB ( GL_VERTEX_PROGRAM_ARB, 1,
                                    light4D );
    glutPostRedisplay ();
}
void init ()
{
    glClearColor ( 0.0, 0.0, 0.0, 1.0 );
    glEnable     ( GL_DEPTH_TEST );
    glEnable     ( GL_VERTEX_PROGRAM_ARB );
    glDepthFunc  ( GL_LEQUAL );
    glHint ( GL_POLYGON_SMOOTH_HINT,         GL_NICEST );
    glHint ( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}
int main ( int argc, char * argv [] )
{
                        // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                        // create window
    int    win = glutCreateWindow ( "Torus with  per-pixel diffuse light-
ing via register combiners" );
```

```
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutMouseFunc    ( mouse   );
    glutMotionFunc   ( motion  );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init           ();
    initExtensions ();
    printfInfo     ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_NV_register_combiners" ) )
    {
        printf ( "NV_register_combiners NOT supported" );
        return 2;
    }
    if ( !isExtensionSupported ( "GL_ARB_vertex_program" ) )
    {
        printf ( "GL_ARB_vertex_program NOT supported" );
        return 3;
    }
    bumpMap     = createNormalMapFromHeightMap ( false,
                                    "../Textures/Bumpmaps/h.tga", 3 );
    decalMap    = createTexture2D              ( true,
                                    "../Textures/block.bmp" );
    normCubeMap = createNormalizationCubemap   ( 32 );
    program     = loadProgram                  ( "diffuse.vp" );
    if ( program == 0 )
    {
        printf ( "error loading diffuse.vp\n" );
        return 1;
    }
    glutMainLoop ();
    return 0;
}
```

В этом листинге функция `loadProgram` отвечает за загрузку вершинной программы из файла.

Вычисление необходимых параметров для попиксельного бликового освещения

Еще одним примером является вычисление необходимых векторов (l и h) для попиксельного бликового (*specular*) освещения.

Здесь выполняется несколько сложений: для каждой вершины необходимо найти векторы на источник света и на наблюдателя, нормировать и вычислить их полусумму (вектор h). Далее они оба переводятся в касательное пространство и записываются в выходные регистры. С этой задачей легко справляется вершинная программа, приводимая в листинге 15.9.

Листинг 15.9. Пример вершинной программы для попиксельного бликового освещения

```
!!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
#      vertex.texcoord [1] - tangent vector  (t)
#      vertex.texcoord [2] - binormal vector (b)
#
#      program.local [0] - eye position
#      program.local [1] - light position
#
# on exit:
#     result.texcoord [0] - texture coordinates
#     result.texcoord [1] - l
#     result.texcoord [2] - h
#
PARAM    eye     = program.local [0];
PARAM    light   = program.local [1];
PARAM    mvp [4] = { state.matrix.mvp };
PARAM    half    = 0.5;
```

```
TEMP    l, l2, v, v2, h, h2, temp;
TEMP    ht;
                # compute l (vector to light)
ADD     l, -vertex.position, light;
                # 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 (l,l)
MUL     l, l, temp.y;           # normalize
                # compute v (vector to viewer)
ADD     v, -vertex.position, eye;
                # 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)/2
ADD     h, l, v;
MUL     h, h, half;
                # transform it into tangent space
DP3     ht.x, h, vertex.texcoord [1];
DP3     ht.y, h, vertex.texcoord [2];
DP3     ht.z, h, vertex.normal;
MOV     ht.w, h.w;
                # store it into texcoord [1]
MOV     result.texcoord [1], ht;
                # store 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, vertex.position, mvp [0];
DP4     result.position.y, vertex.position, mvp [1];
DP4     result.position.z, vertex.position, mvp [2];
DP4     result.position.w, vertex.position, mvp [3];
                # we're done
END
```

Соответствующий код на C++ находится на прилагаемом к книге компакт-диске (файл vp-specular.cpp).

Заворачиваем вершинную программу в класс

Для удобства использования вершинных программ и доступа к их параметрам можно завернуть вершинную программу (вместе с методами загрузки, доступа к параметрам, запросам на ограничения) в класс, предоставляющий весь необходимый доступ, но при этом скрывающий от пользователя внутренние детали.

В частности, довольно удобно было бы реализовать обращение к локальным параметрам и параметрам окружения таким же образом, каким это происходит из самой вершинной программы, например, как показано в листинге 15.10.

Листинг 15.10. Желательный вариант обращения к локальным переменным

```
myProgram.local [1] = Vector3D ( x, y, z );

otherProgram.local [0] = v;

Vector4D    v2 = VertexProgram :: env [2];
```

Фактически хочется реализовать то, что в Delphi, VB и других языках называется свойствами (*property*). С точки зрения внешнего пользователя это обычная переменная, но доступ к ней (чтение и запись) происходит через специальные функции.

На самом деле на языке C++ это можно реализовать. В качестве соответствующей переменной должен выступать экземпляр класса, у которого переопределены операторы присваивания (т. е. запись в него значения) и преобразования в нужный тип (т. е. чтение значения из него). При этом сам класс может содержать внутри себя указатели на функции (или методы класса), обеспечивающие чтение и запись этого значения.

Листинг 15.11 содержит описание простого класса, обеспечивающего доступ к векторным свойствам через скрытые внутри функции доступа.

Листинг 15.11. Описание класса `ParamProxy`

```
class    ParamProxy
{
```

```
public:
      typedef void ( __stdcall *PutFunc)( GLenum target, GLuint index,
                                          const float * vector );
      typedef void ( __stdcall *GetFunc)( GLenum target, GLuint index,
                                          float * params );
protected:
    GetFunc    getter;
    PutFunc    putter;
    int        index;
    int        target;
    unsigned   id;
public:
    ParamProxy ( PutFunc put, GetFunc get, GLenum theTarget,
                 int theIndex, unsigned theId );
                              // put-method
    void    operator = ( const Vector4D& v );
                              // get method
    operator Vector4D () const;
};
```

При попытке присвоить экземпляру данного класса 4-мерный вектор происходит вызов переопределенного оператора присваивания, который делегирует этот запрос функции.

Однако при этом надо помнить, что соответствующие запросы относятся к текущей (т. е. выбранной командой `glBindProgramARB`) программе. Поэтому перед выполнением соответствующего запроса необходимо проверить идентификатор текущей программы и, при несовпадении, выбрать заданную (и восстановить предыдущую текущую программу). Соответствующий фрагмент кода приведен в листинге 15.12.

Листинг 15.12. Запись векторного значения

```
void ParamProxy :: operator = ( const Vector4D& v )
{
    unsigned    idCur;
                // we should ensure that this program
                // is currently bound
    glGetProgramivARB ( target, GL_PROGRAM_BINDING_ARB,
                        (int *)&idCur );
    if ( idCur != id )
        glBindProgramARB ( target, id );
```

```
    putter ( target, index, v );
    if ( idCur != id )
        glBindProgramARB ( target, idCur );
}
```

Аналогично чтение векторного значения приводит к вызову переопределенного оператора приведения типа, который также делегирует эту операцию заданной функции (листинг 15.13).

Листинг 15.13. Чтение векторного значения

```
ParamProxy :: operator Vector4D () const
{
    Vector4D    v1 ( v );
    unsigned    idCur;
    glGetProgramivARB ( target, GL_PROGRAM_BINDING_ARB,
                        (int *)&idCur );
    if ( idCur != id )
        glBindProgramARB ( target, id );
    putter ( target, index, v1 );
    if ( idCur != id )
        glBindProgramARB ( target, idCur );
}
```

Использование приведенного класса `ParamProxy` позволяет реализовать доступ к любому параметру вершинной программы, однако поскольку мы имеем дело с массивами этих параметров, нужно ввести еще один класс, у которого оператор `[]` переопределен таким образом, что он возвращает соответствующий экземпляр класса `ParamProxy`:

```
ParamProxy operator [] ( int index )
{
    return ParamProxy ( putter, getter, target, index, id );
}
```

В листинге 15.14 приводится описание класса, реализующего абстракцию вершинной программы и работы с ней.

Листинг 15.14. Описание класса `VertexProgram`

```
class    VertexProgram
{
```

```
protected:
    int                 errorCode;
    string              errorString;
    unsigned            id;                 // program id
public:
    ParamArray          local;              // local parameters
    static ParamArray   env;                // environment params
    static int          activeProgram;
    static int          maxVertexAttribs ();
    static int          maxLocalParams   ();
    static int          maxEnvParams     ();
    static int          maxMatrices      ();
    static int          maxTemporaries   ();
    static int          maxParams        ();
    static  bool        isSupported       ();
    VertexProgram  ();
    VertexProgram  ( const char * fileName );
   ~VertexProgram  ();
    unsigned    getId () const
    {
        return id;
    }
    string    getErrorString () const
    {
       return errorString;
    }
    void    bind    ();
    void    unbind  ();
    void    enable  ();
    void    disable ();
    bool    load ( Data * data );
    bool    load ( const char * fileName );
};
```

Реализацию данного класса иллюстрирует листинг 15.15.

Листинг 15.15. Реализация класса `VertexProgram`

```
//
// Simple wrapper class for OpenGL vertex programs
```

```
//
#include    "VertexProgram.h"
#include    "Data.h"
#include    <stdio.h>
ParamArray    VertexProgram :: env ( GL_VERTEX_PROGRAM_ARB );
int           VertexProgram :: activeProgram    = 0;
//////////////////// VertexProgram methods /////////////////////
VertexProgram :: VertexProgram  () : local ( GL_VERTEX_PROGRAM_ARB )
{
    id          = 0;
    errorCode   = 0;
    errorString = "";
}
VertexProgram :: VertexProgram  ( const char * fileName )  :
            local ( GL_VERTEX_PROGRAM_ARB )
{
    id = 0;
    load ( fileName );
}
VertexProgram :: ~VertexProgram ()
{
    glDeleteProgramsARB ( 1, &id );
}
bool    VertexProgram :: load ( const char * fileName )
{
    Data    data ( fileName );
    return load ( &data );
}
bool    VertexProgram :: load ( Data * data )
{
    if ( id == 0 )            // allocate an id for program
        glGenProgramsARB ( 1, &id );
                             // set correct pointers to functions,
                              // because at this time extension
                             // should be initialized
    local.setPointers ( glProgramLocalParameter4fvARB,
                        glGetProgramLocalParameterfvARB );
    local.setId       ( id );
```

```
    env.  setPointers ( glProgramEnvParameter4fvARB,
                        glGetProgramEnvParameterfvARB     );
    if ( !data -> isOk () || data -> isEmpty () )
        return false;
    bind ();
    glProgramStringARB ( GL_VERTEX_PROGRAM_ARB,
                         GL_PROGRAM_FORMAT_ASCII_ARB,
                         data -> getLength (), data -> getPtr () );
    if ( glGetError () == GL_INVALID_OPERATION )
    {
              glGetIntegerv ( GL_PROGRAM_ERROR_POSITION_ARB,
                              &errorCode );
              errorString = (const char *)
                            glGetString ( GL_PROGRAM_ERROR_STRING_ARB );
              return false;
    }
    errorCode   = 0;
    errorString = "";
    return true;
}
void    VertexProgram :: enable ()
{
    glEnable ( GL_VERTEX_PROGRAM_ARB );
}
void    VertexProgram :: disable ()
{
    glDisable ( GL_VERTEX_PROGRAM_ARB );
}
void    VertexProgram :: bind ()
{
     glBindProgramARB ( GL_VERTEX_PROGRAM_ARB, activeProgram = id );
}
void    VertexProgram :: unbind ()
{
     glBindProgramARB ( GL_VERTEX_PROGRAM_ARB, activeProgram = 0 );
}
bool    VertexProgram :: isSupported ()
{
    return isExtensionSupported ( "GL_ARB_vertex_program" );
}
```

```
int    VertexProgram :: maxVertexAttribs ()
{
    int        maxVertexAttribs;
    glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB, GL_MAX_VERTEX_ATTRIBS_ARB,
                        &maxVertexAttribs );
    return maxVertexAttribs;
}
int    VertexProgram :: maxLocalParams ()
{
    int        maxLocalParams;
    glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_PROGRAM_LOCAL_PARAMETERS_ARB,
                        &maxLocalParams   );
    return maxLocalParams;
}
int    VertexProgram :: maxEnvParams ()
{
    int        maxEnvParams;
    glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_PROGRAM_ENV_PARAMETERS_ARB,
                        &maxEnvParams     );
    return maxEnvParams;
}
int VertexProgram :: maxMatrices ()
{
    int         maxMatrices;
    glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_PROGRAM_MATRICES_ARB, &maxMatrices );
    return maxMatrices;
}
int    VertexProgram :: maxTemporaries ()
{
    int     maxTemporaries;
    glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_PROGRAM_TEMPORARIES_ARB,
                        &maxTemporaries   );
    return maxTemporaries;
}
```

```
int    VertexProgram :: maxParams ()
{
    int        maxParams;
    glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_PROGRAM_PARAMETERS_ARB, &maxParams );
    return maxParams;
}
int    VertexProgram :: maxAddressRegs ()
{
    int        maxAddressRegs;
    glGetProgramivARB ( GL_VERTEX_PROGRAM_ARB,
                        GL_MAX_PROGRAM_ADDRESS_REGISTERS_ARB,
                        &maxAddressRegs  );
    return maxAddressRegs;
}
```

Как видно, данный класс позволяет получать информацию об ограничениях, накладываемых на вершинные программы, обеспечивать доступ к локальным параметрам и параметрам окружения, осуществлять выбор программы как текущей, а также разрешать или запрещать ее использование.

Реализация EMBM при помощи вершинной программы

Рассмотрим, каким образом можно настроить параметры для EMBM при помощи вершинной программы. Для работы с вершинной программой будет использоваться класс `VertexProgram`, описанный выше.

Сама реализация вершинной программы достаточно проста и аналогична рассмотренным ранее (листинг 15.16).

Листинг 15.16. Вершинная программа для настройки параметров для EMBM

```
!!ARBvp1.0
#
# simple vertex shader to setup data for EMBM (true reflection mapping)
#
# on entry:
#      vertex.position
#      vertex.normal       - normal vector (n) of TBN basic
```

```
#      vertex.texcoord [0] - normal texture coordinates
#      vertex.texcoord [1] - tangent vector  (t)
#      vertex.texcoord [2] - binormal vector (b)
#
#      program.local [0] - eye position
#      program.local [1] - light position
#
# on exit:
#     result.texcoord [0] - texture coordinates
#     other texcoords set for GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV
#     texture shader
#
PARAM    eye      = program.local [0];
PARAM    light    = program.local [1];
PARAM    mvp  [4] = { state.matrix.mvp };
PARAM    mvit [4] = { state.matrix.modelview.invtrans };
PARAM    mv   [4] = { state.matrix.modelview };
TEMP    et;
           # compute and transform eye vector
DP4    et.x, -vertex.position, mv [0];
DP4    et.y, -vertex.position, mv [1];
DP4    et.z, -vertex.position, mv [2];
DP4    et.w, -vertex.position, mv [3];
           # compute 3*3 matrix and put it into texcoord [1],
           # texcoord [2], texcoord [3]
DP3    result.texcoord [1].x, mvit [0], vertex.texcoord [1];
DP3    result.texcoord [1].y, mvit [0], vertex.texcoord [2];
DP3    result.texcoord [1].z, mvit [0], vertex.normal;
DP3    result.texcoord [2].x, mvit [1], vertex.texcoord [1];
DP3    result.texcoord [2].y, mvit [1], vertex.texcoord [2];
DP3    result.texcoord [2].z, mvit [1], vertex.normal;
DP3    result.texcoord [3].x, mvit [2], vertex.texcoord [1];
DP3    result.texcoord [3].y, mvit [2], vertex.texcoord [2];
DP3    result.texcoord [3].z, mvit [2], vertex.normal;
           # store transformed eye vector into w coords
           # of tex1, tex2, tex3
MOV    result.texcoord [1].w, et.x;
MOV    result.texcoord [2].w, et.y;
```

```
MOV    result.texcoord [3].w, et.z;
            # store 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, vertex.position, mvp [0];
DP4    result.position.y, vertex.position, mvp [1];
DP4    result.position.z, vertex.position, mvp [2];
DP4    result.position.w, vertex.position, mvp [3];
            # we're done
END
```

Программа на C++ с классом `VertexProgram` приводится в листинге 15.17.

Листинг 15.17. Программа на C++, реализующая EMBM

```
//
// Sample to to per-pixel true reflective bumpmapping in OpenGL
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Torus.h"
#include    "Data.h"
#include    "VertexProgram.h"
Vector3D    eye   ( 8, 0, 0 );        // camera position
unsigned    envMap;                   // environment cubemap id
unsigned    bumpMap;                  // normal map
float       angle = 0;
Torus       torus ( 0.8, 2.5, 30, 30 );
Vector3D    rot ( 0, 0, 0 );
int         mouseOldX = 0;
int         mouseOldY = 0;
```

```
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",
};
VertexProgram    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 );
    glEnable ( GL_TEXTURE_SHADER_NV );
    glEnable ( GL_REGISTER_COMBINERS_NV );
                    // setup texture stages
                    // stage 0:
                    // bind bump (normal) map to
                    // texture unit 0
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, bumpMap );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                         GL_TEXTURE_2D );
                    // stage 1:
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                         GL_DOT_PRODUCT_NV );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                         GL_PREVIOUS_TEXTURE_INPUT_NV,
                         GL_TEXTURE0_ARB );
```

```
                    // stage 2:
    glActiveTextureARB ( GL_TEXTURE2_ARB );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                         GL_DOT_PRODUCT_NV );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                         GL_PREVIOUS_TEXTURE_INPUT_NV,
                         GL_TEXTURE0_ARB );
                    // stage 3:
// bind environment cube
                    // map to texture unit 3
    glActiveTextureARB ( GL_TEXTURE3_ARB );
    glEnable           ( GL_TEXTURE_CUBE_MAP_ARB );
    glBindTexture      ( GL_TEXTURE_CUBE_MAP_ARB, envMap );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
                         GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV );
    glTexEnvi          ( GL_TEXTURE_SHADER_NV,
                         GL_PREVIOUS_TEXTURE_INPUT_NV,
                         GL_TEXTURE0_ARB );
                    // setup register combiners
    glFinalCombinerInputNV ( GL_VARIABLE_A_NV, GL_TEXTURE3_ARB,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // B.rgb = 1
    glFinalCombinerInputNV ( GL_VARIABLE_B_NV, GL_ZERO,
                             GL_UNSIGNED_INVERT_NV, GL_RGB );
                    // C = 0
    glFinalCombinerInputNV ( GL_VARIABLE_C_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
                    // D = 0
    glFinalCombinerInputNV ( GL_VARIABLE_D_NV, GL_ZERO,
                             GL_UNSIGNED_IDENTITY_NV, GL_RGB );
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();
    glRotatef    ( rot.x, 1, 0, 0 );
    glRotatef    ( rot.y, 0, 1, 0 );
    glRotatef    ( rot.z, 0, 0, 1 );
    program.enable ();
    program.bind   ();
  program.local [0] = eye;
```

```
    torus.draw ();
    program.disable ();
    glPopMatrix     ();
    glutSwapBuffers ();
}
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 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, 1 );              // up
}
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  ( 500, 500 );
                        // create window
    int    win = glutCreateWindow ( "OpenGL true per-pixel reflective
bumpmapping via vertex program" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutMouseFunc    ( mouse   );
    glutMotionFunc   ( motion  );
    init ();
    printfInfo ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_NV_register_combiners" ) )
    {
        printf ( "NV_register_combiners NOT supported" );
        return 2;
    }
    if ( !isExtensionSupported ( "GL_NV_texture_shader" ) )
    {
        printf ( "GL_NV_texture_shader NOT supported" );
```

```
        return 3;
    }
    if ( !isExtensionSupported ( "GL_ARB_vertex_program" ) )
    {
        printf ( "GL_ARB_vertex_program NOT supported" );
        return 3;
    }
    initExtensions ();
    envMap  = createCubeMap                ( true, faces );
    bumpMap = createNormalMapFromHeightMap ( false,
                                      "../Textures/Bumpmaps/h.tga", 3 );
    if ( !program.load ( "embm.vp" ) )
    {
        printf ( "Error loading program: %s\n",
                        program.getErrorString ().c_str () );
        return 1;
    }
    glutMainLoop ();
    return 0;
}
```

Как видно из приведенного листинга, использование класса `VertexProgram` действительно упростило работу с вершинными программами.

Применение вершинной программы для анимации объектов

Еще одним применением вершинных программ является анимация объектов. Мы рассмотрим это на примере простой системы частиц. Вершинная программа будет отвечать за перемещение частиц и изменение их цвета в зависимости от времени.

Пусть закон изменения положения и цвета частицы является периодическим. Так как в системе команд для вершинной программы нет тригонометрических функций, то сама исходная программа на C++ будет снабжать вершинную программу значением $(1 + \sin(t))/2$. Для каждой частицы будет задан вектор, вдоль которого осуществляется анимация. В листинге 15.18 приводится соответствующая вершинная программа.

Листинг 15.18. Вершинная программа для анимации системы частиц

```
!!ARBvp1.0
#
# simple vertex shader particle system animation
#
# on entry:
#      vertex.position
#      vertex.texcoord [0] - r vector
#      vertex.texcoord [1] - amp
#
#      program.local [0] - sin ( current time )
#
ATTRIB   pos     = vertex.position;
ATTRIB   r       = vertex.texcoord [0];
ATTRIB   freq    = vertex.texcoord [1];
PARAM    t       = program.local [0];
PARAM    mvp [4] = { state.matrix.mvp };
PARAM    piVec   = { 0.159154943, 6.283185307, 0, 0 };
PARAM    c1      = { 1, 1, 0, 1 };        # yellow
PARAM    c2      = { 0, 1, 1, 1 };        # cyan
TEMP     v, temp, cd;
                  # move vertex along radii
MUL    temp, amp.x, t.x;
ADD    temp, temp, 1;
MAD    v, r, temp.x, pos;        # newPos = pos + r * ( 1 + freq*t )
                  # transform position into clip space
DP4    result.position.x, v, mvp [0];
DP4    result.position.y, v, mvp [1];
DP4    result.position.z, v, mvp [2];
DP4    result.position.w, v, mvp [3];
                  # compute colors
ADD    cd, c2, -c1;                  # cd = c2 - c1
MAD    result.color, cd, t.x, c1;    # color = (c2-c1) * t + c1
MOV    result.color.secondary, vertex.color.secondary;
                  # we're done
END
```

Данная вершинная программа на входе для каждой вершины (частицы) получает ее положение (`vertex.position`), направление ее движения (`vertex.texcoord [0]`) и амплитуду колебаний (`vertex.texcoord [1].x`).

Текущее время передается программе через `program.local [0]`.

Все, что остается делать программе на C++, — это, один раз определив данные для каждой частицы, постоянно передавать набор этих данных (фактически просто выводить точки), а также передавать вершинной программе параметр анимации через ее локальную переменную. Вся анимация системы происходит полностью на GPU.

Соответствующая программа на C++ приводится в листинге 15.19.

Листинг 15.19. Исходная программа на C++ для анимации системы частиц

```
//
// Sample to particle system simulation via vertex programs in OpenGL
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "VertexProgram.h"
#define    NUM_PARTICLES    700
struct    ParticleData
{
    Vector3D    pos;
    Vector3D    r;
    float        freq;
};
Vector3D    eye   ( 7, 5, 7 );        // camera position
unsigned    decalMap;                 // decal (diffuse) texture
float       angle = 0;
Vector3D    rot ( 0, 0, 0 );
int         mouseOldX = 0;
int         mouseOldY = 0;
ParticleData     particles [NUM_PARTICLES];
VertexProgram    program;
```

```
void    initParticles ( const Vector3D& center )
{
    Vector3D    e1 ( 1,  0, 0 );
    Vector3D    e2 ( 0, -1, 0 );
    for ( int i = 0; i < NUM_PARTICLES; i++ )
    {
        Vector3D    r = Vector3D:: getRandomVector ( 2 );
        particles [i].pos  = center;
        particles [i].r    = r;
        particles [i].freq = 1 + 8 * pow ( 0.5 +
                                           0.5 * fabs ( (e1 & r) ), 4 );
    }
}
void init ()
{
    initParticles ( Vector3D ( 0, 0, 0 ) );
    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 );
    glEnable      ( GL_TEXTURE_2D );
    glDepthMask   ( GL_FALSE );
    glBindTexture ( GL_TEXTURE_2D, decalMap );
    glEnable      ( GL_BLEND );
    glBlendFunc   ( GL_SRC_ALPHA, GL_ONE );
    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 );
    glTexEnvf             ( GL_POINT_SPRITE_ARB, GL_COORD_REPLACE_ARB,
                            GL_TRUE );
```

```
    glPointSize ( 8 );
    program.enable ();
    program.bind   ();
    glPushMatrix   ();
    glRotatef    ( rot.x, 1, 0, 0 );
    glRotatef    ( rot.y, 0, 1, 0 );
    glRotatef    ( rot.z, 0, 0, 1 );
    glBegin ( GL_POINTS );
    for ( int i = 0; i < NUM_PARTICLES; i++ )
    {
        glMultiTexCoord3fv ( GL_TEXTURE0_ARB, particles [i].r    );
        glMultiTexCoord1f  ( GL_TEXTURE1_ARB, particles [i].freq );
        glVertex3fv        ( particles [i].pos                   );
    }
    glEnd ();
    program.disable ();
    glPopMatrix ();
    glPopAttrib ();
    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, 1 );              // 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 ()
{
    float    t  = 0.003f * glutGet ( GLUT_ELAPSED_TIME );
    float    st = 0.5f * ( 1 + sin ( t ) );
    program.local [0] = Vector4D ( st, st, st, st );
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                            // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                            // create window
    int    win = glutCreateWindow ( "OpenGL particle systems animation
via vertex program" );
```

```
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutMouseFunc    ( mouse   );
    glutMotionFunc   ( motion  );
    glutKeyboardFunc ( key     );
    glutIdleFunc     ( animate );
    init           ();
    initExtensions ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
    if ( !isExtensionSupported ( "GL_ARB_point_parameters" ) )
    {
        printf ( "ARB_point_parameters NOT supported.\n" );
        return 2;
    }
    if ( !isExtensionSupported ( "GL_ARB_point_sprite" ) )
    {
        printf ( "ARB_point_sprite NOT supported" );
        return 3;
    }
    if ( !isExtensionSupported ( "GL_ARB_vertex_program" ) )
    {
        printf ( "GL_ARB_vertex_program NOT supported" );
        return 4;
    }
    decalMap = createTexture2D ( true,  "../Textures/particle1.bmp" );

    if ( !program.load ( "particles.vp" ) )
    {
        printf ( "Error loading program: %s\n",
                        program.getErrorString ().c_str () );
        return 1;
    }
    glutMainLoop ();
    return 0;
}
```

Изображение, построенное данной программой, представлено на рис. 15.8.

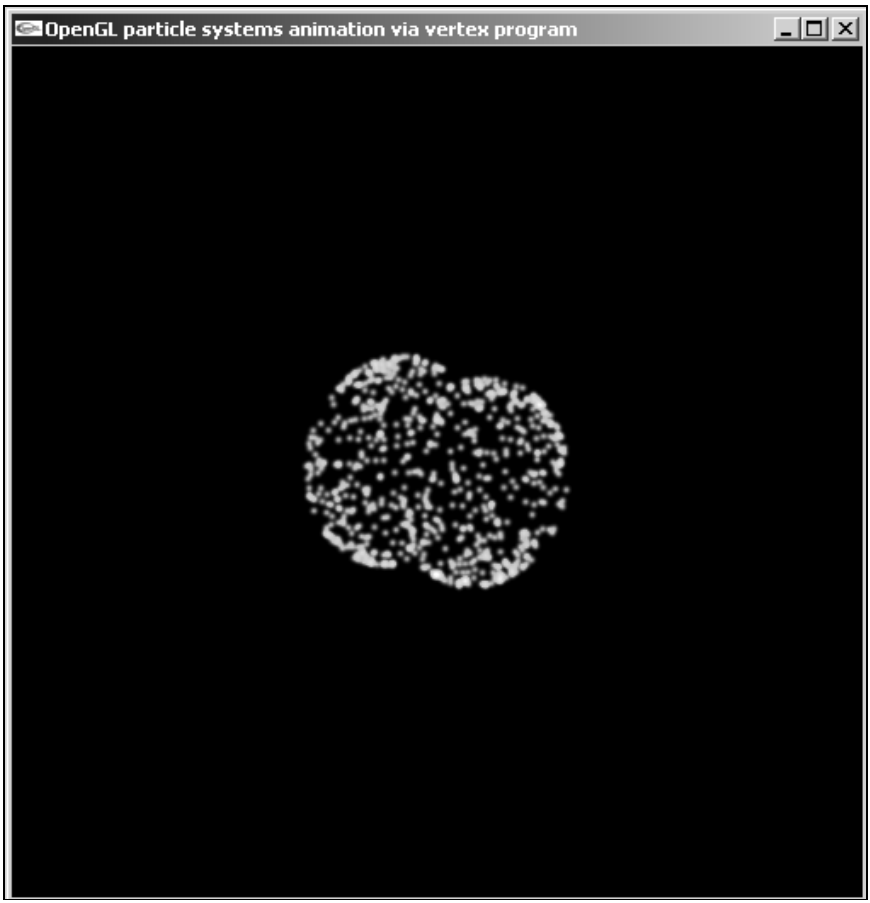
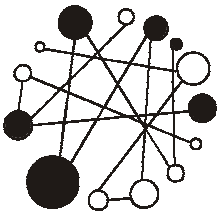


Рис. 15.8. Изображение системы частиц

Очевидно, что вершинные программы представляют собой очень мощный механизм, позволяющий переложить на графический ускоритель большой объем вспомогательных вычислений, необходимых для обработки каждой вершины.

Глава 16



Фрагментные программы и работа с ними через расширение ARB_fragment_program

Рассмотренные в предыдущей главе вершинные программы позволяют программисту явно задавать свои законы преобразования данных в вершинах в виде набора команд, работающих с 4-мерными векторами.

На вход вершинной программы поступают как данные в самих вершинах, так и различные параметры (локальные и окружения). По ним программа вычисляет выходные значения (положение вершины, цвет, наборы текстурных координат и т. п.).

Такой метод позволяет переложить на GPU большую часть преобразований данных в вершинах. Но для получения действительно красивых и сложных эффектов одних операций в вершинах уже недостаточно (на самом деле есть примеры, реализующие довольно сложные шейдеры даже на стандартном OpenGL без всяких расширений, однако это требует очень большого числа проходов, и точность таких вычислений невысока).

Желательно иметь возможность явно задавать законы обработки данных на уровне отдельных фрагментов (пикселов). Именно такая функциональность и предоставляется рассматриваемым ниже расширением ARB_fragment_program.

За его основу взято расширение ARB_vertex_program, поэтому очень многое в данном расширении уже будет вам знакомо (включая создание, загрузку и уничтожение программ, задание параметров и т. п.).

Листинг 16.1 содержит пример кода, который загружает фрагментную программу.

Листинг 16.1. Пример загрузки фрагментной программы

```
const char * vpText =
"!!ARBfp1.0\
 ATTRIB tex0   = fragment.texcoord [0];\
```

```
 ATTRIB color0 = fragment.color;\
 OUTPUT color  = result.color;\
 TEMP temp;\
 TXP temp, tex0, texture [0], 2D;\
 MUL color, temp, color0;\
 END";
GLuint       progId;
glGenProgramsARB   ( 1, &progId );
glBindProgramARB   ( GL_FRAGMENT_PROGRAM_ARB, progId );
glProgramStringARB ( GL_FRAGMENT_PROGRAM_ARB,
                     GL_PROGRAM_FORMAT_ASCII_ARB,
                     strlen ( vpText ), vpText );
if ( glGetLastError () == GL_INVALID_OPERATION )
{
    GLint     errorPos;
    GLubyte * errorStr;
    glGetIntegerv ( GL_PROGRAM_ERROR_POSITION_ARB, &errorPos );
    errorString = glGetString ( GL_PROGRAM_ERROR_STRING_ARB );
    fprintf ( stderr, "Error at position %d.\Line: \"%s\"", errorPos,
              errorStr );
}
```

Сразу бросаются в глаза два отличия: константа `GL_FRAGMENT_PROGRAM_ARB` вместо `GL_VERTEX_PROGRAM_ARB` и первая строка `!!ARBfp1.0`.

Перед детальным изучением фрагментных программ сначала рассмотрим их место в графическом конвейере OpenGL.

Стандартный конвейер (*pipeline*) OpenGL представлен на рис. 16.1.

Фрагментная программа заменяет собой следующие шаги:

- обращение к текстуре;
- ее применение;
- сложение цветов;
- вычисление тумана.

Фрагментная программа выполняется независимо для каждого фрагмента изображения (для точек, линий и растровых изображений). Для заданного текстурного блока она игнорирует виды (одно-, дву-, трехмерная текстуры и кубические текстурные карты) и приоритеты текстур (кубической карты над 3D, 3D над 2D, 2D над 1D), а также то, какие текстурные цели (кубическая

карта, 1-2-3-мерная) разрешены для этого блока. На рис. 16.2 изображена фрагментная программа и различные множества регистров, с которыми она работает.

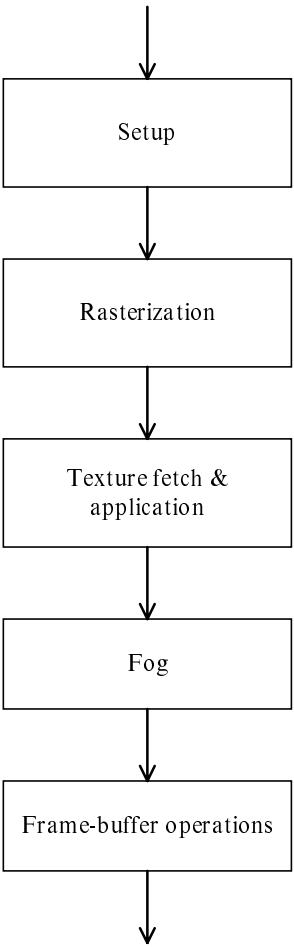


Рис. 16.1. Стандартный конвейер в OpenGL

Как и вершинная, фрагментная программа получает в свое распоряжение несколько наборов регистров (4-мерных вещественных векторов). Это атрибуты самого фрагмента (`fragment.*`), локальные параметры (`program.local[n]`) и параметры окружения (`program.env[n]`). Выходные значения программа записывает в регистры результата (`result.*`).

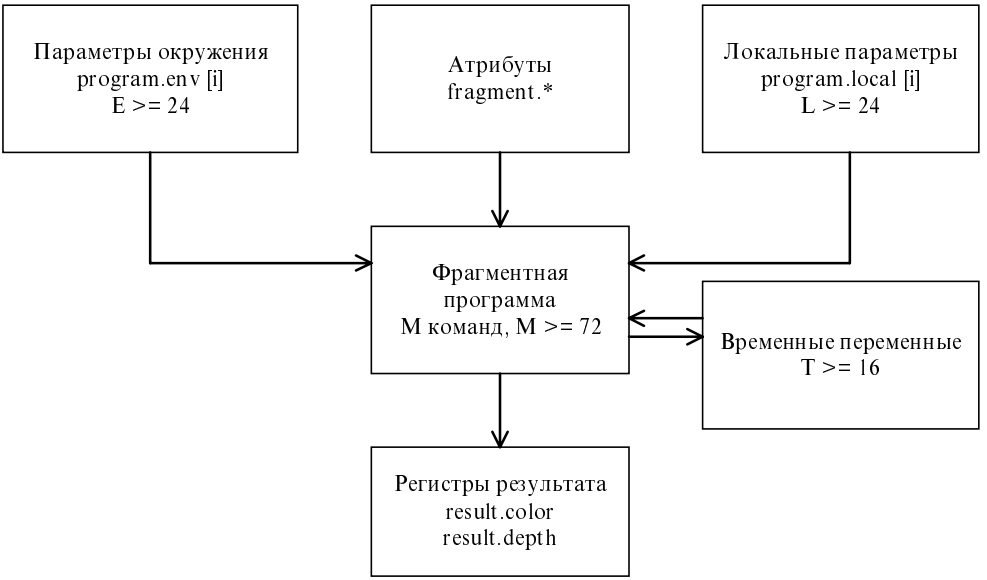


Рис. 16.2. Фрагментная программа и используемые регистры

В распоряжении фрагментной программы также находится набор временных переменных.

Операции создания, загрузки, выбора и уничтожения фрагментной программы полностью аналогичны соответствующим операциям для вершинных программ за исключением того, что константа `GL_VERTEX_PROGRAM_ARB` заменяется на `GL_FRAGMENT_PROGRAM_ARB`.

Для переноса фрагментной программы в другой контекст рендеринга, по аналогии с текстурами и вершинными программами, служит команда `wglShareLists`.

Работа с параметрами (локальными и окружения) также полностью аналогична работе с ними для вершинных программ.

Установленные графический ускоритель и драйвер накладывают определенные ограничения на фрагментные программы. В листинге 16.2 приводится программа, печатающая действующие на конкретной машине ограничения.

Листинг 16.2. Определение ограничений на фрагментную программу

```
//
// Sample to to check for fragment 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
    int    win = glutCreateWindow ( "OpenGL example 1" );
                        // 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_fragment_program" ) )
  {
      printf ( "GL_ARB_fragment_program extension NOT supported.\n" );

      return 1;
  }
  printf ( "ARB_fragment_program extension is supported !\n" );
  int    maxAluInstructions;
  int    maxTexInstructions;
  int    maxTexIndirections;
  int    maxLocalParams;
  int    maxEnvParams;
  int    maxTemporaries;
  int    maxAttribs;
int    maxInstructions;
  glGetProgramivARB ( GL_FRAGMENT_PROGRAM_ARB,
                        GL_MAX_PROGRAM_ALU_INSTRUCTIONS_ARB,
                       &maxAluInstructions );
  glGetProgramivARB ( GL_FRAGMENT_PROGRAM_ARB,
                        GL_MAX_PROGRAM_TEX_INSTRUCTIONS_ARB,
                        &maxTexInstructions );
  glGetProgramivARB ( GL_FRAGMENT_PROGRAM_ARB,
                        GL_MAX_PROGRAM_TEX_INDIRECTIONS_ARB,
                        &maxTexIndirections );
  glGetProgramivARB ( GL_FRAGMENT_PROGRAM_ARB,
                      GL_MAX_PROGRAM_LOCAL_PARAMETERS_ARB,
                      &maxLocalParams     );
  glGetProgramivARB ( GL_FRAGMENT_PROGRAM_ARB,
                      GL_MAX_PROGRAM_ENV_PARAMETERS_ARB,
                      &maxEnvParams       );
```

```
    glGetProgramivARB ( GL_FRAGMENT_PROGRAM_ARB,
                        GL_MAX_PROGRAM_TEMPORARIES_ARB,
                        &maxTemporaries     );
    glGetProgramivARB ( GL_FRAGMENT_PROGRAM_ARB,
                        GL_MAX_PROGRAM_ATTRIBS_ARB,
                        &maxAttribs          );
    glGetProgramivARB ( GL_FRAGMENT_PROGRAM_ARB,
                        GL_MAX_PROGRAM_INSTRUCTIONS_ARB,
                        &maxInstructions );
    printf ( "\nFragment program limits:\n"                        );
    printf ( "\tmax ALU instructions    : %d\n", maxAluInstructions );
    printf ( "\tmax tex instructions    : %d\n", maxTexInstructions );
    printf ( "\tmax tex indirections    : %d\n", maxTexIndirections );
    printf ( "\tmax local parameters    : %d\n", maxLocalParams     );
    printf ( "\tmax env. parameters     : %d\n", maxEnvParams       );
    printf ( "\tmax program temporaries : %d\n", maxTemporaries     );
    printf ( "\tmax attributes          : %d\n", maxAttribs         );
    printf ( "\tmax program instructions: %d\n", maxInstructions    );
    return 0;
}
```

Ограничения конкретного графического ускорителя и драйвера накладываются на максимальное число:

- арифметических операций (`maxAluInstructions`);
- текстурных операций (`maxTexInstructions`);
- всех операций (`maxInstructions`);
- текстурных зависимостей *texture indirections* (`maxTexIndirections`);
- локальных параметров (`maxLocalParams`);
- параметров окружения (`maxEnvParams`);
- временных переменных (`maxTemporaries`);
- используемых вершинных атрибутов (`maxAttribs`).

Ограничение на число текстурных зависимостей связано с тем, что не все реализации могут поддерживать фрагментные программы с большим количеством сложных текстурных зависимостей, например, когда результаты обращения к одной текстуре используются для получения текстурных координат для обращения к следующей текстуре и т. д.

При этом общее число команд может и не превышать допустимых пределов, но программа все равно не будет выполняться.

Вообще под текстурной зависимостью подразумевают вычисление текстурных координат при обращении к текстуре (командами `TEX`, `TXP` и `TXB`) на основе результата предыдущей команды (как текстурной, так и арифметической).

Для конкретной программы можно построить цепочку таких текстурных зависимостей, состоящую из отдельных узлов. Каждый узел содержит набор текстурных команд, которые могут выполняться параллельно и за которыми следуют арифметические команды.

Команда обращения к текстуре является зависимой, если она использует временную переменную в качестве текстурных координат (вместо атрибута или параметра).

Программа без текстурных зависимостей будет представляться цепочкой, состоящей всего из одного узла (т. е. количество текстурных зависимостей будет в ней равно единице).

Новый узел в цепочке текстурных зависимостей создается в одном из следующих двух случаев:

1. Текстурные координаты для текстурной команды берутся из временной переменной, в которую уже была запись в текущем узле (как текстурной, так и арифметической командой).
2. Результат текстурной команды записывается во временную переменную, в которую уже была запись или чтение арифметической командой в текущем узле.

Обратите внимание, что маски записи и перестановки компонент не играют роли при подсчете количества узлов в цепочке текстурных зависимостей.

Листинг 16.3 содержит примеры программ с одной, двумя и тремя текстурными зависимостями.

Листинг 16.3. Примеры фрагментных программ с одной, двумя и тремя текстурными зависимостями

```
!!ARBfp1.0
              # No texture instructions, but always 1 indirection
MOV result.color, fragment.color;
END

!!ARBfp1.0
              # A simple dependent texture instruction, 2 indirections
TEMP myColor;
```

```
MUL myColor, fragment.texcoord[0], fragment.texcoord[1];
TEX result.color, myColor, texture[0], 2D;
END

!!ARBfp1.0
              # A more complex example with 3 indirections
TEMP myColor1, myColor2;
TEX myColor1, fragment.texcoord[0], texture[0], 2D;
MUL myColor1, myColor1, myColor1;
TEX myColor2, fragment.texcoord[1], texture[1], 2D;
              # so far we still only have 1 indirection
              # This is #2
TEX myColor2, myColor1, texture[2], 2D;

              # This is #3
TEX result.color, myColor2, texture[3], 2D;
END
```

В первом примере вообще нет обращений к текстурам, поэтому цепочка зависимостей состоит всего из одного звена, т. е. имеет место всего одна зависимость.

Во втором в качестве текстурных координат служит временная переменная, что создает новое звено в цепочке зависимостей, т. е. получаются два звена и две текстурных зависимости.

В третьем случае первые два обращения к текстурам вообще не создают ни одного нового звена. После этого следуют два обращения, каждое из которых использует временную переменную для адресации текстуры. Тем самым создаются два звена, и общее число текстурных зависимостей становится равным трем.

Не забывайте, что сам факт успешной загрузки фрагментной программы еще не гарантирует ее выполнение непосредственно на GPU, а не эмуляцию. Проверить это можно при помощи следующего фрагмента кода:

```
GLboolean ProgramStringIsNative ( GLenum target, GLenum format,
                                  GLsizei len, const GLvoid *string )
{
    GLint errorPos, isNative;
    glProgramStringARB ( target, format, len, string );
    glGetIntegerv      ( GL_PROGRAM_ERROR_POSITION_ARB, &errorPos );
    glGetProgramivARB  ( GL_FRAGMENT_PROGRAM_ARB,
```

```
                                GL_PROGRAM_UNDER_NATIVE_LIMITS_ARB, &isNative );
    if ( (errorPos == -1) && (isNative == 1) )
        return GL_TRUE;
    return GL_FALSE;
}
```

Таблица 16.1 содержит список фрагментных атрибутов (т. е. индивидуальных атрибутов для каждого фрагмента).

Таблица 16.1. *Фрагментные атрибуты*

Название	Компоненты	Значение
`fragment.position`	(*x*, *y*, *z*, 1/*w*)	Координаты
`fragment.color`	(*r*, *g*, *b*, *a*)	Первичный цвет
`fragment.color.primary`	(*r*, *g*, *b*, *a*)	То же
`fragment.color.secondary`	(*r*, *g*, *b*, *a*)	Вторичный цвет
`fragment.texcoord`	(*s*, *t*, *r*, *q*)	Текстурные координаты для блока 0
`fragment.texcoord [n]`	(*s*, *t*, *r*, *q*)	То же для блока *n*
`fragment.fogcoord`	(*f*, 0, 0, 1)	Координата для тумана (расстояние)

Из фрагментной программы доступны также основные параметры состояния OpenGL. В табл. 16.2—16.6 приводятся основные параметры и имена для доступа к ним. Смысл этих параметров точно такой же, как и в вершинных программах.

Таблица 16.2. *Свойства материала*

Имя	Компоненты	Комментарий
`state.material.ambient`	(*r*, *g*, *b*, *a*)	Фоновый цвет материала для лицевой грани
`state.material.diffuse`	(*r*, *g*, *b*, *a*)	Диффузный цвет материала для лицевой грани
`state.material.specular`	(*r*, *g*, *b*, *a*)	Бликовый цвет материала для лицевой грани
`state.material.emission`	(*r*, *g*, *b*, *a*)	Цвет свечения материала для лицевой грани
`state.material.shininess`	(*s*, 0, 0, 1)	Степень Фонга материала для лицевой грани

Таблица 16.2 (окончание)

Имя	Компоненты	Комментарий
`state.material.front.ambient`	(*r*, *g*, *b*, *a*)	Фоновый цвет материала для лицевой грани
`state.material.front.diffuse`	(*r*, *g*, *b*, *a*)	Диффузный цвет материала для лицевой грани
`state.material.front.specular`	(*r*, *g*, *b*, *a*)	Бликовый цвет материала для лицевой грани
`state.material.front.emission`	(*r*, *g*, *b*, *a*)	Цвет свечения материала для лицевой грани
`state.material.front.shininess`	(*s*, 0, 0, 1)	Степень Фонга материала для лицевой грани
`state.material.back.ambient`	(*r*, *g*, *b*, *a*)	Фоновый цвет материала для нелицевой грани
`state.material.back.diffuse`	(*r*, *g*, *b*, *a*)	Диффузный цвет материала для нелицевой грани
`state.material.back.specular`	(*r*, *g*, *b*, *a*)	Бликовый цвет материала для нелицевой грани
`state.material.back.emission`	(*r*, *g*, *b*, *a*)	Цвет свечения материала для нелицевой грани
`state.material.back.shininess`	(*s*, 0, 0, 1)	Степень Фонга материала для нелицевой грани

Таблица 16.3. *Параметры источников света*

Имя	Компоненты	Комментарий
`state.light[n].ambient`	(*r*, *g*, *b*, *a*)	Фоновый цвет источника света *n*
`state.light[n].diffuse`	(*r*, *g*, *b*, *a*)	То же диффузный
`state.light[n].specular`	(*r*, *g*, *b*, *a*)	То же бликовый
`state.light[n].position`	(*x*, *y*, *z*, *w*)	Координаты источника света *n*
`state.light[n].attenuation`	(*a*, *b*, *c*, *e*)	Коэффициенты затухания в зависимости от расстояния и показатель для конического источника для источника света *n*

Таблица 16.3 *(окончание)*

Имя	Компоненты	Комментарий
`state.light[n].spot.direction`	(*x*, *y*, *z*, *c*)	Направление для конического источника света и косинус его угла (для источника света *n*)
`state.light[n].half`	(*x*, *y*, *z*, 1)	Нормализованный серединный угол между направлением от глаза на источник света и вектором (0, 0, 1)
`state.lightmodel.ambient`	(*r*, *g*, *b*, *a*)	Фоновый цвет модели освещения
`state.lightmodel.scenecolor`	(*r*, *g*, *b*, *a*)	Цвет сцены в модели освещения для лицевых граней
`state.lightmodel.front.scenecolor`	(*r*, *g*, *b*, *a*)	То же
`state.lightmodel.back.scenecolor`	(*r*, *g*, *b*, *a*)	Цвет сцены в модели освещения для нелицевых граней
`state.lightprod[n].ambient`	(*r*, *g*, *b*, *a*)	Произведение фонового цвета источника света и материала
`state.lightprod[n].diffuse`	(*r*, *g*, *b*, *a*)	То же диффузного
`state.lightprod[n].specular`	(*r*, *g*, *b*, *a*)	То же бликового
`state.lightprod[n].front.ambient`	(*r*, *g*, *b*, *a*)	Произведение фонового цвета источника света и материала для лицевой грани
`state.lightprod[n].front.diffuse`	(*r*, *g*, *b*, *a*)	То же диффузного
`state.lightprod[n].front.specular`	(*r*, *g*, *b*, *a*)	То же бликового
`state.lightprod[n].back.ambient`	(*r*, *g*, *b*, *a*)	Произведение фонового цвета источника света и материала для нелицевой грани
`state.lightprod[n].back.diffuse`	(*r*, *g*, *b*, *a*)	То же диффузного
`state.lightprod[n].back.specular`	(*r*, *g*, *b*, *a*)	То же бликового

Таблица 16.4. *Параметры генерации текстурных координат*

Имя	Компоненты	Комментарий
`state.texenv[n].color`	(*r*, *g*, *b*, *a*)	Цвет окружения для текстурного блока *n*

Таблица 16.5. *Параметры вычисления затуманивания*

Имя	Компоненты	Комментарий
`state.fog.color`	(*r*, *g*, *b*, *a*)	RGB-цвет тумана
`state.fog.params`	(*d*, *s*, *e*, *r*)	Плотность тумана (`d`), линейное начало и конец (`s`,`e`) и `1/(end-start)`

Таблица 16.6. *Параметры отсечения по глубине*

Имя	Компоненты	Комментарий
`state.depth.range`	(*n*, *f*, *d*, 1)	Диапазон изменения глубины `near`, `far` и `far-near`

Программе доступен также ряд матриц, перечисленных в табл. 16.7.

Таблица 16.7. *Матрицы*

Имя	Комментарий
`state.matrix.modelview[n]`	*n*-я матрица модельного преобразования. Параметр *n* необязателен
`state.matrix.projection`	Матрица проектирования
`state.matrix.mvp`	Произведение матрицы модельного преобразования и проектирования: `MVP = P×M[0]`
`state.matrix.texture[n]`	*n*-я матрица преобразования текстурных координат. Параметр *n* необязателен
`state.matrix.palette[n]`	Палитровая матрица *n*-го модельного преобразования
`state.matrix.program[n]`	*n*-я матрица программы

При этом к имени матрицы из табл. 16.7 может быть добавлено `.inverse`, `.transpose` и `.invtrans` для обозначения того, что указанную матрицу следует обратить, транспонировать или обратить и транспонировать.

Также к имени может быть добавлено `.row [n]` для получения заданного столбца матрицы как 4-мерного вектора.

Результаты выполнения вершинная программа заносит в выходные регистры, приведенные в табл. 16.8.

Таблица 16.8. *Регистры результата*

Имя	Компоненты	Комментарий
`result.color`	(*r*, *g*, *b*, *a*)	Выходной цвет
`result.depth`	(*, *, *d*, *)	Глубина точки

Если в выходной регистр цвета (`result.color`) не было произведено записи, то выходное значение цвета становится неопределенным.

Фрагментная программа не обязана производить запись в выходной регистр глубины (`result.depth`). В случае, когда в этот регистр не было ни одной операции записи, выходным значением глубины будет значение, полученное при растеризации примитива.

Однако в том случае, когда в фрагментной программе присутствует хотя бы одна команда записи в регистр глубины, программа должна всегда производить запись в него.

Обратите внимание на то, для записи в регистр используется нормированное значение глубины (т. е. приведенное из отрезка [*zNear*, *zFar*] в отрезок [0, 1].

Структура фрагментной программы

Фрагментная программа представляет собой набор строк, каждой строке соответствует какая-то команда (также она может быть пустой или содержать комментарий). Листинг 16.4 иллюстрирует пример вершинной программы, вычисляющей фоновое, диффузное и бликовое освещение от одного бесконечно удаленного источника света.

Листинг 16.4. Пример фрагментной программы

```
!!ARBfp1.0
ATTRIB  h          = fragment.texcoord [1];
PARAM   amb        = { 0, 0, 0.5 };
PARAM   one        = 1;
PARAM   two        = 2;
PARAM   shininess = 10;
```

```
PARAM   specColor = { 1, 1, 0 };
TEMP    n, hn, color, h2, n2;
                                    # get normal
TEX     n, fragment.texcoord [0], texture [0], 2D;
MAD     n, n, two, -one;
                                    # normalize n
DP3     n2.w, n, n;
RSQ     n2.w, n2.w;
MUL     n2.xyz, n, n2.w;
                                    # normalize h
DP3     h2.w, h, h;
RSQ     h2.w, h2.w;
MUL     h2.xyz, h, h2.w;
                                    # compute (n,h) ^ shininess
DP3_SAT hn.a, n2, h2;               # compute max ( (n,h), 0 )
LG2     hn.a, hn.a;
MUL     hn.a, hn.a, shininess;
EX2     hn.a, hn.a;                 # compute max ((h,n), 0) ^ shininess
                                    # return color
MUL     color, specColor, hn.a;     # return amb +
                                    # specColor*max((h,n),0)^shininess
ADD_SAT result.color, color, amb;
END
```

Первая строка программы должна быть `!!ARBfp1.0`. Это значит, что далее идет фрагментная программа, соответствующая версии 1.0. Каждая команда (кроме команды `END`) должна завершаться точкой с запятой. Завершается фрагментная программа командой `END`. Весь текст, идущий после символа `#` и до конца строки, является комментарием и игнорируется.

Обратите внимание на то, что все имена команд, ключевые слова, используемые при объявлении переменных, а также слова `fragment`, `state`, `program` и `result` являются зарезервированными.

В начале фрагментной программы можно задать набор опций при помощи команды `OPTION`.

Фрагментные программы поддерживают две группы опций: точности (*precision*) и тумана (*fog*).

При помощи опций точности можно явно задать свои пожелания: высокая точность (`ARB_precision_hint_nicest`) или скорость (`ARB_precision_hint_`

fastest). Однако это является только пожеланием, необязательным к выполнению.

Опции тумана позволяют вынести явное наложение тумана за пределы фрагментной программы. Тогда на выходное значение цвета будет автоматически накладываться туман при помощи одного из трех стандартных для OpenGL способов, задаваемых опциями ARB_fog_exp, ARB_fog_exp2 и ARB_fog_linear.

Обратите внимание, что опции внутри каждой группы являются взаимоисключающими. Вот пример задания опции:

```
OPTION ARB_precision_hint_fastest;
```

Идентификаторы

Идентификаторами в вершинной программе могут быть произвольные последовательности латинских букв, цифр, символа подчеркивания _ и знака доллара $, начинающиеся не с цифры. Важно, что идентификаторы в вершинных программых чувствительны к регистрам букв, т. е. a1 и A1 различаются.

Каждый идентификатор переменной до его использования должен быть объявлен, само объявление при этом может находиться в любом месте программы (но обязательно до первого обращения к данному идентификатору).

Временные переменные

Для хранения промежуточных значений в вершинных программах могут быть временные переменные — 4-мерные вещественные векторы, которые должны быть заранее объявлены командой TEMP.

Все временные переменные доступны вершинной программе как для чтения, так и для записи. Примеры:

```
TEMP flag;
TEMP a,b,c;
```

Начальные значения временных переменных (как и выходных) не определены и зависят от реализации драйвера.

Параметры

В качестве параметров могут выступать как 4-мерные векторы, так и их массивы.

Параметры могут задаваться как явно (через команду PARAM), так и неявно (путем непосредственной подстановки в текст).

При этом параметры представляют собой постоянные (на время выполнения вершинной программы) величины, т. е. они доступны только для чтения. Далее представлен пример задания параметров.

Пример 1

```
PARAM a = { 1, 2, 3, 4 };                    # vector (1, 2, 3, 4)
PARAM b = { 3 };                             # vector (3, 0, 0, 1)
PARAM c = { 3, 4 };                          # vector (3, 4, 0, 1)
PARAM e = 3;                                 # vector (3, 3, 3, 3)
                                             # array of two vectors
PARAM arr [2] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 } };
ADD a, b, { 1, 2, 3, 4 };
ADD a, c, 3;
```

Как и для вершинных программ, матрица задается в виде массива из четырех векторов-столбцов. Способы задания и работы с матрицами полностью аналогичны случаю вершинных программ.

Вот несколько примеров параметров, основанных на матрицах:

```
PARAM m0   = state.matrix.modelview[1].row[0];
PARAM m1   = state.matrix.projection.transpose.row[3];
PARAM m2[] = { state.matrix.program[0].row[1..2] };
PARAM m3[] = { state.matrix.program[0].transpose };
```

Можно связать параметр с каким-либо локальным параметром или параметром окружения:

```
PARAM c       = program.local [0];
PARAM mat [4] = program.env [0..3];
PARAM ambient = state.material.ambient;
```

Выходные значения

Выходные параметры доступны только для записи и задаются командой `OUTPUT`:

```
OUTPUT oCol = result.color.primary;
```

Фактически эта команда создает ссылку на соответствующий выходной регистр.

Для уже существующего параметра можно задать второе имя (`alias`):

```
ALIAS b = a;
```

Атрибуты

При помощи команды ATTRIB можно явно описать атрибут, дав ему другое имя.

```
ATTRIB    tangent  = fragment.texcoord [1];
ATTRIB    binormal = fragment.texcoord [2];
```

Подобная запись позволяет повысить читаемость и сразу показать, какими атрибутами пользуется данная программа.

Система команд

Всего поддерживается 33 различных инструкции, работающие с 4-мерными вещественными векторами. Каждая такая команда имеет 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.

Все команды делятся на три категории:

1. Скалярные операции (COS, EX2, LG2, RCP, RSQ, SIN, SCS, POW).
2. Операции с текстурами (TEX, TXP, TXB, KIL).
3. Векторные операции.

Векторные операции делятся на арифметико-логические (ABS, FLR, FRC, SUB, XPD, CMP, LPR, MAD, MOV, ADD, DP3, DP4) и специальные (LIT, DPH, DST).

Вот простые примеры команд для фрагментной программы:

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

Необязательный знак "минус" устанавливает в качестве источника не сам регистр, а 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 произвольной перестановки его компонентов, например:

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

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

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

Наряду с именами компонентов `xyzw` возможны также имена `rgba`, но их смешение в одной маске или перестановка компонентов регистра не допускается, т. е. конструкция `R1.xyr` недопустима.

Рассмотрим теперь саму систему команд. В табл. 16.9 приведены скалярные и векторные команды.

Таблица 16.9. *Скалярные и векторные команды*

Название	**Синтаксис**	**Комментарий**
ABS Вычисление модуля	`ABS dest, src;`	Данная команда осуществляет покомпонентное вычисление модуля `dest=fabs(src)`
ADD Сложение	`ADD dest, src0, src1;`	Данная команда покомпонентно складывает два параметра и записывает сумму в результат `dest=src0+src1`

CMP
Сравнение
	`CMP dest, src0,src1, src2;`	Данная команда покомпонентно, в зависимости от знака соответствующей компоненты `src0`, выбирает очередную компоненту 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`

COS Вычисление косинуса угла	`COS dest, src.C;`	Данная команда осуществляет покомпонентное вычисление косинуса скалярного аргумента и записывает его значение во все разрешенные компоненты результата, угол выражается в радианах и не обязан лежать в [–PI, PI]
DP3 Вычисление скалярного произведения по первым трем компонентам	`DP3 dest, src0, src1;`	Данная команда вычисляет скалярное произведение источников как трехмерных векторов `dest=src0.x·src1.x + src0.y·src1.y + src0.z·src1.z`
DP4 Вычисление скалярного произведения	`DP4 dest, src0, src1;`	Данная команда вычисляет скалярное произведение 4-мерных векторов и записывает результат во все компоненты `dest` `dest=src0.x·src1.x+src0.y·src1.y+src0.z·src1.z+src0.w·src1.w`

Таблица 16.9 *(продолжение)*

Название	Синтаксис	Комментарий
DPH Вычисление однородного скалярного произведения	`DPH dest, src0, src1;`	Данная команда служит для вычисления однородного скалярного произведения `dest=src0.x·src1.x + src0.y·src1.y + src0.z·src1.z + src1.w`

DST
Вычисление расстояния
	`DST dest, src0, src1;`	Данная команда эффективно вычисляет вектор `dest=(1,d,d^2,1/d)` по двум значениям
		`src0=(*, d^2, d^2, *)`
		`src1=(*,1/d,*,1/d)`
		Через * обозначено, что соответствующее значение не важно

EX2 Приближенное вычисление 2 в степени	`EX2 dest, src.C;`	Для заданного скалярного операнда `src.C` вычисляет приближенное значение `2^src.C` и записывает во все компоненты регистра `dest`
FLR Вычисление floor	`FLR dest, src;`	Осуществляет покомпонентное вычисление целой части (`floor`) (наибольшего целого, не превосходящего аргумента) от компонентов источника
FRC Вычисление дробной части	`FRC dest, src;`	Осуществляет покомпонентное вычисление дробной части. Дробная часть `frac(x) = x - floor(x)`
LG2 Приближенное вычисление логарифма по основанию 2	`LG2 dest, src.C;`	Вычисляет приближенное значение логарифма по основанию 2 от скалярного аргумента и записывает во все компоненты источника

LIT
Вычисление коэффициентов освещения
	`LIT dest, src;`	Служит для ускорения вычисления диффузной и бликовой освещенности в вершинах
		`src.x=(n,l)`
		`src.y=(n,h)`
		`src.w=p` — степень, приводится к отрезку [–128,128]
		
		На выходе
		`dest.x=1`
		`dest.y=max(0,(n,l))`
		`dest.z=(n.l)>0?roughAppr2ToX(max(0,(n,h))^p):0`
		`dest.w=1`

Таблица 16.9 *(продолжение)*

Название	Синтаксис	Комментарий

LRP
Линейная интерполяция
	`LRP dest, src0, src1, src2;`	Осуществляет покомпонентную линейную интерполяцию 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;`	Вычисляет `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 Покомпонентное умножение	`MUL dest, src0, src1;`	Вычисляет покомпонентное произведение `dest=src0·src1`
POW Возведение в степень	`POW dest, src0.C1, src1.C2;`	Вычисляет приближенное значение первого скалярного операнда, возведенного в степень второго скалярного операнда, и записывает результат во все компоненты регистра `dest` `apprPow(a,b)=apprExp2(b·apprLog2·a))` Обратите внимание, что 0 ^ 0 = 1
RCP Вычисление обратного значения	`RCP dest, src.C;`	Вычисляет приближенное значение обратного к скалярному операнду и записывает во все компоненты `dest` `dest=1/src.C`
RSQ Вычисление значения операнда в степени $^1/_2$	`RSQ dest, src.C;`	`dest=1/sqrt(fabs(src.C))`

SCS
Одновременное вычисление синуса и косинуса
	`SCS dest, src.C;`	`dest.x=cos (src.C)`
		`dest.y=sin(src.C)`
		Значение скалярного операнда должно лежать в [–PI, PI], компоненты *z* и *w* результата не определены

Таблица 16.9 *(окончание)*

Название	Синтаксис	Комментарий

SGE
Установка по "больше или равно"
	`SGE dest, src0, src1;`	`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 Вычисление синуса	`SIN dest, src.C;`	`dest=sin(src.C)` Вычисление синуса от скалярного операнда, выражающего угол в радианах, угол не обязательно должен лежать в [–PI, PI]

SLT
Установка по "меньше"
	`SLT dest, src0, src1;`	`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 Вычисление разности	`SUB dest, src0, src1;`	Покомпонентное вычисление разности операндов `dest=src0-src1`

SWZ
"Перемешивание" компонент
	`SWZ dest, src, <extSwizzle>`	Данная команда позволяет произвольным образом перемешать компоненты (или их часть), в качестве источника для компоненты может выступать 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 | r | g | b | a)`

XPD Вычисление векторного произведения	`XPD dest, src0, src1;`	Данная команда осуществляет вычисление векторного произведения первых трех компонент первого операнда на первые три компоненты второго операнда и записывает в первые три компоненты `dest`. Значение `dest.w` не определено

Таблица 16.10. *Текстурные команды*

Название	Синтаксис	Комментарий
TEX	`TEX dest, src0, texture [n], type;`	Данная команда осуществляет выборку из текстуры типа `type` с *n*-го текстурного блока, используя в качестве текстурных координат первые компоненты параметра `src0`, и записывает результат в `dest`. Параметр `type` принимает одно из следующих значений: `1D`, `2D`, `3D`, `cube` и `rectangle`

Таблица 16.10 *(окончание)*

Название	Синтаксис	Комментарий
TXP	`TXP dest, src0, texture [n], type;`	Данная команда осуществляет выборку из текстуры типа `type` с *n*-го текстурного блока, используя в качестве текстурных координат первые компоненты параметра `src0`, деленные на `src0.w`, и записывает результат в `dest`. Параметр `type` принимает одно из следующих значений: `1D`, `2D`, `3D`, `cube` и `rectangle`
TXB	`TXB dest, src0, texture [n], type;`	Данная команда осуществляет выборку из текстуры типа `type` с *n*-го текстурного блока, используя в качестве текстурных координат первые компоненты параметра `src0`, и записывает результат в `dest` Параметр `type` принимает одно из следующих значений: `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 вершинной программы	`KIL src;`	Данная команда в случае, когда хотя бы одна из компонент операнда меньше нуля, прерывает выполнение фрагментной программы для данного фрагмента. При этом дальнейшие шаги конвейера для данного фрагмента будут пропущены

К командам может быть добавлен префикс `_SAT`, обозначающий, что выходные компоненты результата перед записью в регистр-приемник должны быть отсечены по отрезку [0, 1].

Хотя при обращении к текстуре явно задается ее тип (*texture target*), но использовать разные типы для одного и того же текстурного блока в пределах одной программы нельзя.

Примеры

В листингах 16.5—16.8 приведены примеры использования команд фрагментной программы.

Листинг 16.5. Нормирование трехмерного вектора

```
DP3 result.w, v, v;                 # result.w = vx^2+vy^2+vz^2
RSQ result.w, result.w;             # result.w = 1/sqrt(vx^2+vy^2+vz^2)
MUL result.xyz, result.w, vr;       # normalize by multiplying by 1/length
```

Листинг 16.6. Приведение угла к отрезку [–PI, PI]

```
PARAM myParams = { 0.5, -3.14159, 6.28319, 0.15915 };
                                    # a = (a/(2*PI))+0.5
MAD myOperand.x, myOperand.x, myParams.w, myParams.x;
FRC myOperand.x, myOperand.x;       # a = frac(a)
                                    # a = (a*2*PI)-PI
MAD myOperand.x, myOperand.x, myParams.z, myParams.y
```

Листинг 16.7. Модулирование текстуры

```
!!ARBfp1.0
                                    # Simple program to show how to
code up the default
                                    # texture environment
ATTRIB tex = fragment.texcoord;     # first set of texture coordinates
ATTRIB col = fragment.color.primary; # diffuse interpolated color
OUTPUT out = result.color;
TEMP     tmp;

TXP     tmp, tex, texture, 2D;      # sample the texture
MUL     out, tmp, col;              # perform the modulation
END
```

Листинг 16.8. Различные способы вычисления затуманивания

```
# Exponential fog
# f = exp(-d*z)
#
ATTRIB  fogCoord = fragment.fogcoord.x;
PARAM   p        = {DENSITY/LN(2), NOT USED, NOT USED, NOT USED};
PARAM    fogColor = state.fog.color;
TEMP     fogFactor;
MUL      fogFactor.x, p.x, fogCoord.x;
EX2_SAT  fogFactor.x, -fogFactor.x;
LRP      result.color.rgb, fogFactor.x, finalColor, fogColor;
```

```
#
# 2nd-order Exponential fog
# f = exp(-(d*z)^2)
#
ATTRIB   fogCoord = fragment.fogcoord.x;
PARAM    p        = {DENSITY/SQRT(LN(2)), NOT USED, NOT USED, NOT USED};
PARAM    fogColor = state.fog.color;
TEMP     fogFactor;

MUL      fogFactor.x, p.x, fogCoord.x;
MUL      fogFactor.x, fogFactor.x, fogFactor.x;
EX2_SAT  fogFactor.x, -fogFactor.x;
LRP      result.color.rgb, fogFactor.x, finalColor, fogColor;

#
# Linear fog
# f = (end-z)/(end-start)
#
ATTRIB    fogCoord = fragment.fogcoord.x;
PARAM     p        = {-1/(END-START),END/(END-START),NOT USED,NOT USED};
PARAM     fogColor = state.fog.color;
TEMP      fogFactor;

MAD_SAT fogFactor.x, p.x, fogCoord.x, p.y;
LRP     result.color.rgb, fogFactor.x, finalColor, fogColor;
```

Примеры использования фрагментных программ

Реализация попиксельного бликового освещения

Рассмотрим, каким образом можно с помощью фрагментной программы вычислить попиксельное бликовое (*specular*) освещение.

При этом мы будем считать, что вектор h передается нам в виде текстурных координат для одного из текстурных блоков соответствующей вершинной программой, причем он задан в касательном пространстве.

Тогда вектор нормали *n* можно брать из карты нормалей, но при этом, поскольку из текстуры мы получаем вектор в RGB-представлении, его необходимо отобразить обратно в координаты.

Также желательно провести его нормирование и нормирование вектора *h*, причем можно обойтись без нормирующих кубических карт, а сделать это непосредственно командами `DP3`, `RSQ` и `MUL`.

В результате мы получаем вектор нормали и вектор *h* сразу в касательном пространстве, и нам остается найти их скалярное произведение и возвести в степень.

Соответствующая фрагментная программа приводится в листинге 16.9.

Листинг 16.9. Фрагментная программа для вычисления бликового освещения

```
!!ARBfp1.0
#
# simple specular shader
# on entry:
#     fragment.texcoord [0] - texture coordinates
#     fragment.texcoord [1] - h
#
#     texture [0] - bump map
#
ATTRIB  h         = fragment.texcoord [1];
PARAM   amb       = { 0, 0, 0.5 };
PARAM   one       = 1;
PARAM   two       = 2;
PARAM   shininess = 10;
PARAM   specColor = { 1, 1, 0 };
TEMP    n, ln, hn, color, h2, l2, n2;
                                  # get normal perturbation
TEX     n, fragment.texcoord [0], texture [0], 2D;
MAD     n, n, two, -one;
                                   # normalize n
DP3     n2.w, n, n;
RSQ     n2.w, n2.w;
MUL     n2.xyz, n, n2.w;
                                   # normalize h
DP3     h2.w, h, h;
RSQ     h2.w, h2.w;
MUL     h2.xyz, h, h2.w;
```

```
                                    # compute (n,h) ^ shininess
DP3_SAT hn.a, n2, h2;               # compute max ( (n,h), 0 )
LG2     hn.a, hn.a;
MUL     hn.a, hn.a, shininess;
EX2     hn.a, hn.a;                 # compute max ((h,n), 0) ^ shininess
                                    # return color
MUL     color, specColor, hn.a;     # return amb +
                                    # specColor*max((h,n),0)^shininess
ADD_SAT result.color, color, amb;
END
```

Обратите внимание, что после получения значения вектора нормали из карты нормалей он переводится из RGB-представления при помощи всего одной команды (`MAD`).

```
TEX     n, fragment.texcoord [0], texture [0], 2D;
MAD     n, n, two, -one;
```

Заметьте также, что вычисление скалярного произведения осуществляется командой `DP3_SAT`, позволяющей сразу же отсечь получившееся значение по отрезку [0, 1].

```
DP3_SAT hn.a, n2, h2;               # compute max ( (n,h), 0 )
```

Вместо команд `LG2` и `EX2` возведение в степень можно выполнить командой `POW` или сразу вычислить освещенность при помощи команды `LIT`.

Заворачиваем фрагментную программу в класс

Для удобства работы с вершинными программами и доступа к их параметрам можно завернуть вершинную программу (вместе с методами загрузки, доступа к параметрам, запросам на ограничения) в класс, предоставляющий весь необходимый доступ, но при этом скрывающей внутренние детали.

Листинг 16.10 содержит описание класса, реализующего абстракцию вершинной программы и работы с ней.

Листинг 16.10. Описание класса `FragmentProgram`

```
class    FragmentProgram
{
protected:
    int                 errorCode;
```

```
    string              errorString;
    unsigned            id;                 // program id
public:
    ParamArray          local;              // local parameters
    static ParamArray   env;                // environment params
    static int          activeProgram;
    static int          maxAluInstructions ();
    static int          maxTexInstructions ();
    static int          maxTexIndirections ();
    static int          maxInstructions    ();
    static int          maxLocalParams     ();
    static int          maxEnvParams       ();
    static int          maxTemporaries     ();
    static int          maxProgramAttribs  ();
    static  bool        isSupported ();
    FragmentProgram  ();
    FragmentProgram  ( const char * fileName );
    ~FragmentProgram ();
    unsigned    getId () const
    {
        return id;
    }
    string    getErrorString () const
    {
        return errorString;
    }
    void    bind   ();
    void    unbind ();
    void    enable  ();
    void    disable ();
    bool    load ( Data * data );
    bool    load ( const char * fileName );
};
```

Поскольку данный класс очень похож на рассмотренный ранее `VertexProgram`, его полный исходный текст здесь не приводится (соответствующий файл — FragmentProgram.cpp — вы найдете на компакт-диске).

Реализация общего случая освещения при помощи вершинной и фрагментной программ

С помощью двух типов рассмотренных программ легко реализовать общий случай освещения поверхности. В нем сразу четыре текстурные карты, причем одна из них задает фоновую освещенность поверхности (т. е. фактически ее собственную светимость). Карта нормалей служит для задания микрорельефа на поверхности, а диффузная и бликовая карты — для управления вкладом диффузной и бликовой компонент в общую освещенность поверхности.

Фактически мы будем использовать следующую модель освещенности:

$$I = I_{amb} + \max(1 - r^2, 0) \cdot \left(I_{diff} \cdot \max\left((l,\ n), 0\right) + I_{spec} \cdot C_{light} \cdot \max\left((h,\ n), 0\right)^p \right).$$

Фрагментная программа позволяет легко реализовать учет влияния расстояния до источника света на освещенность точки.

Вершинная программа (листинг 16.11) в этом случае аналогична уже рассмотренным ранее, она вычисляет необходимые векторы и переводит их в систему координат касательного пространства. Листинг 16.12 содержит соответствующую фрагментную программу.

Листинг 16.11. Вершинная программа для общего случая освещенности

```
!!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
#      vertex.texcoord [1] - tangent vector  (t)
#      vertex.texcoord [2] - binormal vector (b)
#
#      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] - l
#     result.texcoord [2] - h
#
#    We assume that object whose vertices are passed are transformed
#    by program [0] matrix
#    Instead of transforming vertex and TBN basis with this matrix
#    we transform light pos and eye pos with inverse matrix to leave
#    TBN intact.
#    This way we keep l, v and h vectors correct
#
ATTRIB    pos     = vertex.position;
PARAM    eye     = program.local [0];
PARAM    light   = program.local [1];
PARAM    mvp [4] = { state.matrix.mvp };
                                    # inverse rotation matrix
PARAM    mv0 [4] = { state.matrix.program [0].inverse };
PARAM    half    = 0.5;
TEMP    l, l2, v, v2, h, h2, temp;
TEMP    lt, ht, lt2;
TEMP    vt, et;
                              # transform light position
DP4    lt2.x, light, mv0 [0];
DP4    lt2.y, light, mv0 [1];
DP4    lt2.z, light, mv0 [2];
DP4    lt2.w, light, mv0 [3];
                     # transform eye position
DP4    et.x, eye, mv0 [0];
DP4    et.y, eye, mv0 [1];
DP4    et.z, eye, mv0 [2];
DP4    et.w, eye, mv0 [3];
                              # compute l (vector to light)
ADD    l, -pos, lt2;          # l = light - pos
                              # transform it into tangent space
DP3    lt.x, l, vertex.texcoord [1];
```

```
DP3    lt.y, l, vertex.texcoord [2];
DP3    lt.z, l, vertex.normal;
MOV    lt.w, l.w;
                            # store it into texcoord [1]
MOV    result.texcoord [1], lt;
                            # compute v (vector to viewer)
ADD    v, -pos, et;         # 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)/2
ADD    h, l, v;
MUL    h, h, half;
                            # transform it into tangent space
DP3    ht.x, h, vertex.texcoord [1];
DP3    ht.y, h, vertex.texcoord [2];
DP3    ht.z, h, vertex.normal;
MOV    ht.w, h.w;
                            # store it into texcoord [2]
MOV    result.texcoord [2], ht;
                            # store 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, vertex.position, mvp [0];
DP4    result.position.y, vertex.position, mvp [1];
DP4    result.position.z, vertex.position, mvp [2];
DP4    result.position.w, vertex.position, mvp [3];
                            # we're done
END
```

Задача фрагментной программы — по полученным через текстурные координаты значениям векторов найти диффузную и бликовую освещенность

точки и с помощью соответствующих текстур получить итоговую освещенность точки поверхности.

Обратите внимание на преобразование положения наблюдателя и источника света при помощи передаваемой матрицы:

```
                              # transform light position
DP4    lt2.x, light, mv0 [0];
DP4    lt2.y, light, mv0 [1];
DP4    lt2.z, light, mv0 [2];
DP4    lt2.w, light, mv0 [3];
                              # transform eye position
DP4    et.x, eye, mv0 [0];
DP4    et.y, eye, mv0 [1];
DP4    et.z, eye, mv0 [2];
DP4    et.w, eye, mv0 [3];
```

Листинг 16.12. Фрагментная программа для вычисления освещения в общем случае

```
!!ARBfp1.0
#
# simple specular shader
# on entry:
#    fragment.texcoord [0] - texture coordinates
#    fragment.texcoord [1] - l in tangent space
#    fragment.texcoord [2] - h in tangent space
#
#    texture [0] - bump map
#    texture [1] - emission (ambient) map
#    texture [2] - diffuse map
#    texture [3] - specular (gloss) map
#
ATTRIB    l         = fragment.texcoord [1];
ATTRIB    h         = fragment.texcoord [2];
PARAM    amb       = { 0, 0, 0.5 };
PARAM    one       = 1;
PARAM    two       = 2;
```

```
PARAM    shininess = 20;
PARAM    specColor = { 1, 1, 1 };
PARAM    scale     = 0.03;
TEMP    n, ln, hn, color, h2, l2, n2, temp, diffuse, dist, atten;
                                  # get normal perturbation
TEX        n, fragment.texcoord [0], texture [0], 2D;
MAD        n, n, two, -one;
                                  # normalize n
DP3        n2.w, n, n;
RSQ        n2.w, n2.w;
MUL        n2.xyz, n, n2.w;
                                  # normalize l
DP3        dist.w, l, l;
RSQ        l2.w, dist.w;
MUL        l2.xyz, l, l2.w;
                                  # compute distance attenuation
MUL        dist.w, dist.w, scale;
SUB_SAT    atten.w, one.w, dist.w;    # as  clamp ( 1 - (l&l) )
                                  # normalize h
DP3        h2.w, h, h;
RSQ        h2.w, h2.w;
MUL        h2.xyz, h, h2.w;
                                  # compute (n,l) * diffuse
TEX        temp, fragment.texcoord [0], texture [2], 2D;
DP3_SAT    ln.a, l2, n2;
MUL        diffuse, temp, ln.a;
                                  # now diffuse holds diffuse lighting
                                  # add distance attenuation
MUL        diffuse, diffuse, atten.w;
                                  # compute (n,h) ^ shininess
DP3_SAT    hn.a, n2, h2;       # compute max ( (n,h), 0 )
LG2        hn.a, hn.a;
MUL        hn.a, hn.a, shininess;
EX2        hn.a, hn.a;
                                  # compute max ((h,n), 0) ^ shininess
MUL        color, specColor, hn.a;
TEX        temp, fragment.texcoord [0], texture [3], 2D;
```

```
MUL        color, temp, color;     # now color is specular lighting
                                   # add distance attenuation
MUL        color, color, atten.w;
                                   # get emission
TEX        temp, fragment.texcoord [0], texture [1], 2D;
ADD_SAT    color, color, temp;
                                   # return emission + diffuse + specular
ADD_SAT        result.color, color, diffuse;
END
```

Листинг 16.13 содержит текст программы на C++ для вывода тора на основе общей модели освещения.

Листинг 16.13. Программа вывода тора с общей моделью освещения

```
//
// Sample to show generic lighting model in OpenGL
//
#include    "libExt.h"
#include    <glut.h>
#include    <stdio.h>
#include    <stdlib.h>
#include    "libTexture.h"
#include    "Vector3D.h"
#include    "Vector2D.h"
#include    "Torus.h"
#include    "VertexProgram.h"
#include    "FragmentProgram.h"
Vector3D    eye   ( 7, 5, 7 );        // camera position
Vector3D    light ( 5, 0, 4 );        // light position
unsigned    normCubeMap;              // normalization cubemap id
unsigned    bumpMap;                  // normal map
unsigned    ambMap;                   // ambient texture
unsigned    diffuseMap;               // diffuse texture
unsigned    glossMap;                 // gloss (specular) map
float       angle = 0;
Torus       torus ( 1, 3, 30, 30 );
```

```
Vector3D    rot ( 0, 0, 0 );
int         mouseOldX = 0;
int         mouseOldY = 0;
VertexProgram        vertexProgram;
FragmentProgram      fragmentProgram;
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
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();
    glTranslatef       ( light.x, light.y, light.z );
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glActiveTextureARB ( GL_TEXTURE2_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glActiveTextureARB ( GL_TEXTURE3_ARB );
    glDisable          ( GL_TEXTURE_2D );
    glColor4f          ( 1, 1, 1, 1 );
    glutSolidSphere    ( 0.1f, 15, 15 );
    glPopMatrix        ();
                        // setup texture units
                        // bind bump (normal) map to
                        // texture unit 0
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glEnable           ( GL_TEXTURE_2D );
```

```
    glBindTexture      ( GL_TEXTURE_2D, bumpMap );
                        // bind ambient map to unit 1
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, ambMap );
                        // bind diffuse map to unit 2
    glActiveTextureARB ( GL_TEXTURE2_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, diffuseMap );
                        // bind gloss (specular) map
    glActiveTextureARB ( GL_TEXTURE3_ARB );
    glEnable           ( GL_TEXTURE_2D );
    glBindTexture      ( GL_TEXTURE_2D, glossMap );
    glMatrixMode   ( GL_MATRIX0_ARB );
    glLoadIdentity ();
    glRotatef      ( rot.x, 1, 0, 0 );
    glRotatef      ( rot.y, 0, 1, 0 );
    glRotatef      ( rot.z, 0, 0, 1 );
    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   ();
    torus.draw  ();
    vertexProgram.disable   ();
    fragmentProgram.disable ();
    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, 1 );              // 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 );
    light.x = 2*cos ( angle );
    light.y = 2*sin ( angle );
    light.z = 3 + 0.3 * sin ( angle / 3 );
                            // setup data
    vertexProgram.local [0] = eye;
    vertexProgram.local [1] = light;
    glutPostRedisplay ();
}
int main ( int argc, char * argv [] )
{
                        // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 500, 500 );
                        // create window
    int    win = glutCreateWindow ( "OpenGL per-pixel specular lit torus
via fragment & vertex programs" );
                        // register handlers
    glutDisplayFunc  ( display );
    glutReshapeFunc  ( reshape );
    glutKeyboardFunc ( key     );
    glutMouseFunc    ( mouse   );
    glutMotionFunc   ( motion  );
    glutIdleFunc     ( animate );
    init ();
    printfInfo ();
    if ( !isExtensionSupported ( "GL_ARB_multitexture" ) )
    {
        printf ( "ARB_multitexture NOT supported.\n" );
        return 1;
    }
```

```
    if ( !isExtensionSupported ( "GL_ARB_vertex_program" ) )
    {
        printf ( "GL_ARB_vertex_program NOT supported" );
        return 2;
    }
    if ( !isExtensionSupported ( "GL_ARB_fragment_program" ) )
    {
        printf ( "GL_ARB_fragment_program NOT supported" );
        return 3;
    }
    initExtensions ();
    bumpMap     = createNormalMapFromHeightMap ( false,
                          "../Textures/Bumpmaps/oxidatedBump.png", 12 );
    ambMap      = createTexture2D              ( true,
                          "../Textures/emission.jpg" );
    diffuseMap  = createTexture2D              ( true,
                          "../Textures/oxidated.jpg" );
    glossMap    = createTexture2D              ( true,
                          "../Textures/oxidatedGloss.tga" );
    if ( !vertexProgram.load ( "specular3.vp" ) )
    {
        printf ( "Error loading vertex program:\n %s\n",
                         vertexProgram.getErrorString ().c_str () );
        return 1;
    }
    if ( !fragmentProgram.load ( "specular3.fp" ) )
    {
        printf ( "Error loading fragment program:\n %s\n",
                        fragmentProgram.getErrorString ().c_str () );
        return 2;
    }
    glutMainLoop ();
    return 0;
}
```

На рис. 16.3 приводится изображение, полученное при помощи этих программ.

Рис. 16.3. Изображение тора с общей моделью освещения

Реализация анизотропного освещения

Все рассмотренные ранее модели освещения (диффузное и бликовое) являются так называемыми изотропными, в них освещенность точки определяется только взаимным расположением векторов n, l, v и h. При этом ориентация самой поверхности (при условии сохранения углов между этими векторами) не играет никакой роли.

Однако в реальной жизни мы часто сталкиваемся и с анизотропными поверхностями, т. е. теми, для которых важной оказывается еще и ориентация самой поверхности относительно вектора нормали.

Простейшим примером такой поверхности является компакт-диск с его структурой кольцевых дорожек (рис. 16.4).

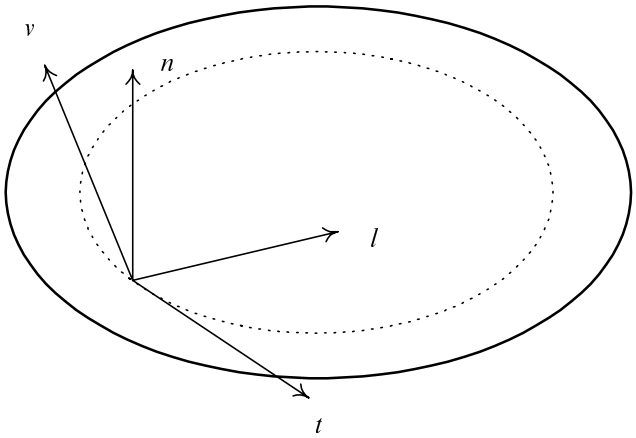


Рис. 16.4. Структура поверхности компакт-диска

Наиболее распространенная модель анизотропной поверхности строится в предположении, что структура поверхности может быть задана при помощи семейства линий, проходящих через каждую точку поверхности.

В случае компакт-диска таким семейством линий является семейство окружностей.

Тогда удобно характеризовать свойства поверхности в точке при помощи касательного вектора к линии семейства, проходящего через данную точку. Тем самым мы получаем описание структуры поверхности при помощи поля касательных (*tangent map*).

Для выражения диффузной и бликовой освещенности используется уже не нормаль к поверхности, а соответствующий касательный вектор t.

Чаще всего диффузная и бликовая составляющие освещенности описываются следующими уравнениями:

$$I_d = C \cdot K_d \cdot \sqrt{1 - (t,\ l)^2}\ ,$$

$$I_s = K_s \cdot \left(1 - (t,\ h)^2\right)^{p/2}\ .$$

Здесь векторы l и h имеют тот же смысл, что и для ранее рассмотренных моделей.

Для реализации освещения поверхности при помощи данной модели необходима вершинная программа для нахождения векторов l и h и перевода их в касательное пространство.

Непосредственно вычисление освещенности по рассмотренным формулам осуществляется фрагментной программой, пример которой приводится в листинге 16.14.

Листинг 16.14. Фрагментная программа для вычисления анизотропного освещения

```
!!ARBfp1.0
#
# simple anisotropic fragment shader
# on entry:
#     fragment.texcoord [0] - texture coordinates
#     fragment.texcoord [1] - l in tangent space
#     fragment.texcoord [2] - h in tangent space
#
#     texture [0] - bump map
#     texture [1] - diffuse map
#     texture [2] - specular (gloss) map
#
ATTRIB   l          = fragment.texcoord [1];
ATTRIB   h          = fragment.texcoord [2];
PARAM    amb        = { 0, 0, 0.5 };
PARAM    one        = 1;
PARAM    two        = 2;
PARAM    shininess = 20;
PARAM    specColor = { 1, 1, 1 };
PARAM    scale      = 0.03;
PARAM    diffusePower  = 1;
PARAM    specularPower = 30;
TEMP     t, t2, dots, color, h2, l2, temp, diffuse, dist, atten;
TEX    t, fragment.texcoord [0], texture [0], 2D;    # get tangent
MAD    t, t, two, -one;                       # normalize n
DP3        dist.w, l, l;
RSQ        l2.w, dist.w;
MUL        l2.xyz, l, l2.w;
                # compute distance attenuation
DP3        h2.w, h, h;
RSQ        h2.w, h2.w;
MUL        h2.xyz, h, h2.w;
DP3    dots.x, l2, t;
```

```
DP3     dots.y, h2, t;
MUL     dots.xy, dots, dots;
ADD     dots.xy, one, -dots;
POW     dots.x, dots.x, diffusePower.x;
POW     dots.y, dots.y, specularPower.x;
                # diffuse
TEX         temp, fragment.texcoord [0], texture [1], 2D;
MUL         diffuse, temp, dots.x;
                # compute (n,h) ^ shininess
MAD_SAT     result.color, specColor, dots.y, diffuse;
END
```

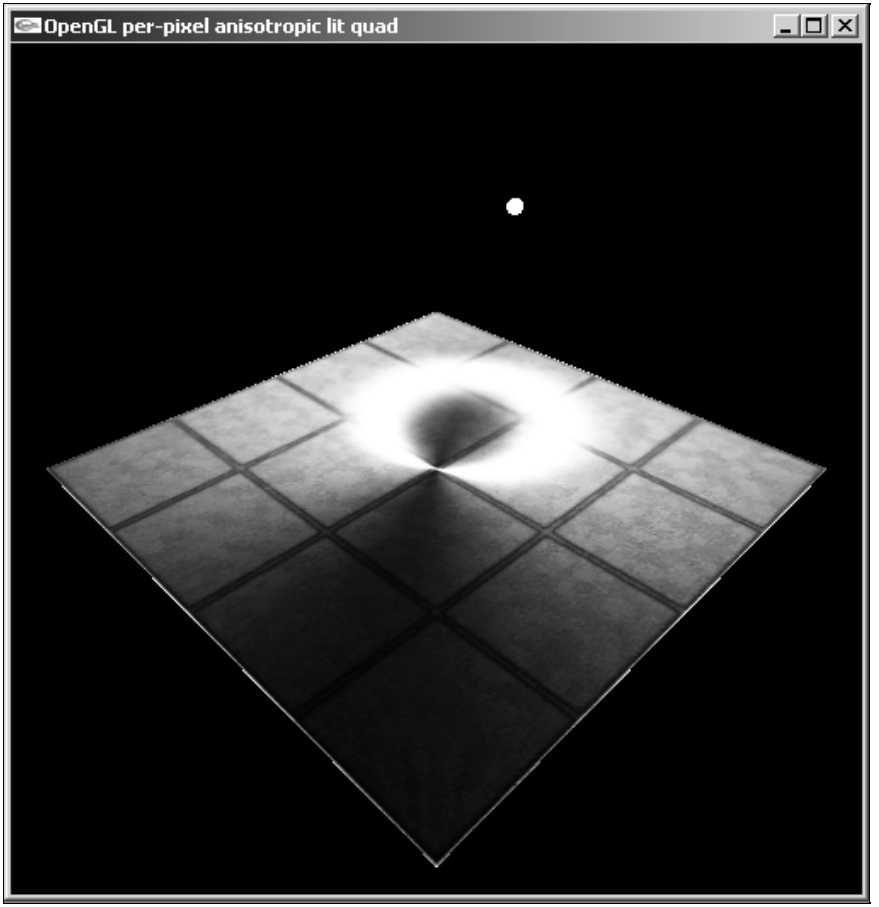


Рис. 16.5. Изображение анизотропной поверхности

На рис. 16.5 приводится изображение, полученное при помощи данной фрагментной программы. Полностью текст примера для анизотропного освещения (включая вершинную программу и программу на C++) можно найти на прилагаемом к книге компакт-диске (файлы aniso.vp и aniso.cpp).

Обработка изображений при помощи фрагментных программ

Еще одним интересным применением фрагментных программ является обработка изображений. При этом обрабатываемое изображение(я) просто помещается в текстуру(ы).

Одним из подобных применений может быть постобработка результатов рендеринга. Для этого изображение сцены строится в *p*-буфере, содержимое которого на стадии постобработки преобразуется как текстура.

Одним из простейших примеров является так называемый *sepia*-эффект (фильтр), позволяющий придать изображению желтоватый оттенок, характерный для старых фильмов. Самым простым способом реализации данного эффекта является нахождение для каждого тексела его яркости (которая обычно вычисляется как взвешенная сумма цветовых компонент) и умножения ее на заданный цвет. Более сложный вариант заключается в использовании полученной яркости для индексации в одномерную текстуру.

Яркость по RGB-значению цвета будем определять по формуле

$$I = 0{,}3R + 0{,}59G + 0{,}11B.$$

Как легко заметить, данная формула есть ни что иное, как скалярное произведение вектора цвета (RGB) на вектор весов.

В листинге 16.15 приводится простейшая реализация *sepia*-эффекта через фрагментную программу.

Листинг 16.15. Фрагментная программа для реализации *sepia*-эффекта

```
!!ARBfp1.0
#
# simple sepia fragment filter
# on entry:
#     fragment.texcoord [0] - texture coordinates
#
#     texture [0] - image to process
#
PARAM    luminance = { 0.3, 0.59, 0.11, 1 };
PARAM    color     = { 1, 0.89, 0.54, 1 };
```

```
TEMP     c, lum;
                # get texel
TEX        c, fragment.texcoord [0], texture [0], 2D;
                # get luminance
DP3    lum, c, luminance;
MUL    c.rgb, lum, color;
MOV    c.a, 1;
                # compute (n,h) ^ shininess
MOV    result.color, c;
END
```

Вершинная программа в данном случае очень проста (листинг 16.16).

Листинг 16.16. Вершинная программа для *sepia*-эффекта

```
!!ARBvp1.0
#
# simple vertex shader to setup data for fragment image processing
#
# on entry:
#      vertex.position
#      vertex.texcoord [0] - normal texture coordinates
#
# on exit:
#     result.texcoord [0] - texture coordinates
#
ATTRIB    pos     = vertex.position;
PARAM    mvp [4] = { state.matrix.mvp };
           # store 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, vertex.position, mvp [0];
DP4    result.position.y, vertex.position, mvp [1];
DP4    result.position.z, vertex.position, mvp [2];
DP4    result.position.w, vertex.position, mvp [3];
           # we're done
END
```

В листинге 16.17 приводится программа на C++, выводящая простую сцену в *p*-буфер и после этого вызывающая фрагментную программу для обработки результатов рендеринга.

Листинг 16.17. Рендеринг сцены с постобработкой

```
//
// Sample to image postprocessing via p-buffer and fragment 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    "PBuffer.h"
#include    "VertexProgram.h"
#include    "FragmentProgram.h"
Vector3D    eye   ( -0.5, -0.5, 1.5 );    // camera position
unsigned    decalMap;                     // decal (diffuse) texture
unsigned    stoneMap;
unsigned    teapotMap;
float       angle     = 0;
float       rot       = 0;
bool        useFilter = true;
VertexProgram      vertexProgram;
FragmentProgram    fragmentProgram;
PBuffer        pbuffer ( 512, 512, PBuffer :: modeTextureMipmap |
                         PBuffer :: modeAlpha | PBuffer :: modeDepth |
                         PBuffer :: modeTexture2D, true );
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 ()
{
                            // select the projection matrix
    glMatrixMode   ( GL_PROJECTION );
    glPushMatrix   ();       // store the projection matrix
    glLoadIdentity ();       // reset the projection matrix
                            // set up an ortho screen
    glOrtho        ( 0, 512, 0, 512, -1, 1 );
                            // select the modelview matrix
    glMatrixMode   ( GL_MODELVIEW );
    glPushMatrix   ();       // store the modelview matrix
    glLoadIdentity ();       // reset the modelview matrix
}
void    endOrtho ()
{
    glMatrixMode ( GL_PROJECTION );    // select the projection matrix
    glPopMatrix  ();                   // restore projection matrix
    glMatrixMode ( GL_MODELVIEW );     // select the modelview matrix
```

```
    glPopMatrix  ();                    // restore projection matrix
}
void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    renderToBuffer ();
    if ( !pbuffer.bindToTexture () )
        pbuffer.printLastError ();
    startOrtho ();
    glBindTexture ( GL_TEXTURE_2D, pbuffer.getTexture () );
    if ( useFilter )
    {
        vertexProgram.enable   ();
        vertexProgram.bind     ();
        fragmentProgram.enable ();
        fragmentProgram.bind   ();
    }
    glBegin ( GL_QUADS );
        glTexCoord2f ( 0, 0 );
        glVertex2f   ( 0, 0 );
        glTexCoord2f ( 1,   0 );
        glVertex2f   ( 511, 0 );
        glTexCoord2f ( 1, 1 );
        glVertex2f   ( 511, 511 );
        glTexCoord2f ( 0, 1 );
        glVertex2f   ( 0, 511 );
    glEnd   ();
    if ( useFilter )
    {
        vertexProgram.disable   ();
        fragmentProgram.disable ();
    }
    endOrtho ();
    if ( !pbuffer.unbindFromTexture () )
        pbuffer.printLastError ();
    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 );
    glutPostRedisplay ();
}
void    initPBuffer ()
{
    static    bool    inited = false;
    if ( inited )
        return;
```

```
    inited = true;

    if ( !pbuffer.create () )
    {
        printf ( "create error\n" );
        pbuffer.printLastError ();
    }
    pbuffer.makeCurrent ();
    init ();
    reshape ( pbuffer.getWidth (), pbuffer.getHeight () );
    pbuffer.restoreCurrent ();
}
void    renderToBuffer ()
{
    if ( pbuffer.isLost () )
        pbuffer.create ();
    if ( !pbuffer.makeCurrent () )
        printf ( "makeCurrent failed\n" );
    glClearColor ( 0, 0, 1, 1 );
    glClear      ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    reshape ( pbuffer.getWidth (), pbuffer.getHeight () );
    displayBoxes ();
    pbuffer.restoreCurrent ();
}
int main ( int argc, char * argv [] )
{
                        // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 512, 512 );
                        // create window
    int    win = glutCreateWindow ( "OpenGL image postprocessing - sepia"
);
                        // register handlers
    glutDisplayFunc  ( display    );
    glutReshapeFunc  ( reshape    );
    glutKeyboardFunc ( key        );
    glutSpecialFunc  ( specialKey );
    glutIdleFunc     ( animate    );
```

```
    init ();
    initExtensions ();
    assertExtensionsSupported ( "GL_ARB_fragment_program
GL_ARB_vertex_program" );
    if ( !pbuffer.checkExtensions () )
    {
        printf ( "Pbuffer extensions not found\n" );
        return 1;
    }
    if ( !vertexProgram.load ( "sepia.vp" ) )
    {
        printf ( "Error loading vertex program:\n %s\n",
                  vertexProgram.getErrorString ().c_str () );
        return 1;
    }
    if ( !fragmentProgram.load ( "sepia.fp" ) )
    {
        printf ( "Error loading fragment program:\n %s\n",
                fragmentProgram.getErrorString ().c_str () );
        return 2;
    }
    decalMap  = createTexture2D ( true, "../Textures/oak.bmp" );
    stoneMap  = createTexture2D ( true, "../Textures/block.bmp" );
    teapotMap = createTexture2D ( true, "../Textures/Oxidated.jpg" );
    printf ( "Press F key to turn filtering on/off\n" );
    initPBuffer  ();
    glutMainLoop ();
    return 0;
}
```

При помощи клавиши <f> можно прямо во время выполнения программы включать и выключать эффект.

Еще одним простым эффектом, который легко реализуется при помощи фрагментной программы, является фильтр выделения границ. Он выделяет те места изображения, где происходит сильное изменение интенсивности.

Фильтрация выражается следующей формулой:

$$I = k \cdot \left(\left| t(x+h, y) - t(x-h, y) \right| + \left| t(x, y+h) - t(x, y-h) \right| \right).$$

Через $t(x, y)$ обозначена исходная текстура, а через h — некоторый шаг (в идеале равный одному текселу).

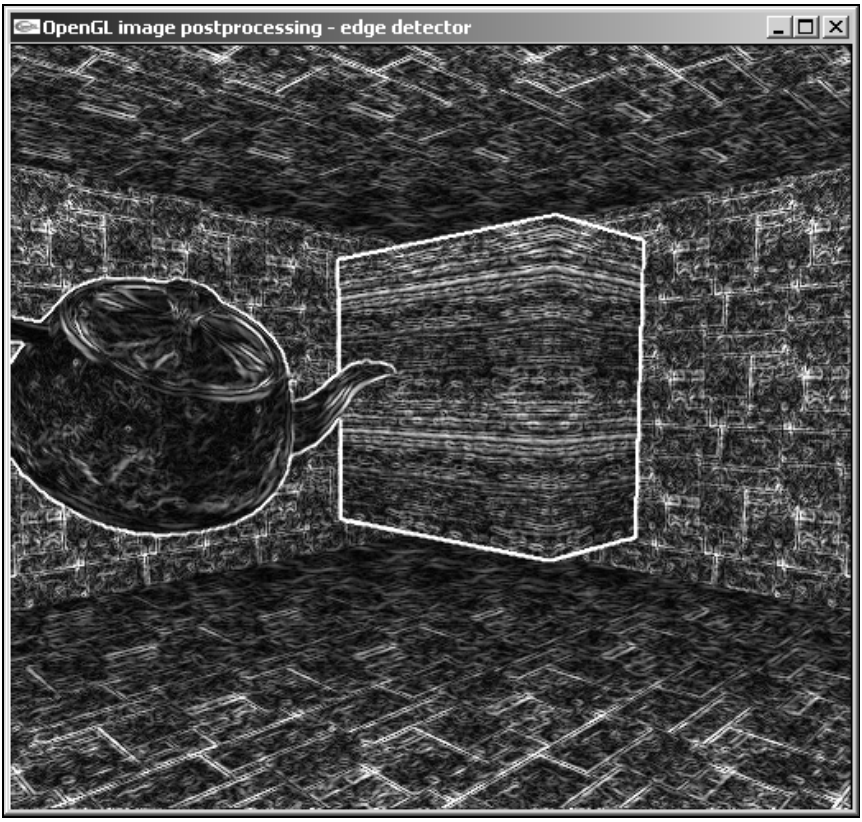


Рис. 16.6. Изображение с эффектом выделения границ

В листинге 16.18 приводится соответствующая фрагментная программа, а на рис. 16.6 — результат ее работы.

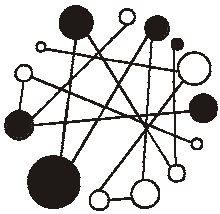
Листинг 16.18. Фрагментная программа, реализующая эффект выделения границы

```
!!ARBfp1.0
#
# simple edge detector fragment shader
# on entry:
#     fragment.texcoord [0] - texture coordinates
```

```
#
#     texture [0] - image to process
#
PARAM    luminance = { 0.3, 0.59, 0.11, 1 };
PARAM    color     = { 1, 0.89, 0.54, 1 };
PARAM    step      = 0.001953125;        # 1/512
PARAM    scale     = 1.2;                # scale factor to enhance edges
TEMP    c, lum;
TEMP    c0, c1, c2, c3;
TEMP    t0, t1, t2, t3;
TEMP    out;
                # compute neighbouring texcoords
MOV    t0, fragment.texcoord [0];
MOV    t1, fragment.texcoord [0];
MOV    t2, fragment.texcoord [0];
MOV    t3, fragment.texcoord [0];
ADD    t0.x, t0, -step;
ADD    t1.x, t1, step;
ADD    t2.y, t2, -step;
ADD    t3.y, t3, step;
                # now fetch values and get intensities
TEX    c0, t0, texture [0], 2D;
DP3    c0.a, c0, luminance;
TEX    c1, t1, texture [0], 2D;
DP3    c1.a, c1, luminance;
TEX    c2, t2, texture [0], 2D;
DP3    c2.a, c2, luminance;
TEX    c3, t3, texture [0], 2D;
DP3    c3.a, c3, luminance;
ADD    out.x, c1.a, -c0.a;
ADD    out.y, c3.a, -c2.a;
ABS    out.xy, out;
ADD    out.x, out.x, out.y;
MUL    out, out, scale;
MOV    out.w, 1;
MOV    result.color, out.x;
END
```

В *главе 18* будет рассмотрен более сложный эффект — эффект старого черно-белого фильма.

Глава 17



Язык GLSL для написания шейдеров

Рассмотренные в *главах 15 и 16* вершинные и фрагментные программы позволяют заменять собой части фиксированного конвейера рендеринга OpenGL, обеспечивая необходимую во многих случаях гибкость.

Однако сам язык, на котором пишутся эти программы, представляет собой всего лишь ассемблер, специально "заточенный" для выполнения поставленных перед вершинными и фрагментными программами задач, т. е. это язык довольно низкого уровня.

В системах профессионального фотореалистичного рендеринга уже давно стандартом языка для написания шейдеров является *RenderMan Shading Language*. Это высокоуровневый язык, сильно напоминающий C, с добавленными в него специальными типами данных и функциями.

Попытки реализации хотя бы некоторого подмножества *RenderMan Shading Language* для аппаратно-ускоренной графики ведутся уже давно. Одна из первых попыток такого рода заключается в трактовке команд OpenGL как некоторого ассемблера, средствами которого реализуются шейдеры. При этом специальный компилятор осуществляет перевод программы из некоторого подмножества *RenderMan Shading Language* в программы на языке C, вызывающие OpenGL. Для выполнения сложных вычислений применяется рендеринг в текстуру и много проходов. Так, даже простейшая реализация диффузного освещения (аналогичного расширению GL_EXT_texture_env_dot3) требует нескольких десятков проходов.

В силу этого данная работа имела довольно небольшую практическую применимость. Первой достаточно удобной с практической точки зрения реализацией шейдерного языка высокого уровня является язык Cg (*C for graphics*), созданный и поддерживаемый компанией NVIDIA. Разработчики программ на языке Cg имеют возможность прямо в своих OpenGL-программах вызывать шейдеры (написанные на Cg) для рендеринга объектов.

Реализация данного языка использует различные расширения OpenGL (при отсутствии поддержки фрагментных программ — используются расширения

NV_texture_shader и NV_register_combiners). Одним из преимуществ данного языка является то, что многие шейдеры, написанные на нем, вполне нормально выполняются на графических ускорителях класса GeForce 3/GeForce 4.

Однако по-настоящему удобным и мощным языком для написания шейдеров под OpenGL является так называемый *OpenGL Shading Language* (далее мы будем употреблять для его обозначения аббревиатуру GLSL). Этот язык играет важную роль в стандарте OpenGL 2.0, а пока доступен для остальных версий OpenGL через механизм расширений.

Таким образом, даже для OpenGL версии 1.1 (с которой приходится работать под Microsoft Windows) пользователь получает возможность применения в своих программах шейдеров, написанных на GLSL.

Однако поскольку GLSL ориентирован на работу с новейшими графическими ускорителями, то для его использования вам понадобятся достаточно мощный GPU и последняя версия драйвера для него.

Для программ на GLSL требуется поддержка следующих расширений:

- GL_ARB_shader_objects;
- GL_ARB_shading_language_100;
- GL_ARB_vertex_shader;
- GL_ARB_fragment_shader.

В *главе 18* мы подробно рассмотрим эти расширения, а пока изучим сам язык GLSL.

Язык GLSL

GLSL очень близок к языкам C, C++ и *RenderMan Shading Language*. При этом из него, по сравнению с C++, выброшены элементы, затрудняющие понимание и способствующие возникновению ошибок и неясностей.

На GLSL в настоящий момент возможно написание всего двух типов шейдеров (хотя, возможно, в дальнейшем появятся и новые) — вершинных и фрагментных.

Вершинные шейдеры

По аналогии в вершинной программой, вершинный шейдер выполняется для каждой вершины и полностью заменяет следующие функции OpenGL:

- преобразования при помощи модельной матрицы и матрицы проектирования;
- преобразование нормали и нормализацию;
- вычисление текстурных координат;

- преобразование текстурных координат;
- освещение;
- работу с цветными материалами.

Обратите внимание, что вершинный шейдер отвечает за реализацию всей этой функциональности, поэтому невозможно какую-либо часть ее переложить на стандартный конвейер.

Вершинный шейдер не располагает никакой топологической информацией, поэтому он не может выполнить функции, для которых она необходима:

- перспективное деление;
- преобразование в окно (*viewport mapping*);
- сборку примитивов;
- отсечение нелицевых граней;
- работу с двустороннними материалами;
- работу с диапазоном глубин.

Все параметры состояния OpenGL отслеживаются и являются доступными для шейдера.

Вершинный шейдер обрабатывает по одной вершине за раз. Его задачей является вычисление однородных координат преобразованной вершины, также он может вычислять освещение, текстурные координаты и другие параметры для передачи фрагментному шейдеру. Выполняется вершинный шейдер в момент задания координат очередной вершины. Одной из интересных его особенностей является возможность обращаться к текстурам.

Фрагментный шейдер

Этот шейдер заменяет собой ряд шагов в традиционном конвейере рендеринга, а именно:

- операции над интерполированными значениями;
- обращения к текстуре;
- наложение текстуры;
- наложение тумана;
- сложение цветов (*color sum*).

Фрагментный шейдер задает последовательность шагов общего вида для обработки каждого фрагмента. Фрагментный шейдер (как и вершинный) должен выполнять всю требуемую функциональность из приведенного выше списка, нельзя какую-либо ее часть передать стандартному конвейеру.

Фрагментный шейдер не может заменить собой следующие шаги стандартного конвейера:

- модель освещения;
- покрытие;
- тест принадлежности пиксела;
- тест на отсечение (*scissor test*);
- маски (*stipple*) для вывода линий и граней;
- проверку прозрачности;
- проверку трафарета;
- смешение цветов (*alpha blending*);
- логические операции;
- тонирование (*dithering*);
- цветовые маски.

Фрагментный шейдер не может изменять *xy*-положение фрагмента. Также фрагментной программе недоступны соседние фрагменты (это ограничение связано с возможностью распараллеливания фрагментных шейдеров).

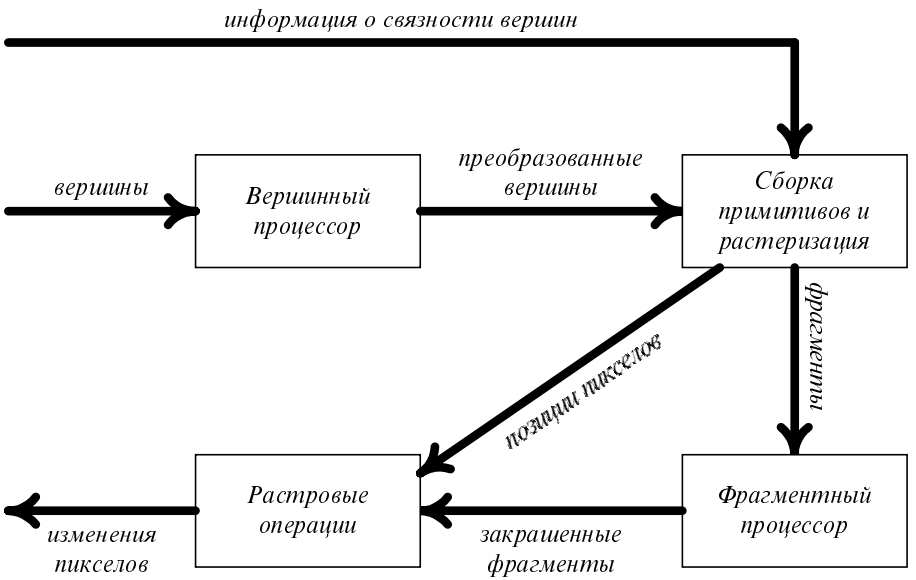


Рис. 17.1. Место шейдеров в конвейере рендеринга OpenGL

Для каждого фрагмента шейдер должен вычислить итоговое значение цвета и (или) глубины или полностью отбросить данный фрагмент.

На рис. 17.1 приведена упрощенная схема графического конвейера, показывающая место вершинного и фрагментного шейдеров.

Основные типы данных и переменных

Как и для языка C++, все переменные и функции сначала обязательно должны быть описаны. Какие-либо типы по умолчанию здесь отсутствуют, сам язык является *type-safe*, т. е. нет автоматических преобразований данных из одного типа в другой, все эти преобразования следует задавать явно в тексте самой программы.

В табл. 17.1 приводятся основные типы языка GLSL, однако пользователь может вводить структуры. Далее приведены примеры задания базовых типов и структур.

Таблица 17.1. *Основные типы языка GLSL*

Тип	Комментарий
`void`	Функции, не возвращающие значения, должны быть описаны как возвращающие `void`
`bool`	Булев тип, принимающий всего два значения `true` и `false`
`int`	Целочисленное число со знаком. Точность может зависеть от реализации, но гарантируется точность в 16 бит
`float`	Вещественное скалярное значение
`vec2`	Двумерный вектор из вещественных (`float`) чисел
`vec3`	То же трехмерный
`vec4`	То же четырехмерный
`bvec2`	Двумерный вектор из `bool`
`bvec3`	То же трехмерный
`bvec4`	То же четырехмерный
`ivec2`	Двумерный вектор из целых чисел
`ivec3`	То же трехмерный
`ivec4`	То же четырехмерный
`mat2`	Матрица 2×2 из вещественных чисел
`mat3`	То же 3×3

Таблица 17.1 *(окончание)*

Тип	Комментарий
`mat4`	То же 4×4
`sampler1D`	Ссылка на одномерную текстуру
`sampler2D`	То же на двумерную
`sampler3D`	То же на трехмерную
`samplerCube`	Ссылка на кубическую текстурную карту
`sampler1DShadow`	Ссылка на одномерную карту глубин
`sampler2DShadow`	Ссылка на двумерную 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;
```

Кроме стандартных, пользователь может строить собственные типы — структуры. Обратите внимание, что каждая структура должна быть именована.

Пример задания структур

```
struct Light
{
    vec3    pos;
    float   intensity;
    vec4    color;
};
Light    light1;
```

Структура должна содержать как минимум одно поле. Здесь 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.

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 большим или равным, чем первоначальный.

При обращении к массиву по отрицательному индексу или по индексу, выходящему за размер массива, поведение программы непредсказуемо. Для массива, являющегося формальным параметром функции, задание размера обязательно.

Индексация массива начинается с нуля, доступ к компонентам массива происходит при помощи оператора `[]`. Вот несколько примеров.

Примеры описания массивов

```
float        frequencies [4];
uniform vec3 directions  [4];
Light        light [3];
```

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 она 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 действия 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 самого типа переменной, определяющего тип значений, описания переменных могут включать в себя дополнительные описатели, идущие перед именем типа.

Возможные типы описателей приведены в табл. 17.2.

Таблица 17.2. *Возможные типы описателей переменных*

Тип	Комментарий
Отсутствует	Локальная *read/write* переменная или входной параметр для функции
`const`	Константа времени компиляции или *read/only* параметр функции

Таблица 17.2 *(окончание)*

Тип	Комментарий
`attribute`	Вершинный атрибут. Обеспечивает связь между вершинным шейдером и вершинными данными в OpenGL
`uniform`	Величина не меняет своего значения вдоль всего обрабатываемого примитива; образует связь между вершинным шейдером, OpenGL и приложением
`varying`	Обеспечивает связь между вершинным и фрагментным шейдером. Интерполируются (с учетом перспективы) вдоль всего примитива
`in`	Входной параметр для функции
`out`	Выходной параметр функции, не инициализированный при передаче внутрь функции
`inout`	Параметр функции, служащий как для передачи данных внутрь функции, так и для возвращения значения наружу

Обратите внимание, что глобальные переменные могут быть только `const`, `attribute`, `uniform` или `varying`. При описании переменной допустим только один из этих описателей. Локальные переменные допускают только описатель `const`. Для описания параметров функций пригодны только описатели `const`, `in`, `out` и `inout`.

Если при описании глобальной переменной не был задан ни один из этих описателей, то эта переменная не может участвовать в обмене данными с приложением, OpenGL и другими шейдерами.

Атрибуты (описатель attribute)

Описатель `attribute` служит для описания величин, которые передаются вершинному шейдеру из OpenGL, связанными с каждой вершиной. Переменные с этим описателем могут объявляться только в вершинном шейдере. Для вершинного шейдера атрибуты доступны только для чтения.

В качестве атрибутов могут выступать только переменные следующих типов: `float`, `vec2`, `vec3`, `vec4`, `mat2`, `mat3` и `mat4`.

В качестве атрибутов не могут выступать массивы и структуры. Вот пример задания атрибутов.

Пример задания атрибутов

```
attribute vec4     position;
attribute vec3     normal;
attribute vec2     texCoord;
```

Всем стандартным вершинным атрибутам OpenGL сопоставлены имена встроенных переменных (см. табл. 17.5 и 17.6).

Обычно место под вершинные атрибуты (за исключением матриц) выделяется как под четырехмерный вещественный вектор (`vec4`), за исключением матриц, представленных при помощи 2, 3 или 4 , соответственно, четырехмерных векторов.

Важно помнить, что практические реализации накладывают ограничения на число вершинных атрибутов. В *главе 18* будет приведена программа, печатающая информацию о поддержке GLSL, включая различные ограничения, накладываемые графическим ускорителем и его драйвером.

Все атрибуты должны быть глобальными, объявлены вне функций и 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` служит для описания глобальных переменных, значения которых не изменяются вдоль обрабатываемого примитива. Простейший пример — величины, задаваемые вне блока `glBegin/glEnd`.

Все такие переменные для шейдеров доступны только для чтения, запись в них может производиться либо явно самим приложением через соответствующий API, либо опосредованно самим OpenGL.

Описатель `uniform` подходит для переменных любых типов, массивов и структур. При сборке нескольких шейдеров вместе (например, вершинного и фрагментного) они обладают общим пространством имен `uniform`-переменных.

Реализация также может накладывать ограничения на число `uniform`-переменных. Далее приведен пример описания таких переменных.

Пример описания `uniform`-переменных

```
uniform vec4    lightPosition;
uniform vec4    lightColor;
```

Varying-переменные

Служат для связи между вершинным и фрагментным шейдерами. Вершинный шейдер, вычисляя значения для вершин, записывает их в соответствующие `varying`-переменные (а также может читать записанные значения).

При передаче фрагментному шейдеру эти переменные интерполируются вдоль всего примитива (с учетом перспективы), и фрагментный шейдер получает уже проинтерполированное значение для каждого фрагмента. Для фрагментного шейдера `varying`-переменные доступны только для чтения.

Поскольку эти переменные служат для связи между вершинным и фрагментным шейдером, то они должны быть объявлены в каждом из них и их типы должны совпадать. Вот пример их описания.

Пример 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 их массивам. Структуры не могут быть `varying`.

Как и `uniform`-переменные, `varying`-переменные также должны быть глобальными и объявлены до первого использования.

Операторы и выражения языка GLSL

В табл. 17.3 приведены все операторы языка GLSL с учетом их приоритетов и порядка выполнения.

Таблица 17.3. *Операторы языка GLSL*

Приоритет	Класс	Оператор	Ассоциативность
1 (наивысший)	Группировка при помощи скобок	`( )`	Не применима
2	Индексирование массивов	`[ ]`	Слева направо
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

Таблица 17.3 *(окончание)*

Приоритет	Класс	Оператор	Ассоциативность
3	Унарные операторы (оператор ~ зарезервирован)	+ - ~ !	Справа налево
4	Мультипликативные операторы (оператор % зарезервирован)	* / %	Слева направо
5	Аддитивные	+ -	Слева направо
6	Побитовые (зарезервированы)	<< >>	Слева направо
7	Сравнения	< > <= >=	Слева направо
8	Равенства	== !=	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 (%=, сдвиги и битовые операции зарезервированы)	+= -= *= /= %= <<= >>= &= |= ^=	Справа налево
17	Последовательность	,	Слева направо

Конструкторы

Конструкторы служат 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. Так, например,

конструкторы могут использоваться 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 и переданные параметры были соответствующих типов и в нужном количестве для получения требуемого значения.

Далее приводятся примеры преобразования скалярных типов друг в друга.

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` происходит отбрасывание дробной части. При преобразовании к типу `bool` значения `0` и `0.0` переводятся в `false`, а все остальные — в `true`.

При преобразовании значений типа `bool` в типы `int` и `float` значение `false` переводится в ноль (`0` или `0.0`), а `true` — в единицу (`1` или `1.0`).

Конструкторы скалярных типов могут принимать значения нескалярных типов, возвращая в этом случае первый элемент переданного векторного (или матричного) значения.

Конструкторы могут применяться для получения векторов, матриц из наборов скалярных значений векторов и матриц.

Если для инициализации вектора используется скалярное значение, то оно используется для инициализации всех его компонент:

```
vec3    v = vec3 ( 0.5 );       // will build (0.5, 0.5, 0.5)
```

Если единственное скалярное значение является параметром для конструктора матрицы, то оно используется для инициализации ее главной диагонали, 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 компонент, чем нужно для построения соответствующего значения, то лишние компоненты будут просто проигнорированы. Таким образом можно

строить более короткие векторы из более длинных. Передача дополнительных аргументов за последним необходимым является ошибкой.

Матрицы строятся по столбцам (первый столбец, затем второй и т. д.). Не допускается построение матриц из матриц. Вот примеры инициализации.

Примеры инициализации векторов и матриц

```
vec2    v2   = vec2 ( 1.0, 2.0 );  // will build (1.0, 2.0)
vec3    v3   = vec3 ( v2, 0.3 );    // will build (1.0, 2.0, 0.3)
vec4    v4   = vec4 ( v2, v2 );     // will build (1.0, 2.0, 1.0, 2.0)
vec4    v5   = vec4 ( 4.5, v3 );   // will build (4.5, 1.0, 2.0, 0.3)
vec4    rgba = vec4 ( 0.0 );       // will build (0.0, 0.0, 0.0, 0.0)
vec3    rgb  = vec3 ( rgba );      // will build (0.0, 0.0, 0.0)
mat3    m3   = mat3 ( 2.0 );
mat2    m2   = mat2 ( v2, v2 );    // will build matrix with both columns
                                   // equal (1.0, 2.0)
                                   // build 2D identity matrix
mat2    m    = mat2 ( 1.0, 0.0, 0.0, 1.0 );
```

Инициализация структур осуществляется конструктором, имя которого совпадает с именем самой структуры. В качестве параметров выбираются значения тех же типов и в том же порядке, в котором они были 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 компонент векторов используются комбинации символов (например, *xyz*). Поскольку с различными применениями векторов связаны разные имена компонент, то в GLSL предусмотрены три набора имен, приведенные в табл. 17.4.

Таблица 17.4. *Используемые в GLSL имена компонент векторов*

Обозначение	Использование
(*x*, *y*, *z*, *w*)	Удобно для доступа к векторам, обозначающим положение или направление

Таблица 17.4 *(окончание)*

Обозначение	Использование
(*r*, *g*, *b*, *a*)	Для доступа к векторам, представляющим собой значения цвета
(*s*, *t*, *p*, *q*)	Для доступа к векторам, являющимся текстурными координатами

Обратите внимание, что третья компонента в текстурных координатах (*r*) была переименована (в *p*), чтобы не возникло путаницы с *r*-компонентой цвета.

Доступ к полям вектора осуществляется через оператор . (точка), при этом после символа . может стоять произвольный набор имен компонент, что позволяет перемешивать компоненты и (или) дублировать их. Однако после точки должны идти имена полей из одной и той же группы. Таким образом, запись `v.xgp` недопустима, так как в ней есть имена из всех трех групп имен.

Указание набора имен полей после символа . предоставляет еще один способ (помимо конструкторов) для построения одних векторов из других.

Кроме того, конструкции с именами полей после символа . могут находиться и в левой части оператора присваивания, обеспечивая запись только в заданные поля вектора. Вот несколько примеров использования полей.

Примеры правильного использования имен полей векторов

```
vec4    v4 ( 1.0, 2.0, 3.0, 4.0 );
float   f 1= v4.x;        // all values f1, f2 and f3 will receive the
float   f2 = v4.r;        // same component of 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 полей векторов

```
v6.xx = 1.0;                        // два раза используется компонента x
v6.xy = vec3 ( 1.0, 1.0, 2.0 );     // несовпадение размеров
vec2    v7 = v5.xgbt;               // используются имена полей из
                                    // различных групп
```

Для доступа к компонентам векторов можно также использовать индексацию. Вектор трактуется как массив из соответствующего числа элементов (2,

3 или 4) и, 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 применяется индексирование, при этом задание только первой компоненты 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. нумерация столбцов происходит слева направо. При двух индексах первый из них выбирает столбец матрицы, а второй — элемент из столбца. Далее приведены примеры работы с матрицами.

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 полей вектора, для обращения к полям структуры используется оператор . (точка).

Вообще к структурам применимы только три оператора: обращение к полю (.), сравнение (== и !=) и присваивание (=).

Сравнение структур трактуется как покомпонентное сравнение всех полей.

Основные операции над векторами и матрицами

Большинство операций над векторами и матрицами (за небольшим числом исключений) являются покомпонентными, т. е. они независимо выполняются для каждой пары соответствующих компонент (рис. 17.2 и 17.3).

Исключением из правила являются операции умножения матрицы на вектор, вектора на матрицу и матрицы на матрицу (рис. 17.4 и 17.5). Для их выполнения необходимы определенные условия, накладываемые на размеры операндов (*см. Приложение 1*).

Функция `dot` служит для вычисления скалярного произведения векторов.

В случае умножения вектора на матрицу вектор трактуется как строка, умножаемая слева на матрицу (рис. 17.6). Сами же эти операции полностью эквивалентны операциям, вводимым в курсе линейной алгебры.

Все унарные операции выполняются только покомпонентно.

```
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. 17.2. Примеры покомпонентных операций

```
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. 17.3. Еще один пример покомпонентной операции

```
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. 17.4. Пример умножения матрицы на вектор

```
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. 17.5. Умножение матрицы на матрицу

```
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. 17.6. Умножение вектора на матрицу

Стандартные переменные

И для вершинного, и для фрагментного шейдеров определены некоторые стандартные переменные, 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, представленные следующими объявлениями:

```
vec4    gl_Position;
float   gl_PointSize;
vec4    gl_ClipVertex;
```

Все они являются глобальными, т. е. доступны в любом месте вершинного шейдера и не требуют объявления.

Три эти переменные (табл. 17.5) доступны только в вершинном шейдере и имеют следующее предназначение:

- в `gl_Position` должны быть записаны однородные координаты вершины. Эта переменная доступна также и для чтения;
- в `gl_PointSize` шейдер может записать размер точки для растеризации (в пикселах). Если запись в эту переменную не производилась, то чтение из нее дает неопределенные значения;
- в `gl_ClipVertex` могут быть записаны координаты для использования с задаваемыми пользователем плоскостями отсечения. Убедитесь, что переданные координаты и плоскости отсечения заданы в одном и том же координатном пространстве.

Таблица 17.5. *Специальные переменные для вершинного шейдера*

Переменная	Тип	Запись	Чтение
`gl_Position`	`vec4`	Обязательно	Возможно, но если до этого запись не производилась, то значение не определено
`gl_PointSize`	`float`	Не обязательна	То же
`gl_ClipVertex`	`vec4`	То же	То же

Специальные переменные для фрагментных шейдеров

Далее приводятся описания специальных переменных для фрагментного шейдера.

```
vec4    gl_FragCoord;
bool    gl_FrontFacing;
vec4    gl_FragColor;
vec4    gl_FragData [gl_MaxDrawBuffers];
float   gl_FragDepth;
```

Фрагментный шейдер должен либо отбросить фрагмент (командой `discard`), либо записать его параметры 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 `gl_FragDepth` — выходное значение глубины для фрагмента. При этом если шейдер не производит записи в эту переменную, то в качестве глубины выбирается 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. 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-компонента этой переменной содержит значение глубины, которое будет использовано, если шейдер не производит запись в `gl_FragDepth`.

Фрагментному шейдеру также доступна (только для чтения) булева переменная `gl_FrontFacing`, принимающая значение `true`, если фрагмент принадлежит лицевому примитиву. Свойства всех специальных переменных для фрагментного шейдера сведены в табл. 17.6.

Таблица 17.6. *Специальные переменные для фрагментного шейдера*

Переменная	Тип	Запись	Чтение
`gl_FragCoord`	`vec4`	Нет	Да
`gl_FrontFacing`	`bool`	Нет	Да
`gl_FragColor`	`vec4`	Обязательна, если фрагмент не отбрасывается	Возможно, но если до этого запись не производилась, то значение не определено
`gl_FragData`	`vec4 []`	Обязательна, если фрагмент не отбрасывается и не производится запись в `gl_FragColor`	То же
`Gl_FragDepth`	`float`	Не обязательна	То же

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). Вот их примеры.

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. Далее приведены их описания.

Стандартные атрибуты для вершинного шейдера

```
attribute vec4 gl_Color;
attribute vec4 gl_SecondaryColor;
attribute vec3 gl_Normal;
attribute vec4 gl_Vertex;
attribute vec4 gl_MultiTexCoord0;
attribute vec4 gl_MultiTexCoord1;
attribute vec4 gl_MultiTexCoord2;
attribute vec4 gl_MultiTexCoord3;
attribute vec4 gl_MultiTexCoord4;
attribute vec4 gl_MultiTexCoord5;
attribute vec4 gl_MultiTexCoord6;
attribute vec4 gl_MultiTexCoord7;
attribute float gl_FogCoord;
```

Атрибуты `gl_Color` и `gl_SecondaryColor` содержат в себе основной и дополнительный (вторичный) цвета в текущей вершине, `gl_Normal` — вектор нормали в текущей вершине, `gl_Vertex` — координаты самой текущей вершины, `gl_MultiTexCoordN` — текстурные координаты для блока `N`, `gl_FogCoord` — величину затуманивания текущей вершины.

Стандартные *uniform*-переменные состояния

Также для шейдеров на GLSL доступен набор `uniform`-переменных, содержащих в себе параметры состояния OpenGL. Основные из них приведены далее.

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];
```

Все эти переменные содержат в себе как стандартные матрицы OpenGL, так и их транспонированные и (или) обращенные варианты.

Параметры, задающие преобразование глубины в отрезок [0, 1], представлены в виде структуры.

Параметры преобразования глубины

```
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 четырехмерных векторов, где каждой плоскости соответствует вектор из коэффициентов ее уравнения.

Параметры плоскостей отсечения

```
uniform vec4 gl_ClipPlane[gl_MaxClipPlanes];
```

Параметры, задающие свойства точки (см. расширение ARB_point_parameters), также доступны в виде структуры.

Свойства точки

```
struct gl_PointParameters
{
       float size;
       float sizeMin;
       float sizeMax;
       float fadeThresholdSize;
       float distanceConstantAttenuation;
       float distanceLinearAttenuation;
       float distanceQuadraticAttenuation;
};
uniform gl_PointParameters gl_Point;
```

Шейдеры также имеют доступ к свойствам материала, представленными в виде структуры.

Переменные, содержащие свойства материала

```
struct gl_MaterialParameters
{
       vec4 emission;
       vec4 ambient;
       vec4 diffuse;
       vec4 specular;
       float shininess;
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

Аналогичным образом доступна информация об источниках света.

Переменные, содержащие свойства 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 о модели освещения и вспомогательных величинах, удобных для расчета освещенности.

Параметры модели освещения

```
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 одним типом доступной информации являются параметры автоматического вычисления текстурных координат (для каждого текстурного блока).

Параметры автоматического вычисления текстурных координат и цвета для текстурных блоков

```
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];
```

При помощи еще одной структуры представлены параметры для вычисления тумана.

Параметры для вычисления тумана

```
struct gl_FogParameters
{
       vec4  color;
       float density;
       float start;
       float end;
       float scale;         // Derived: 1.0 / (end - start)
};
uniform gl_FogParameters gl_Fog;
```

Varying-переменные

Кроме `uniform`-переменных существует также стандартный набор `varying`-переменных, перечисленных далее.

Стандартные `uniform`-переменные

```
varying vec4  gl_FrontColor;
varying vec4  gl_BackColor;
```

```
varying vec4  gl_FrontSecondaryColor;
varying vec4  gl_BackSecondaryColor;
varying vec4  gl_TexCoord [];     // at most will be gl_MaxTextureCoords
varying float gl_FogFragCoord;
varying vec4  gl_Color;
varying vec4  gl_SecondaryColor;
varying vec4  gl_TexCoord [];     // at most will be gl_MaxTextureCoords
varying float gl_FogFragCoord;
```

Стандартные функции

Программам на GLSL (как вершинным, так и фрагментным) доступен целый ряд встроенных функций.

Их можно разделить на следующие классы:

- функции, реализующие непосредственную работу с графическим ускорителем (например, доступ к текстуре);
- функции, выполняющие обычные простые операции (например, `clamp`, `mix` и т. п.), которые хотя и очень просты, но очень часто встречаются и могут иметь аппаратную поддержку;
- функции, которые могут, начиная с какого-то момента времени, получить аппаратную поддержку.

Многие из них похожи на свои аналоги в C, однако в качестве аргументов принимают не только скалярные, но и векторные значения.

В табл. 17.7 в качестве типа *T* может быть любой из следующих типов: `float`, `vec2`, `vec3` и `vec4`. Типом *M* может выступать любой из матричных: `mat2`, `mat3` и `mat4`.

Тригонометрические функции и функции для работы с углами

Все тригонометрические функции (табл. 17.7) работают с углами, выраженными в радианах.

Таблица 17.7. *Тригонометрические функции и функции для работы с углами*

Функция	Описание
`T radians ( T deg )`	Перевод угла из градусов в радианы, возвращает $\frac{\pi \cdot deg}{180}$

Таблица 17.7 *(окончание)*

Функция	Описание
`T degrees ( T radians )`	Перевод угла из радиан в градусы, возвращает $\frac{180 \cdot radians}{\pi}$
`T sin ( T angle )`	Вычисление синуса угла (заданного в радианах)
`T cos ( T angle )`	Вычисление косинуса угла
`T tan ( T angle )`	Вычисление тангенса угла
`T asin ( T x )`	Вычисление арксинуса величины, результат принадлежит отрезку $[-\pi/2, \pi/2]$. Результат не определен для $\|x\|>1$
`T acos ( T x )`	Вычисление арккосинуса величины, результат принадлежит отрезку $[0, \pi]$. Результат не определен для $\|x\|>1$
`T atan ( T y, T x )`	Вычисление арктангенса y/x, знаки аргументов определяют, в каком квадранте лежит угол. Результат принадлежит отрезку $[-\pi, \pi]$
`T atan ( T y_over_x )`	Вычисляет арктангенс величины *y_over_x*. Результат принадлежит отрезку $[-\pi/2, \pi/2]$

Экспоненциальные функции (возведение в степень, нахождение логарифмов)

Все эти функции (табл. 17.8) выполняются покомпонентно.

Таблица 17.8. *Экспоненциальные функции*

Функция	Описание
`T pow ( T x, T y )`	Вычисляет x^y. Результат не определен, если $x < 0$ или $x = 0$ и $y \le 0$
`T exp ( T x )`	Вычисляет e^x
`T log ( T x )`	Вычисляет $\ln x$ Результат не определен для $x \le 0$

Таблица 17.8 *(окончание)*

Функция	Описание
`T exp2 ( T x )`	Вычисляет 2^x
`T log2 ( T x )`	Вычисляет $\log_2 x$ Результат не определен для $x \le 0$
`T sqrt ( T x )`	Вычисляет $\sqrt{x}$. Результат не определен для $x < 0$
`T inversesqrt ( T x )`	Вычисляет $1/\sqrt{x}$. Результат не определен для $x \le 0$

Функции общего назначения

Все эти функции (табл. 17.9) являются покомпонентными.

Таблица 17.9. *Функции общего назначения*

Функция	Описание
`T abs ( T x )`	Вычисляет $\lvert x \rvert$
`T sign ( T x )`	Возвращает –1, если $x < 0$; 0, если $x = 0$; 1, если $x > 0$
`T floor ( T x )`	Возвращает наибольшее целое, не превосходящее x
`T ceil ( T x )`	Возвращает наименьшее целое, не меньшее, чем x
`T fract ( Type x)`	Возвращает дробную часть x. Результат равен $x - floor(x)$
`T mod ( T x, float y )`	Возвращает $x - y \cdot floor(x/y)$
`T mod ( T x, T y )`	Возвращает $x - y \cdot floor(x/y)$
`T min ( T x, T y )` `T min ( T x, float y )`	Возвращает минимум из двух величин
`T max ( T x, T y )` `T max ( T x, float y )`	Возвращает максимум из двух величин

Таблица 17.9 *(окончание)*

Функция	Описание
`T clamp ( T x, T minVal, T maxVal)` `T clamp (T x, float minVal, float maxVal)`	Возвращает min(max(*x*, min*Value*), max*Value*). Фактически первый аргумент отсекается по отрезку [min*Value*, max*Value*]
`T mix ( T x, T y, Te a )` `T mix ( T x, T y, float a )`	Возвращает $x \cdot (1 - a) + y \cdot a$
`T step ( T edge, T x )` `T step ( float edge, T x )`	Возвращает 0,0, если $x < edge$ Возвращает 1,0 в противном случае

`T smoothstep ( T edge0, T edge1, T x )`
`T smoothstep ( float edge0, float edge1, T x )`
	Возвращает 0,0, если $x \le edge0$.
	Возвращает 1,0, если $x \ge edge1$.
	На интервале (*edge*0, *edge*1) используется интерполяция Эрмита.
	Функция описывается следующим фрагментом кода:
	`T t = clamp ((x-edge0)/(edge1-edge0),0.0,1.0);`
	`return t*t*(3-2*t);`

Последняя функция `smoothstep` использует интерполяцию Эрмита для гладкого перехода от значения 0,0 к 1,0. На рис. 17.7 приведен ее график.

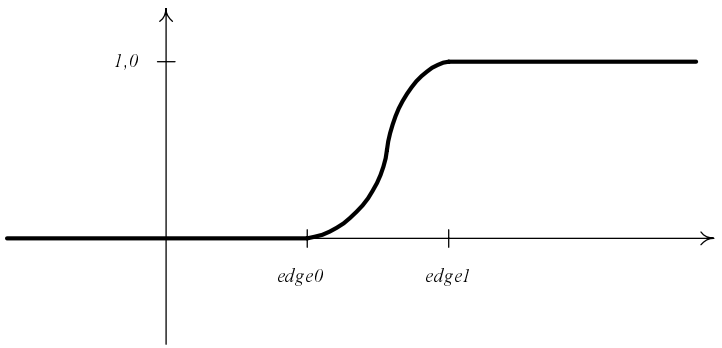


Рис. 17.7. График функции `smoothstep`

Геометрические функции

Все геометрические функции (табл. 17.10) работает непосредственно с векторами, т. е. не являются покомпонентными.

Таблица 17.10. *Геометрические функции*

Функция	Описание
`float length ( T x)`	Возвращает длину вектора *x*
`float distance ( T p0, T p1 )`	Возвращает расстояние между точками *p*0 и *p*1, т. е. *length* (*p*0 – *p*1)
`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 *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 *N*, если *dot* (*I*, *Nref*) < 0. Иначе возвращается –*N*
`T reflect ( T I, T N )`	Возвращает отражение вектора *I* для нормали в точке *N*, т. е. *I*–2 · *dot*(*I*, *N*) · *N*. Вектор *N* должен быть единичным

`T refract ( T I, T N, float eta)`	Возвращает преломленный вектор *I*, где *N* – это вектор нормали, а *eta* — относительный коэффициент преломления.
	Действие функции описывается следующим фрагментом кода:
	`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`

Функция `ftransform` обычно применяется в том случае, когда необходим рендеринг в несколько проходов, причем часть из них осуществляется через

стандартный конвейер. Использование этой функции позволяет вершинному шейдеру возвращать то же самое значение для `gl_Position`, что и в остальных проходах. В известном смысле является аналогом опции ARB_position_invariant для вершинных программ (*см. главу 15*).

Матричные функции

Единственная матричная функция представлена в табл. 17.11.

Таблица 17.11. *Матричные функции*

Функция	Описание
`M matrixCompMult ( M x, M y )`	Возвращает поэлементное произведение двух матриц

Функции для сравнения векторов

Данные функции служат для поэлементного сравнения векторов, поскольку стандартные операторы сравнения (<, >, <=, >=, ==, !=) работают только со скалярными аргументами.

В табл. 17.12, содержащей функции сравнения векторов, через `vec` обозначен любой вещественнозначный вектор (`vec2`, `vec3` и `vec4`), через `ivec` обозначен любой вектор с целочисленными компонентами (`ivec2`, `ivec3` и `ivec4`).

Через `bvec` обозначен любой вектор из логических значений, т. е. вектор одного из следующих типов: `bvec2`, `bvec3` или `bvec4`.

Таблица 17.12. *Функции для сравнения векторов*

Функция	Описание
`bvec lessThan ( vec x, vec y )` `bvec lessThan ( ivec x, ivec y )`	Возвращает результат поэлементного сравнения операндов $x < y$
`bvec lessThanEqual ( vec x, vec y )` `bvec lessThanEqual (ivec x, ivec y )`	Выполняет поэлементное сравнение $x \le y$
`bvec greaterThan ( vec x, vec y )` `bvec greaterThan ( ivec x, ivec y )`	Выполняет поэлементное сравнение $x > y$
`bvec greaterThanEqual ( vec x, vec y )` `bvec greaterThanEqual ( ivec x, ivec y )`	Выполняет поэлементное сравнение $x \ge y$

Таблица 17.12 *(окончание)*

Функция	Описание

`bvec equal ( vec x, vec y )`
`bvec equal ( ivec x, ivec y )`
`bvec equal ( bvec x, bvec y )`
	Выполняет поэлементное сравнение *x* == *y*

`bvec notEqual ( vec x, vec y )`
`bvec notEqual ( ivec x, ivec y )`
`bvec notEqual ( bvec x, bvec y )`
	Выполняет поэлементное сравнение *x* != *y*

`bool any ( bvec x )`	Возвращает `true`, если хотя бы одна компонента *x* принимает значение `true`
`bool all ( bvec x )`	Возвращает `true`, если все компоненты *x* принимают значение `true`
`bvec not ( bvec x )`	Возвращает покомпонентное логическое отрицание *x*

Функции для доступа к текстурам

Все эти функции (табл. 17.13) доступны как фрагментным, так и вершинным шейдерам. 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 величины `LOD` он добавляется к вычисленному значению перед обращением к текстуре. Если его нет, то обращение к текстуре производится стандартным образом.

Все функции, имена которых заканчиваются на `Lod`, предназначены только для вершинных шейдеров, при этом параметр `lod` задает уровень детализации.

Если в имени функции присутствует суффикс `Proj`, то перед обращением текстурные координаты делятся на свою последнюю компоненту.

Таблица 17.13. *Функции для доступа к текстурам*

Функция	Описание

`vec4 texture1D (sampler1D sampler,`
`float coord [, float bias] )`
`vec4 texture1DProj (sampler1D sampler,`
`vec2 coord [, float bias] )`
`vec4 texture1DProj (sampler1D sampler,`
`vec4 coord [, float bias] )`
`vec4 texture1DLod (sampler1D sampler,`
`float coord, float lod)`
`vec4 texture1DProjLod (sampler1D sampler,`
`vec2 coord, float lod)`
`vec4 texture1DProjLod (sampler1D sampler,`
`vec4 coord, float lod)`
	Использует параметр `coord` для доступа к одномерной текстуре, задаваемой параметром `sampler`.
	В Proj-версии для индексации текстуры используется величина `coord.s/coord.t`

`vec4 texture2D (sampler2D sampler,`
`vec2 coord [, float bias] )`
`vec4 texture2DProj (sampler2D sampler,`
`vec3 coord [, float bias] )`
`vec4 texture2DProj (sampler2D sampler,`
`vec4 coord [, float bias] )`
`vec4 texture2DLod (sampler2D sampler,`
`vec2 coord, float lod)`
`vec4 texture2DProjLod (sampler2D sampler,`
`vec3 coord, float lod)`
`vec4 texture2DProjLod (sampler2D sampler,`
`vec4 coord, float lod)`
	Использует параметр `coord` для доступа к двухмерной текстуре, задаваемой параметром `sampler`.
	В Proj-версии для индексации текстуры используется величина `(coord.s/coord.p, coord.t/coord.p)`

Таблица 17.13 *(продолжение)*

Функция	Описание

`vec4 texture3D (sampler3D sampler,`
`vec3 coord [, float bias] )`
`vec4 texture3DProj (sampler3D sampler,`
`vec4 coord [, float bias] )`
`vec4 texture3DLod (sampler3D sampler,`
`vec3 coord, float lod)`
`vec4 texture3DProjLod (sampler3D sampler,`
`vec4 coord, float lod)`
	Использует параметр `coord` для доступа к трехмерной текстуре, задаваемой параметром `sampler`.
	В Proj-версии для индексации текстуры используется величина `(coord.s/coord.q, coord.t/coord.q, coord.p/coord.q)`

`vec4 textureCube (samplerCube sampler,`
`vec3 coord [, float bias] )`
`vec4 textureCubeLod (samplerCube sampler,`
`vec3 coord, float lod)`
	Использует параметр `coord` для доступа к кубической текстурной карте, задаваемой параметром `sampler`

`vec4 shadow1D (sampler1DShadow sampler,`
`vec3 coord [, float bias] )`
`vec4 shadow2D (sampler2DShadow sampler,`
`vec3 coord [, float bias] )`
`vec4 shadow1DProj (sampler1Dshadow sampler,`
`vec4 coord [, float bias] )`
`vec4 shadow2DProj (sampler2Dshadow sampler,`
`vec4 coord [, float bias] )`
`vec4 shadow1DLod (sampler1Dshadow sampler,`
`vec3 coord, float lod)`
`vec4 shadow2DLod (sampler2Dshadow sampler,`
`vec3 coord, float lod)`
	Использует параметр `coord` для доступа к одномерной или двумерной текстуре, содержащей значения глубины, задаваемой параметром `sampler`.
	В Proj-версии перед индексацией текстуры параметр `coord` делится на свою последнюю компоненту

Таблица 17.13 *(окончание)*

Функция	Описание
`vec4 shadow1DProjLod(sampler1Dshadow sampler, vec4 coord, float lod)` `vec4 shadow2DProjLod(sampler2Dshadow sampler, vec4 coord, float lod)`	

Функции для работы с производными

Эти функции (табл. 17.14) позволяют приближенно вычислять значения производных указанных величин. Однако реализация может давать не очень точные значения.

Справедливо следующее приближенное равенство:

$$F(x+h) - F(x) \approx dFdx(x) \cdot h.$$

Таблица 17.14. *Функции для работы с производными*

Функция	Описание
`T dFdx ( T p )`	Возвращают приближенное значение первой производной параметра `p` по `x`
`T dFdy ( T p )`	Возвращают приближенное значение первой производной параметра `p` по `y`
`T fwidth ( T p )`	Возвращает `abs(dFdx(p))+ abs(dFdy(p))`

Шумовые функции

Доступны как вершинным, так и фрагментным шейдерам. Все эти функции (табл. 17.15) обладают следующими свойствами:

- значения принадлежат отрезку [−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.

Таблица 17.15. *Шумовые функции*

Функция	Описание
`float noise1 ( T x )`	Возвращает значение одномерной шумовой функции, соответствующей параметру *x*
`vec2 noise2 ( T x )`	То же для двумерной
`vec3 noise3 ( T x )`	То же для трехмерной
`vec4 noise4 ( T x )`	То же для четырехмерной

Основные операторы и конструкции GLSL

Имена переменных и функций в GLSL строятся из прописных и строчных латинских букв, цифр и символа подчеркивания (_). При этом они обязаны начинаться не с цифры.

Имена, начинающиеся с `gl_`, зарезервированы и не могут быть использованы для обозначения своих переменных и функций.

Комментарии в GLSL обозначаются тем же способом, что и в языке C++. Как текст между символами `/*` и `*/`, либо от `//` и до конца строки.

Программы на GLSL сначала обрабатываются препроцессором, поэтому в языке для этого есть ряд команд (табл. 17.16).

Таблица 17.16. *Команды для препроцессора GLSL*

Команда	Значение
`#`	Игнорируется
`#define`	Аналогично языку C (как с параметрами, так и без)
`#undef`	То же
`#if`	Аналогично языку C
`#ifdef`	То же
`#ifndef`	То же
`#else`	То же

Таблица 17.16 *(окончание)*

Команда	Значение
`#elif`	То же
`#endif`	То же
`#error`	Помещает сообщение, идущее за этой командой и до конца строки в лог. Компиляция считается завершившейся неудачно
`#pragma`	Позволяет работать с особенностями отдельных компиляторов
`#extension`	Позволяет управлять различными расширениями языка
`#version`	Заявляет, для какой версии языка написана данная программа
`#line`	Позволяет задать новую нумерацию для строк

Также доступен оператор `defined`, позволяющий узнать, определен ли данный символ.

В табл. 17.17 перечислены предопределенные макросы GLSL.

Таблица 17.17. *Предопределенные макросы*

Макрос	Значение
`__LINE__`	Число символов от начала строки в текущей строке
`__FILE_`	Номер текущей строки
`__VERSION__`	Версия GLSL

Все макросы, содержащие два подчеркивания подряд (`__`), зарезервированы для дальнейшего использования.

Командой `#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 для управления выполнением программы:

```
if ( bool expression )
      …
else
     …
```

```
for ( initialization; bool expression; loop expression )
     …
while ( bool expression )
    ……
do
   ……
while ( bool expression )
```

Доступны также команды для осуществления переходов в программе: `continue`, `break` и `discard`. Действие первых двух из них полностью аналогично языку C. Команда `discard`, доступная только в фрагментном шейдере, немедленно прекращает обработку текущего фрагмента (аналогично команде `KIL` для фрагментных программ).

Программы на языке GLSL могут содержать функции, причем для каждой из них должен быть явно задан тип возвращаемого значения:

```
returnType functionName ( type0 arg0, type1 arg1, ..., typen argn )
{
       // do some computation
}
```

Можно просто объявлять функции, задавая их реализацию позже:

```
returnType functionName ( type0 arg0, type1 arg1, ..., typen argn );
```

Далее приводится пример задания функции.

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)
{
   …..
}
```

Простейший пример использования вершинных и фрагментных шейдеров

В листингах 17.1 и 17.2 приведены минимальные вершинный и фрагментный шейдеры на GLSL.

Минимальная функциональность, которую обязан предоставлять вершинный шейдер, — запись координат преобразованной вершины в переменную `gl_Position`.

Листинг 17.1. Простейший вершинный шейдер на GLSL

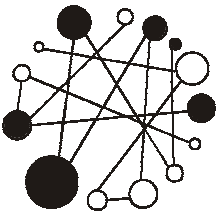
```
//
// Simplest GLSL vertex shader
//
void main (void)
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Минимальная функциональность, которой должен обладать фрагментный шейдер, — запись цвета в переменную `gl_FragColor`.

Листинг 17.2. Простейший фрагментный шейдер на GLSL

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

Глава 18



Практика программирования на GLSL

Рассмотрим сначала некоторые расширения, необходимые для работы с GLSL, а также вводимые ими функции.

Расширения для работы с GLSL-шейдерами и вводимые ими функции

Для работы с GLSL необходима поддержка следующих расширений: GL_ARB_shader_objects, GL_ARB_shading_language_100, GL_ARB_vertex_shader и GL_ARB_fragment_shader.

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.

Определение поддерживаемой версии GLSL

```
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 с номером 1.051 (по номеру версии описания).

Расширение GL_ARB_shader_objects

Вводит необходимый API для поддержки шейдеров. При этом разделяется понятие шейдера (вершинного или фрагментного) и программы, включающей в себя эти шейдеры.

Непосредственная поддержка вершинных и фрагментных шейдеров вводится расширениями GL_ARB_vertex_shader и GL_ARB_fragment_shader, рассматриваемыми 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`.

Этот тип идентифицирует внутренний объект OpenGL. Соответствующие объекты могут содержать в себе какие-то данные, создаваться, уничтожаться и изменяться. Нулевому значению не соответствует никакой объект, поэтому если возвращенный идентификатор равен нулю, то это признак ошибки создания этого объекта.

Некоторые из объектов могут выступать как контейнеры, т. е. содержать в себе другие объекты. Типичным примером такого объекта-контейнера служит объект программы.

Для создания и уничтожения соответствующих объектов служат следующие функции:

```
GLhandleARB glCreateProgramObjectARB ();
GLhandleARB glCreateShaderObjectARB  ( GLenum shaderType  );
void        glDeleteObjectARB        ( GLhandleARB object );
```

`glCreateProgramObjectARB` создает объект программы и возвращает его идентификатор.

`glCreateShaderObjectARB` служит для создания объектов-шейдеров. Параметр `shaderType` задает требуемый тип шейдера и может пока принимать одно из следующих двух значений: `GL_VERTEX_SHADER_ARB` и `GL_FRAGMENT_SHADER_ARB`.

Функция `glDeleteObjectARB` служит 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 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 );
```

`glAttachObjectARB` помещает (подсоединяет *attaches*) объект `obj` в объект-контейнер `containerObj`. В случае если `containerObj` не является объектом-контейнером или подсоединение невозможно, возникает ошибка. Ошибкой также является попытка повторного помещения объекта (т. е. когда он уже присутствует в контейнере).

`glDetachObjectARB` убирает 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 служит функция

```
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, `strings` является указателем на массив указателей на строки.

Параметр `lengths` является указателем на массив, задающий для каждой строки ее длину. В случае, когда `lengths` равен `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 дальше).

После того как исходный код для шейдера загружен, его необходимо (в отличие от ранее рассмотренных вершинных и фрагментных программ) откомпилировать. Для этого служит функция

```
void glCompileShaderARB ( GLhandleARB shaderObj );
```

Ее вызов приводит к компиляции ранее загруженного исходного текста шейдера на GLSL. Узнать результат компиляции программы можно при помощи следующего фрагмента кода:

```
GLint    compileStatus;
glGetObjectParameterivARB ( shader, GL_OBJECT_COMPILE_STATUS_ARB,
                            &compileStatus );
```

В случае если программа была успешно скомпилирована, значение переменной `compileStatus` будет равно `GL_TRUE`.

После того как все шейдеры объекта-программы были успешно откомпилированы, необходим еще один шаг, после которого программа будет готова к использованию, — линковка (связывание шейдеров и окружения между собой). Для этого служит функция

```
void glLinkProgramARB ( GLhandleARB programObj );
```

Здесь programObj идентифицирует объект-программу. Проверить результаты линковки можно, выполнив следующий фрагмент кода:

```
GLint       linked;
glGetObjectParameterivARB ( program, GL_OBJECT_LINK_STATUS_ARB,
                            &linked );
```

Если после выполнения этого кода значение переменной linked равно GL_FALSE, то это значит, что линковка завершилась неудачно.

В случае успешной компиляции следующий вызов делает заданную программу активной.

```
void glUseProgramObjectARB ( GLhandleARB programObj );
```

Если значение параметра programObj равно нулю, то этот запрос отключает шейдеры и устанавливает стандартный конвейер рендеринга OpenGL.

Даже когда программа является текущей, по-прежнему можно загружать и компилировать исходные тексты шейдеров, подсоединять и отсоединять отдельные шейдеры. Любые такие изменения вступают в силу только после того, как программа будет снова слинкована.

Иногда оказывается, 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 возможности успешного выполнения программы.

Получить идентификатор активной в данный момент программы можно при помощи функции

```
GLhandleARB glGetHandleARB ( GLenum pname );
```

Значением параметра pname должна быть константа GL_PROGRAM_OBJECT_ARB.

С каждым из объектов-шейдеров и объектов-программ связан свой внутренний лог — текстовый буфер, куда заносится информация о компиляции и линковке. Для его получения необходимо сначала узнать размер, выделить буфер необходимого объема, после чего скопировать содержимое лога в этот буфер. Далее приведен пример чтения лога объекта.

Чтение лога объекта

```
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 следующим образом:

```
void glGetInfoLogARB ( GLhandleARB obj, GLsizei maxLength,
                       GLsizei * length, GLcharARB * infoLog );
```

Параметр `maxLength` задает максимальное количество байтов, которое может быть записано в буфер по адресу, содержащемуся в параметре `infoLog`. Параметр `length` содержит адрес переменной, в которую будет записано количество скопированных в буфер байтов.

Данное расширение также предоставляет ряд функций, служащих для работы с `uniform`-переменными. Несмотря на то, что внутри шейдеров все `uniform`-переменные имеют имена, по которым с ними осуществляется работа, функции для работы с `uniform`-переменными вместо имени переменной используют ее целочисленный индекс. Для получения индекса (*location*) по имени (*name*) переменной для данной программы служит функция `glGetUniformLocationARB`.

```
int glGetUniformLocationARB ( GLhandleARB programObj,
                              const GLcharARB * name );
```

Она возвращает индекс соответствующей `uniform`-переменной или -1 в случае ошибки.

С ее помощью можно также получить индексы отдельных элементов массивов (добавляя к имени массива индекс в квадратных скобках, например, `myArray[1]`) и полей структуры (добавляя к имени после оператора . имя поля, например, `light1.intensity`).

После получения индекса (*location*) с помощью следующих функций можно 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 из вещественных и целых чисел. В них параметр `count` задает число значений, которые следует записать, а `value` — адрес, из которого следует взять 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 отдельных `uniform`-матриц служит следующий класс функций:

```
void glUniformMatrix{234}fvARB ( int location, GLsizei count,
                                 GLboolean transpose, T value );
```

Параметр `count` содержит число матриц, которые необходимо записать (1 — для одной матрицы, больше единицы — для записи массива матриц), `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. Число 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 служит функция

```
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, число 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 число элементов соответствующего типа.

Для uniform-переменных типа sampler в качестве записываемого значения служит номер текстурного блока, в котором выбрана данная текстура; это значение считается целочисленным.

Расширение GL_ARB_vertex_shader

Данное расширение вводит целый ряд функций и констант для поддержки вершинных шейдеров. Далее приводится их полный список.

Список функций, вводимых расширением GL_ARB_vertex_shader

```
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 glVertexAttrib4NsvARB  ( 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 glGetVertexAttribdvARB ( GLuint index, GLenum pname,
                              double * params );
void glGetVertexAttribfvARB ( GLuint index, GLenum pname,
                              float * params );
void glGetVertexAttribivARB ( GLuint index, GLenum pname,
                              int * params );
```

```
void glGetVertexAttribPointervARB ( GLuint index, GLenum pname,
                                    void ** pointer );
```

Большинство из этих функций уже были рассмотрены в *главе 15*, посвященной вершинным программам (многие из вводимых этим расширением функций также вводятся расширением GL_ARB_vertex_program), однако есть и ряд специфичных для работы с шейдерами функций.

Одна из них — `glGetAttribLocationARB` — служит для получения индекса по имени атрибута.

Для получения информации об атрибуте по его индексу служит 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), число записанных в буфер байтов будет занесено в переменную по адресу `length`. Тип переменной будет записан в переменную, адрес которой содержится в `type`. Возможными значениями 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 служит функция `glBindAttribLocationARB`. Для стандартных атрибутов (т. е. начинающихся с префикса `gl_`) явное задание индекса не допускается. Явное задание индекса вступает в силу только после того, как программа будет слинкована заново.

Расширение GL_ARB_fragment_shader

Данное расширение добавляет необходимую поддержку для написания фрагментных шейдеров. Оно не вводит никаких дополнительных функций, а только несколько констант.

Получение информации о поддержке GLSL

Приводимая в листинге 18.1 программа проверяет, поддерживается ли GLSL, и в случае его поддержки выдает информацию об ограничениях на шейдеры.

Листинг 18.1. 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
int win = 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 поддерживаемую версию языка, а также ряд ограничений на шейдеры.

Величины, использованные в программе, определяют следующее:

- `maxVertexAttribs` — максимально возможное число атрибутов на вершину;

- `maxVertexTextureUnits` — максимальное число текстурных блоков, доступных вершинному шейдеру;
- `maxFragmentTextureUnits` — максимально возможное число текстурных блоков, доступных фрагментному шейдеру;
- `maxCombinedTextureUnits` — максимально возможное число текстурных блоков, доступных и вершинному, и фрагментному шейдерам;
- `maxVertexUniformComponents` — максимальный объем памяти (в вещественных числах) доступный вершинному шейдеру для создания `uniform`-переменных;
- `maxFragmentUniformComponents` — максимальный объем памяти (в вещественных числах) доступный фрагментному шейдеру для создания `uniform`-переменных;
- `maxVaryingFloats` — максимальный объем памяти (в вещественных числах), доступный для создания `varying`-переменных;
- `maxTextureCoords` — максимально возможное число текстурных координат, которые можно связать с вершиной.

Простейшая программа на GLSL

Опираясь на приведенные описания и фрагменты кода, можно написать простейшую программу на C++, использующую GLSL-шейдеры. В листинге 18.2 приводится исходный код такой программы, причем в качестве шейдеров выступают простейшие фрагменты (см. листинги 17.1 и 17.2).

Листинг 18.2. Программа на C++, использующая шейдеры на GLSL (glsl-example.cpp)

```
//
// 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;
//
// Returns 1 if an OpenGL error occurred, 0 otherwise.
//
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 );
    int          len  = data.getLength ();
    GLint   compileStatus;
    glShaderSourceARB ( shader, 1, &body,  &len  );
                                       // 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
    int win = 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 (см. листинги 17.1 и 17.2) и использует их для рендеринга чайника (рис. 18.1).

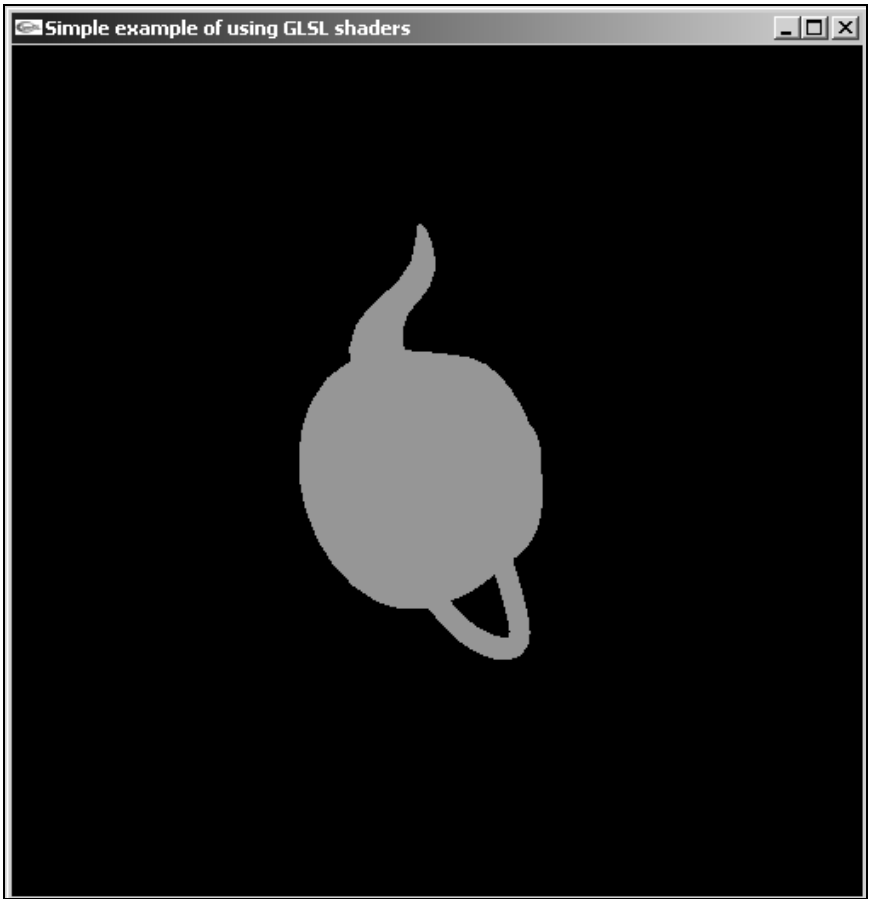


Рис. 18.1. Изображение, построенное программой из листинга 18.2

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 );
```

Затем 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 );
```

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 стандартные шаги, которые необходимо выполнять для каждой программы. Поэтому было бы очень удобно завернуть их в класс, спрятав от пользователя отдельные технические детали.

Заворачиваем шейдеры на GLSL в класс

В листинге 18.3 приводится описание класса `GlslProgram`, инкапсулирующего работу с GLSL-программами.

Листинг 18.3. Файл GlslProgram.h

```
//
// 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  ( int loc,
                                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        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 `GlslProgram` содержит ряд простых и удобных функций для поддержки шейдеров.

В первую очередь к ним относятся функции `isSupported` и `version`, позволяющие узнать, поддерживается ли GLSL данным графическим ускорителем и драйвером, и версию GLSL в случае поддержки.

Функции `loadShaders` служат для загрузки шейдеров, одновременно осуществляя компиляцию и линковку шейдеров. Все возникающие при этом ошибки накапливаются во внутренней переменной `log`.

Данный класс предоставляет также 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 18.4 приводится реализация данного класса.

Листинг 18.4. Файл GlslProgram.cpp

```
//
// 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   ( vertexFileName   );
    if ( !vertexData.isOk () || vertexData.isEmpty () )
        return false;
    Data    fragmentData ( fragmentFileName );
    if ( !fragmentData.isOk () || fragmentData.isEmpty () )
        return false;
    return loadShaders ( &vertexData, &fragmentData );
}
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
    vertexShader   = glCreateShaderObjectARB ( GL_VERTEX_SHADER_ARB   );
    fragmentShader = glCreateShaderObjectARB ( GL_FRAGMENT_SHADER_ARB );
    log += "Loading vertex shader\n";
    if ( !loadShader ( vertexShader, vertexShaderData ) )
        return false;
    log += "Loading fragment shader\n";
    if ( !loadShader ( fragmentShader, fragmentShaderData ) )
        return false;
                                     // attach shaders to program object
    glAttachObjectARB ( program, vertexShader   );
    glAttachObjectARB ( program, fragmentShader );
    GLint   linked;
```

```
    log += "Linking programs\n";
                                    // link the program object and print
                                   // out the info log
    glLinkProgramARB ( program );
    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 )
{
    const char * body = (const char *) data -> getPtr ( 0 );
    int          len  = data -> getLength ();
    GLint        compileStatus;
    glShaderSourceARB ( shader, 1, &body,  &len  );
                                         // compile the particle vertex
                                        // shader, and print out
    glCompileShaderARB ( shader );
    if ( !checkGlError() )                // check for OpenGL errors
        return false;
    glGetObjectParameterivARB ( shader, GL_OBJECT_COMPILE_STATUS_ARB,
                                &compileStatus );
    loadLog ( shader );
    return compileStatus != 0;
}
bool    GlslProgram :: checkGlError ()
{
    GLenum glErr;
    bool    retCode = true;
```

```
    for ( ; ; )
    {
        GLenum  glErr = glGetError();
        if ( glErr == GL_NO_ERROR )
            return retCode;
        glError = (const char *) gluErrorString ( glErr );
        retCode = false;
        glErr   = glGetError();
    }
    return retCode;
}
void    GlslProgram :: clear ()
{
    glDeleteObjectARB ( program        );         // it will also
                                                  // detach shaders
    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;
}
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 )
        return Vector4D ( 0, 0, 0, 0 );
    glGetUniformfvARB ( program, loc, values );
    return Vector4D ( values [0], values [1], values [2], values [3] );
}
bool    GlslProgram :: setTexture ( const char * name, int texUnit )
{
    int loc = glGetUniformLocationARB ( program, name );
```

```
    if ( loc == -1 )
        return false;
    glUniform1iARB ( loc, texUnit );
    return true;
}
bool    GlslProgram :: setTexture ( int loc, int texUnit )
{
    if ( loc < 0 )
        return false;
    glUniform1iARB ( loc, texUnit );
    return true;
}
bool    GlslProgram :: bindAttributeTo  ( int loc, const char * name )
{
    glBindAttribLocationARB ( program, loc, name );
    return true;
}
bool    GlslProgram :: setAttribute ( const char * name,
                                      const Vector4D& value )
{
    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 18.5 приводится вариант программы из листинга 18.2, модифицированный с применением класса `GlslProgram`.

Листинг 18.5. 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
    int win = 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;
    }
    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;
}
```

Примеры шейдеров на GLSL

Рассмотрим сначала реализацию простейшего случая — попиксельного диффузного и бликового освещения. Первым будет вариант, в котором никаких текстур нет вообще, т. е. все свойства и коэффициенты постоянны вдоль поверхности объекта (листинги 18.6 и 18.7).

Листинг 18.6. Вершинный шейдер для простейшего диффузного и бликового освещения

```
varying        vec3 lt;
varying        vec3 ht;
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
    vec3    l = normalize ( vec3 ( lightPos ) - p );
                // vector to the eye
    vec3    v = normalize ( vec3 ( eyePos )   - p );
    vec3    h = normalize ( l + v );
                // transformed n
    vec3    n = gl_NormalMatrix * gl_Normal;
                // transformed t
    vec3 t = gl_NormalMatrix * gl_MultiTexCoord1.xyz;
                // transformed b
    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;
}
```

Листинг 18.7. Фрагментный шейдер для простейшего диффузного и бликового освещения

```
varying        vec3 lt;
varying        vec3 ht;
void main (void)
{
    vec3    nt   = vec3 ( 0.0, 0.0, 1.0 );
    vec3    l2   = normalize ( lt );
    vec3    h2   = normalize ( ht );
    vec4    diff = vec4 ( 0.5, 0.0, 0.0, 1.0 ) *
                                        max ( dot ( nt, l2 ), 0.0 );
    vec4    spec = vec4 ( 0.7, 0.7, 0.0, 1.0 ) *
                                  pow ( max ( dot ( nt, h2 ), 0.0 ), 20 );
    gl_FragColor = diff + spec;
}
```

Реализация похожа на фрагментные программы. Через `varying`-переменные в фрагментный шейдер передаются направление на источник света и вектор

h в касательном пространстве. После их нормализации освещенность явно вычисляется по стандартным формулам.

Довольно легко переделать эти шейдеры для общего случая, когда при помощи текстур задаются карта нормалей, коэффициенты светимости, диффузной и бликовой освещенности. Соответствующие шейдеры приводятся в листингах 18.8 и 18.9.

Листинг 18.8. Вершинный шейдер для общего случая диффузного и бликового освещения

```
varying        vec3 lt;
varying        vec3 ht;
uniform        vec4 lightPos;
uniform        vec4 eyePos;
void main(void)
{
                    // do neccessary transformations
    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;
                    // compute tangent and binormal
    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 18.9. Фрагментный шейдер для общего случая диффузного и бликового освещения

```
varying        vec3 lt;
varying        vec3 ht;
uniform sampler2D bumpMap;
```

```
uniform sampler2D diffuseMap;
uniform sampler2D specularMap;
void main (void)
{
                    // get normal perturbation
    vec3    n    = texture2D ( bumpMap, gl_TexCoord [0] );
    vec3    nt   = normalize ( 2.0*n - 1.0 );
    vec3    l2   = normalize ( lt );
    vec3    h2   = normalize ( ht );
    float   diff = max ( dot ( nt, l2 ), 0.0 );
    float   spec = pow ( max ( dot ( nt, h2 ), 0.0 ), 30 );
    gl_FragColor = diff * texture2D ( diffuseMap, gl_TexCoord [0] ) +
                      spec * texture2D ( specularMap, gl_TexCoord [0] );
}
```

Модель освещения Гуч

Довольно интересный эффект дает модель освещения Гуч (Gooch). При этом, когда источник света оказывается позади объекта, объект окрашивается другим цветом, как бы прошедшим сквозь объект (рис. 18.2).

Сами шейдеры оказываются на удивление простыми (листинги 18.10 и 18.11).

Листинг 18.10. Вершинный шейдер для освещения Гуч

```
//
// Vertex shader for Gooch shading
// Based on 3Dlabs code
//
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();
}
```

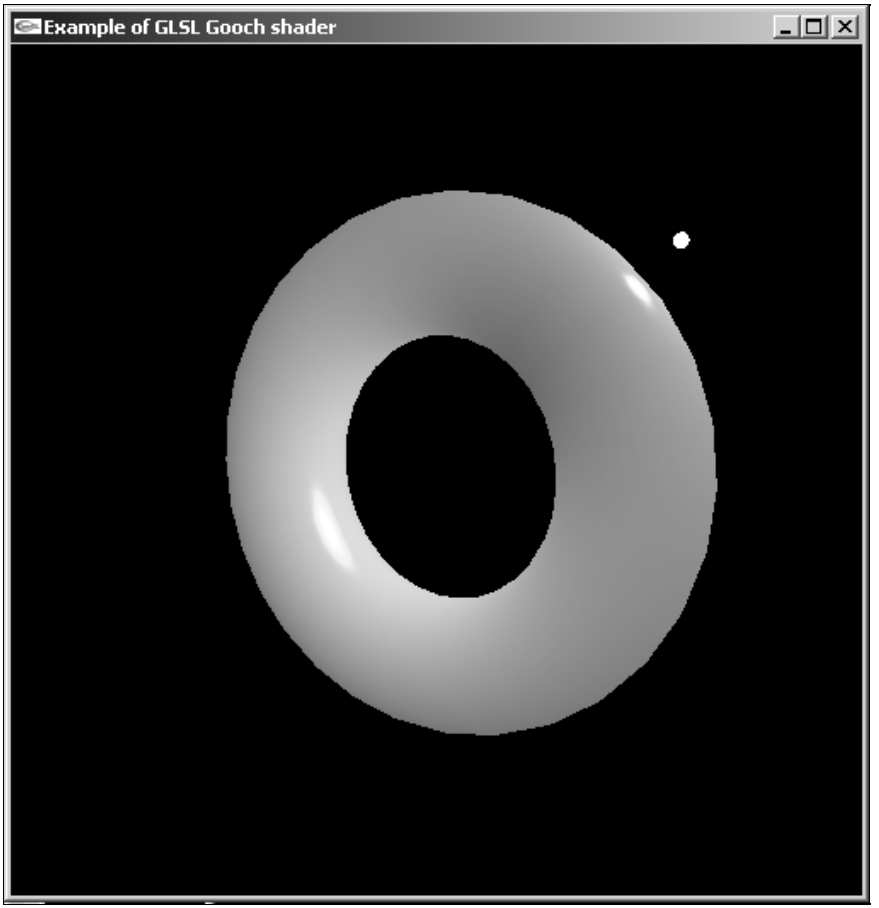


Рис. 18.2. Изображение тора с использованием модели освещения Гуч

Листинг 18.11. Фрагментный шейдер для освещения Гуч

```
//
// Fragment shader for Gooch shading
//
```

```
// Based on 3Dlabs code by Randi Rost
//
uniform vec3  SurfaceColor; // (0.75, 0.75, 0.75)
uniform vec3  WarmColor;    // (0.6, 0.6, 0.0)
uniform vec3  CoolColor;    // (0.0, 0.0, 0.6)
uniform float DiffuseWarm;  // 0.45
uniform float DiffuseCool;  // 0.45
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 );
}
```

Учет интерференции в тонком слое

Очень красивым эффектом, довольно легко реализуемым с помощью шейдеров, является интерференция в тонком слое.

Если на поверхности объекта присутствует очень тонкий прозрачный слой, то будет происходить интерференция между светом, отраженным от этого слоя, и от границы между объектом и слоем (рис. 18.3).

Это приводит к появлению на поверхности объекта радужных полос, как на мыльных пузырях.

Данный эффект довольно легко смоделировать, применив следующий подход (в основу данного примера лег код из NVSDK). Цвет, получающийся в результате интерференции, зависит от пути, пройденного внутри слоя. А этот путь можно выразить как $2h/(n, v)$, где через h обозначена толщина

слоя (мы считаем ее неизменной вдоль всей поверхности), а через n и v — единичные векторы нормали и направления на наблюдателя.

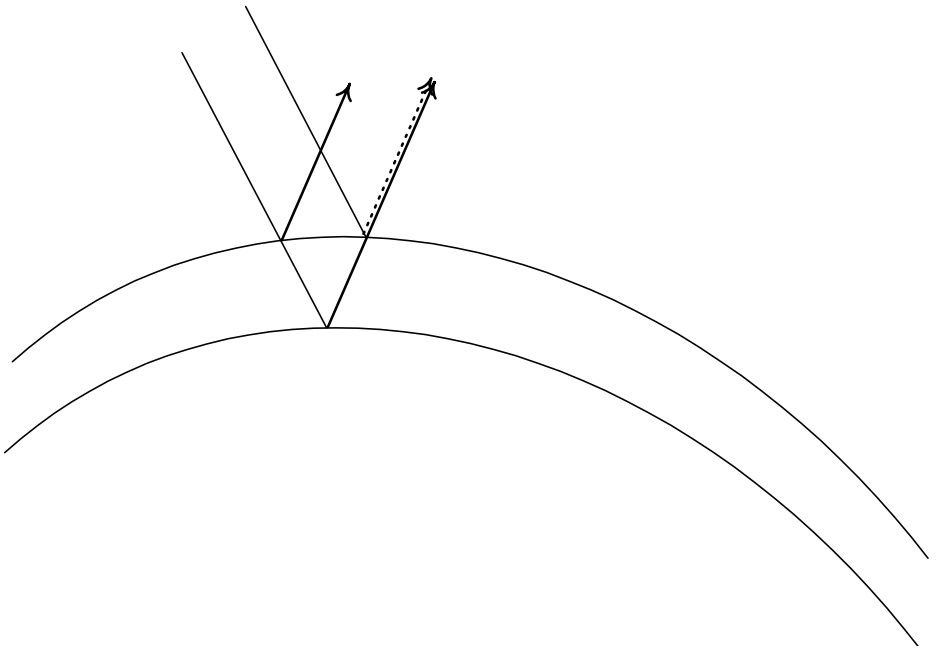


Рис. 18.3. Интерференция в тонком слое

Эту величину довольно просто вычислить в вершинном шейдере, а далее ее можно передать в виде индекса в одномерную текстуру, содержащую значения цветов, получающиеся при интерференции.

На компакт-диске в каталоге *scripts* вы найдете простую программу на языке Python, строящую данную текстуру, а в листингах 18.12 и 18.13 приводятся соответствующие вершинные и фрагментные шейдеры.

Листинг 18.12. Вершинный шейдер, реализующий эффект интерференции в тонком слое

```
//
// Thin-layer effect (based on NVidia shader)
//
uniform vec4    lightPos;
uniform vec4    eyePos;
varying vec3    diffColor;
varying vec3    specColor;
```

```
varying vec2    viewDepth;
void main()
{
    const    float    filmDepth = 0.4;
    gl_Position = ftransform ();    // transform position to clip space
                        // 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 );
                        // compute the eye->vertex vector
    vec3 v = normalize ( eyePos.xyz - p );
    vec3 l = normalize ( lightPos - p );
    vec3 h = normalize ( l + v );
    float diffuse  = max ( dot ( n, l ), 0.0 );
    float specular = pow ( max ( dot ( n, h ), 0.0 ), 30 );
                        // compute the view depth for the
                        // thin film
    viewDepth = vec2 ( filmDepth / dot ( n, v ) );
                        // output final lighting results
    diffColor = vec3 ( diffuse  );
    specColor = vec3 ( specular );
                        // output texture coordinates
                        // for diffuse map
    gl_TexCoord [0] = gl_MultiTexCoord0;
}
```

Листинг 18.13. Фрагментный шейдер, реализующий эффект интерференции в тонком слое

```
//
// Thin-layer effect (based on NVidia shader)
//
uniform sampler2D diffuseMap;
uniform sampler1D fringeMap;
varying vec3 diffColor;
varying vec3 specColor;
varying vec2 viewDepth;
void main()
```

```
{
                    // diffuse material color
    vec3 diffMaterial = texture2D ( diffuseMap, gl_TexCoord [0].xy ).rgb;
                    // lookup fringe value based on view depth
    vec3 fringeColor = texture1D ( fringeMap, viewDepth.x ).rgb;
                    // modulate specular lighting by fringe color
                    // and combine with regular lighting
    gl_FragColor = vec4 ( diffColor * diffMaterial +
                          fringeColor * specColor, 1.0 );
}
```

Шейдер, использующий шумовую функцию

С помощью шумовой функции можно создавать шейдеры, моделирующие естественные материалы или явления.

Поскольку не все реализации GLSL поддерживают шумовую функцию, в следующих примерах она заменяется двумерными и трехмерными текстурами, содержащими значение периодической шумовой функции (обратите внимание, что в этом случае крайне важна периодичность текстуры, иначе в изображении будут присутствовать разрывы цветов, положений и т. п.).

Классическим примером шейдера, основанного на шумовой функции, является *eroded*, впервые описанный в книге *Rendermap Companion*[1], моделирующий появление на поверхности ржавчины (рис. 18.4).

Эффект "ржавчины" получается путем сложения значений шумовой функции с разными масштабами, что позволяет передать фрактальную природу этого явления.

Листинги 18.14 и 18.15 содержат соответствующие вершинный и фрагментный шейдеры.

Листинг 18.14. Вершинный шейдер для "ржавого" тора

```
//
// Eroded vertex shader (a'la RenderMan's)
//
varying float    diffuse;
varying float    specular;
varying vec2     tex;
uniform vec4     lightPos;
uniform vec4     eyePos;
```

[1] The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics Steve Upstill. Addison — Wesley Publishing Company, 1990.

```
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 = normalize ( gl_NormalMatrix * gl_Normal );
    diffuse  = max ( dot ( n, l ), 0.0 );
    specular = pow ( max ( dot ( n, h ), 0.0 ), 100 );
    tex      = gl_MultiTexCoord0.xy;
    tex.y   *= 2.0;
    gl_Position     = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord [0] = gl_MultiTexCoord0;
}
```

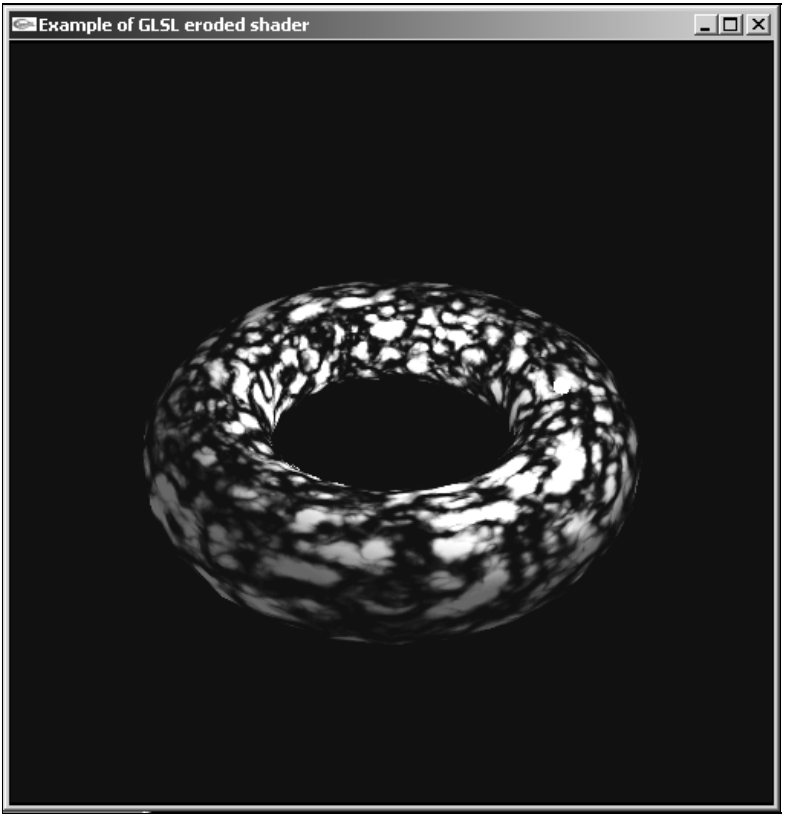


Рис. 18.4. "Ржавый" тор

Листинг 18.15. Фрагментный шейдер для "ржавого" тора

```
//
// Eroded fragment shader (a'la RenderMan's)
//
varying float        diffuse;
varying float        specular;
varying vec2        tex;
uniform sampler2D    myNoise;
void main (void)
{
    const    vec4    diffColor = vec4 ( 1.0, 0.0, 0.0, 1.0 );
    const    vec4    specColor = vec4 ( 1.0, 0.0, 0.0, 1.0 );
    const    vec3    half      = vec3 ( 0.5, 0.5, 0.0 );
                    // compute turbulence
    vec2    arg  = tex + vec2 ( 0.123, 0.345 );
    vec2    arg1 = arg;
    vec2    arg2 = 2.0 * arg + vec2 ( 0.537, 0.953 );
    vec2    arg3 = 4.0 * arg + vec2 ( 0.463, 0.543 );
    float    n1   = 2*abs ( texture2D ( myNoise, arg1 ).x - half ) /
                                                              1.0;
    float    n2   = 2*abs ( texture2D ( myNoise, arg2 ).x - half ) /
                                                               2.0;
    float    n3   = 2*abs ( texture2D ( myNoise, arg3 ).x - half ) /
                                                               4.0;
    float    turb = n1 + n2 + n3;
                    // compute f value
    float    f     = smoothstep ( 0.1, 0.5, turb );
    float    Ka    = 0.2 * f;
    float    Kd    = 0.7 * f;
    float    Ks    = 0.7 * f;
    gl_FragColor   = Ka* diffColor + diffuse * Kd * diffColor +
                        specular * Ks * specColor;
    gl_FragColor.a = 1.0 - f;
}
```

Еще одним примером использования шумовых текстур является создание моря, когда волны хаотично перемещаются на его поверхности. Это довольно легко можно реализовать, если задать с помощью шумовой функ-

ции нормаль к поверхности (точнее сумму с масштабирующими коэффициентами).

На рис. 18.5 приведено изображение морской сцены, реализованной сразу двумя шейдерами — для воды и неба. В листингах 18.16—18.20 приведены исходные шейдеры для получения этого изображения и соответствующий код на C++.

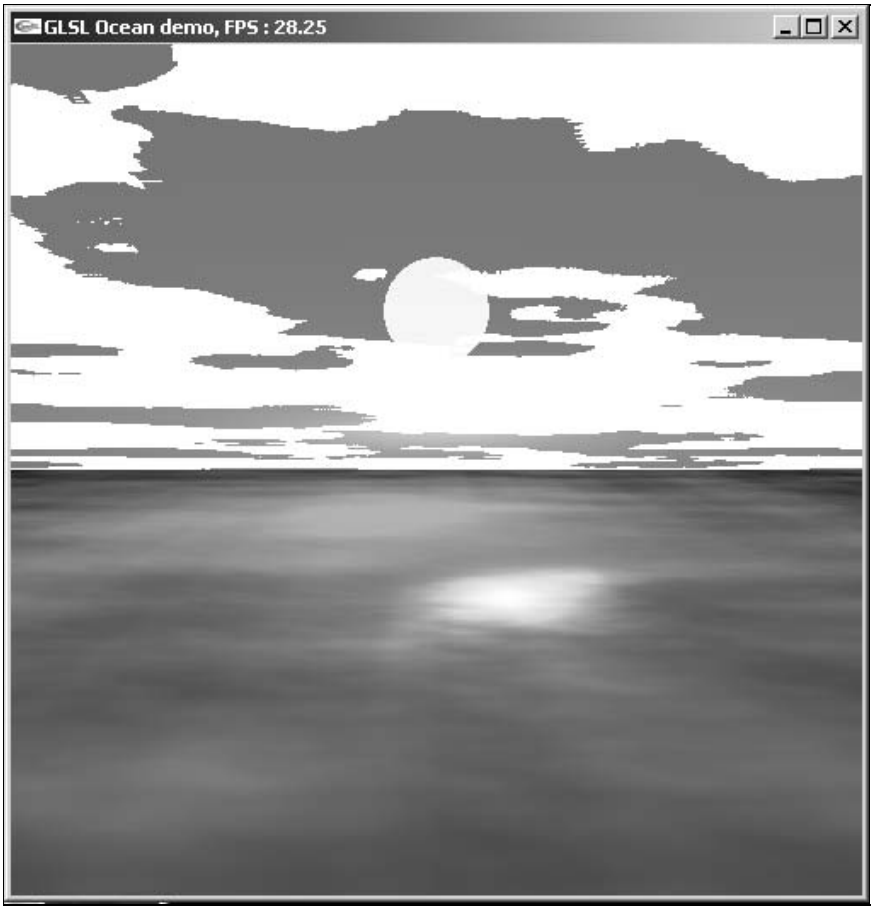


Рис. 18.5. Море

Листинг 18.16. Вершинный шейдер "моря"

```
varying     vec3 lt;
varying     vec3 ht;
```

```
uniform     vec4    lightPos;
uniform     vec4    eyePos;
void main(void)
{
    vec3    p = vec3      ( gl_ModelViewMatrix * gl_Vertex );
    vec3    v = normalize ( vec3 ( eyePos )   - p );
    lt = normalize ( vec3 ( lightPos ) - p );
    ht = normalize ( lt + v );
    gl_Position     = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord [0] = gl_MultiTexCoord0;
}
```

Листинг 18.17. Фрагментный шейдер "моря"

```
varying     vec3 lt;
varying     vec3 ht;
uniform     vec4        time;
uniform     sampler3D    myNoise;
uniform     sampler2D    myNoise2;
void main (void)
{
    const    vec3    ambColor  = vec3 ( 0.0, 0.1, 0.1 );
    const    vec3    diffColor = vec3 ( 0.0, 1.0, 0.9 );
    const    vec3    specColor = vec3 ( 1.0, 0.8, 0.5 );
    const    vec3    up        = vec3 ( 0.0, 1.0, 0.0 );
    const    vec3    half      = vec3 ( 0.5, 0.0, 0.5 );
    vec3    arg1 = arg;
    vec3    arg2 = 4.1235 * arg;
    vec3    arg3 = 8.3173 * arg;
    vec3    n1   = (texture3D ( myNoise, arg1 ).xyz - half) * 2.0;
    vec3    n2   = (texture3D ( myNoise, arg2 ).xyz - half) / 2.0;
    vec3    n3   = (texture3D ( myNoise, arg3 ).xyz - half) / 4.0;
    vec3    n    = normalize ( n1 + n2 + n3 + up );
    vec3    l2   = normalize ( lt );
    vec3    h2   = normalize ( ht );
    float    diff = max ( dot ( n, l2 ), 0.0 );
    float    spec = pow ( max ( dot ( n, h2 ), 0.0 ), 20 );
```

```
    gl_FragColor = vec4 ( ambColor + diff * diffColor +
                          spec * specColor, 1.0 );
}
```

Листинг 18.18. Вершинный шейдер "неба"

```
//
// Simple sky vertex shader
//
varying     vec3 l;
varying     vec3 v;
uniform     vec4    lightPos;
uniform     vec4    eyePos;
void main(void)
{
    vec3    p = vec3 ( gl_ModelViewMatrix * gl_Vertex );
    v = vec3 ( eyePos )   - p;        // vector to the eye
    l = vec3 ( lightPos ) - p;        // vector to light source
    gl_Position     = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord [0] = gl_MultiTexCoord0;
}
```

Листинг 18.19. Фрагментный шейдер "неба"

```
//
// Simple sky vertex shader
//
varying vec3 l;
varying     vec3 v;
uniform     vec4        time;
uniform     sampler3D    myNoise;
uniform     sampler2D    myNoise2;
void main (void)
{
    const    vec4    skyColorLight  = vec4 ( 0.5, 0.9, 1.0, 1.0 );
    const    vec4    skyColorNormal = vec4 ( 0.0, 0.6, 0.6, 1.0 );
    const    vec4    cloudColor     = vec4 ( 1.0, 1.0, 1.0, 1.0 );
    const    vec3    maxLight       = vec3 ( 1.0, 0.7, 2.1 );
```

```
    float    t    = 0.01 * time.x;
    vec3     tm   = vec3 ( t ) + 0.3 * texture2D ( myNoise2,
                                vec2 ( 0.3579, 0.001 * time.x ) );
    vec3     arg  = vec3 ( gl_TexCoord [0].xy + vec2 ( 0.246, 0.7531 ),
                                0.777 ) + tm;
    vec3     arg1 = 2.1397  * ( arg + vec3 ( 0.0, time.x * 0.0075,
                                0.22221 ) );
    vec3     arg2 = 7.9793  * ( arg + vec3 ( -time.x * 0.0058, 0.1,
                                 0.33 ) );
    vec3     arg3 = 15.9191 * ( arg + vec3 ( 0.1313, time.x * 0.004,
                                 0.777 ) );
    vec3     n1   = texture3D ( myNoise, arg1 ).xyz / 1.0;
    vec3     n2   = texture3D ( myNoise, arg2 ).xyz / 4.0;
    vec3     n3   = texture3D ( myNoise, arg3 ).xyz / 8.0;
    vec3     n    = n1 + n2 + n3;
    float    f        = normalize ( l ).y;
    vec4     skyColor = mix ( skyColorNormal, skyColorLight, f );
                    // now compute clouds based on n
    float    factor = step ( 0.67, n.x );
    vec4     cloud = mix ( skyColor, cloudColor, factor );
    gl_FragColor = cloud;
}
```

Листинг 18.20. Исходная программа на C++ для "морской" сцены

```
//
// Sample showing how to use GLSL programs
//
#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    "Torus.h"
Vector3D    eye   ( 0, 0, 1 );           // camera position
Vector3D    light ( 0, 1.5, -6 );        // light position
float       angle = 0;
Vector3D    rot ( 0, 0, 0 );
int         mouseOldX = 0;
int         mouseOldY = 0;
unsigned    noiseMap;
unsigned    noise2Map;
GlslProgram program;
GlslProgram skyProgram;
int         frames       = 0;
int         totalFrames  = 0;
int         elapsedTime  = 0;
float       fps          = 0;
const char * caption = "GLSL Ocean demo";
void    updateFps ()
{
    frames++;
    totalFrames++;
    if ( (frames % 5) == 0 )                // compute every 5 frames
    {
        int     time2 = glutGet ( GLUT_ELAPSED_TIME );
        char    str [256];
        fps         = (float)frames / (0.001 * (time2 - elapsedTime));
        elapsedTime = time2;
        frames      = 0;
        sprintf ( str, "%s, FPS : %5.2f", caption, fps );
        glutSetWindowTitle ( str );
    }
}
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 ();
                                          // the sun
    glColor3f       ( 1, 0.8, 0 );
    glTranslatef    ( light.x, light.y, light.z );
    glutSolidSphere ( 0.5, 30, 30);
    glTranslatef    ( -light.x, -light.y, -light.z );
    glEnable    ( GL_BLEND );
    glBlendFunc ( GL_ONE, GL_ONE );
                                                  // the sky
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glBindTexture      ( GL_TEXTURE_3D, noiseMap );
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glBindTexture      ( GL_TEXTURE_2D, noise2Map );
    glColor3f       ( 0, 0.8, 0.8 );
    skyProgram.bind ();
    glBegin ( GL_QUADS );
        glTexCoord2f ( 0.0, 0.0);
        glVertex3f   ( -10.0, 1.0, 1.0);
        glTexCoord2f ( 1.0, 0.0);
        glVertex3f   ( 10.0, 1.0, 1.0);
        glTexCoord2f ( 1.0, 1.0);
        glVertex3f   ( 10.0, 0.0, -10.0);
        glTexCoord2f ( 0.0, 1.0);
        glVertex3f   ( -10.0, 0.0, -10.0);
    glEnd ();
    skyProgram.unbind ();
    glDisable ( GL_BLEND );
                                                 // the ocean
    glColor3f       ( 0, 0.4, 0.8 );
    program.bind ();
    glBegin ( GL_QUADS );
        glTexCoord2f ( 0, 0 );
```

```
        glVertex3f   ( -10.0, -1.0, 1.0 );
        glTexCoord2f ( 1, 0 );
        glVertex3f   ( 10.0, -1.0, 1.0 );
        glTexCoord2f ( 1, 1 );
        glVertex3f   ( 10.0, 0.0, -10.0 );
        glTexCoord2f ( 0, 1 );
        glVertex3f   ( -10.0, 0.0, -10.0 );
    glEnd();
    program.unbind ();
    glPopMatrix ();
    updateFps ();
    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,
                     0, 0, 0,
                     0, 1, 0 );
    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  = 0.001f * glutGet ( GLUT_ELAPSED_TIME );
    program.bind ();
    program.setUniformVector ( "eyePos",   Vector4D ( eye,   1 ) );
    program.setUniformVector ( "lightPos", Vector4D ( light, 1 ) );
    program.setUniformVector ( "time",     Vector4D ( angle, 0, 0, 0 ) );
    program.unbind ();
    skyProgram.bind ();
    skyProgram.setUniformVector ("eyePos",  Vector4D (eye,   1 ) );
    skyProgram.setUniformVector ("lightPos",Vector4D (light, 1 ) );
    skyProgram.setUniformVector ("time",    Vector4D (angle, 0, 0, 0));
    skyProgram.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
    int win = glutCreateWindow ( caption );
                                // 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 ( "ocean.vsh", "ocean.fsh" ) )
    {
        printf ( "Error loading ocean shaders:\n%s\n",
                  program.getLog ().c_str () );
        return 3;
    }
    if ( !skyProgram.loadShaders ( "sky.vsh", "sky.fsh" ) )
    {
        printf ( "Error loading sky shaders:\n%s\n",
                 skyProgram.getLog ().c_str () );
        return 3;
    }
    noiseMap  = createTexture3D(false, "../Textures/Noise/noise-3D.dds");
    noise2Map = createTexture2D(true, "noise-1.png");
                                    // install program object as part of
                                    // current state
    program.bind ();
    program.setTexture ( "myNoise",  0 );
    program.setTexture ( "myNoise2", 1 );
    program.unbind ();
    skyProgram.bind ();
    skyProgram.setTexture ( "myNoise",  0 );
```

```
    skyProgram.setTexture ( "myNoise2", 1 );
    skyProgram.unbind ();
    glutMainLoop ();
    return 0;
}
```

Эффект "старого фильма"

Шейдеры на GLSL пригодны также для постобработки результатов рендеринга, причем на нем их писать гораздо легче, чем на ассемблере фрагментных программ.

Следующий шейдер реализует эффект, придающий изображению вид старого черно-белого кинофильма (рис. 18.6).

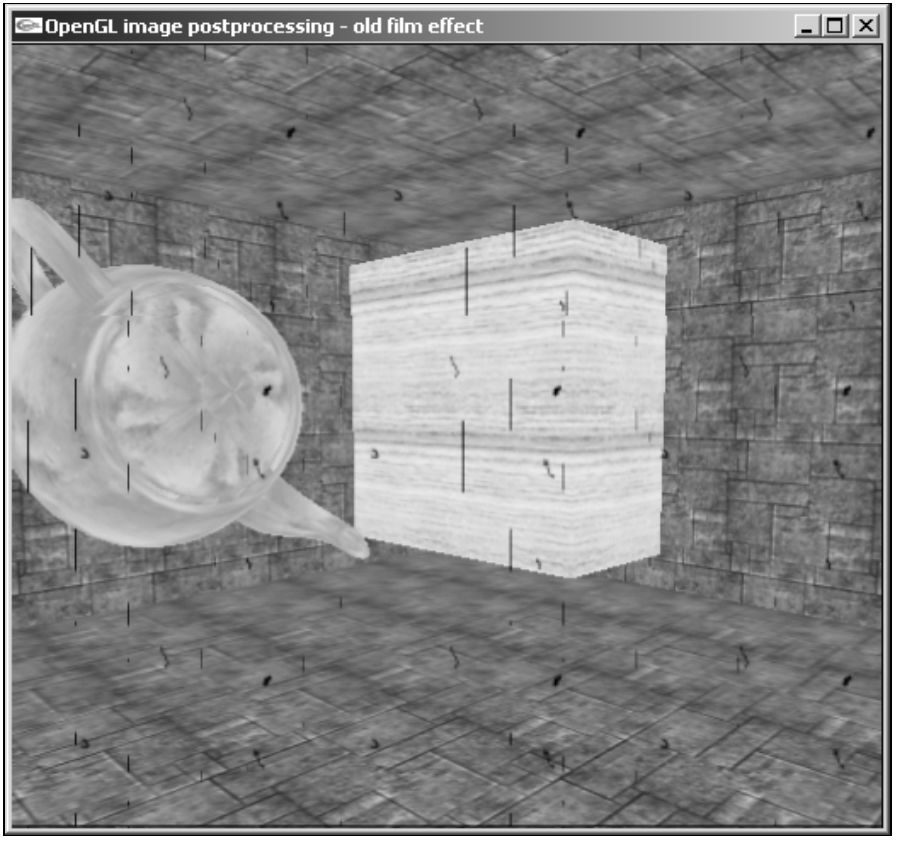


Рис. 18.6. Изображение, полученное при помощи эффекта "старого фильма"

Характерными элементами этого фильма (не с художественной точки зрения) являются изображения в оттенках серого (что мы уже делали в эффекте *sepia*), причем картинка все время слегка дергается случайным образом, и на нее накладываются вертикальные черные полосы и случайно появляющиеся (и так же исчезающие) пылинки.

Как переводить изображение в оттенок серого, мы уже рассматривали в *главе 16*. Для этого достаточно всего одного скалярного произведения. "Дергание" изображения легко достигается небольшим сдвигом текстурных координат, для обеспечения "случайности" можно взять значения какой-либо из тригонометрических функций, передавая аргументом значение текущего времени с очень большим масштабирующим множителем.

Формирование полос и пылинок — это просто наложение готовых текстур случайным образом.

Листинги 18.21 и 18.22 содержат вершинный и фрагментный шейдеры для такого эффекта, а листинг 18.23 — исходный код на C++.

Листинг 18.21. Вершинный шейдер для эффекта "старого фильма"

```
//
// Old-film vertex shader as a post-processing effect
//
uniform float    time;
varying vec2    shudder;
varying float    brightness;
varying vec2    blipCoord;
varying vec2    scratchCoord;
void main(void)
{
    shudder      = 0.0015 * vec2 ( sin ( time * 111.0 ),
                                   cos ( time * 157.3 ) );
    brightness   = clamp ( 3.0 * sin ( time * 5.7 ), 0.7, 1.0 );
    blipCoord    = 3.0 * gl_MultiTexCoord0.xy +
                        vec2 ( sin ( 20.0 * time ) + cos ( 7 * time ),
                               cos ( 35.0 * time ) + sin ( 13.0 * time ) );
    scratchCoord = gl_MultiTexCoord0.xy +
                        vec2 ( 0.01 * cos ( 1234.0 * time ),
                               0.2 * sin ( 157.3 * time ) );
    gl_Position     = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord [0] = gl_MultiTexCoord0;
}
```

Листинг 18.22. Фрагментный шейдер для эффекта "старого фильма"

```
//
// Old-film vertex shader as a post-processing effect
//
uniform float    time;
varying vec2    shudder;
varying float    brightness;
varying vec2    blipCoord;
varying vec2    scratchCoord;
uniform sampler2D    mainTex;
uniform sampler2D    scratchTex;
uniform sampler2D    blipTex;
void main (void)
{
    const    vec3    luminance = vec3 ( 0.3, 0.59, 0.11 );
    vec4    color   = texture2D ( mainTex,    gl_TexCoord [0].xy +
                                  shudder );
    vec4    scratch = texture2D ( scratchTex, 2.0*scratchCoord );
    vec4    blip    = texture2D ( blipTex,    blipCoord    );
    gl_FragColor   = vec4 ( dot ( color.rgb, luminance ) ) *
                               brightness * scratch * blip;
    gl_FragColor.a = 1.0;
}
```

Листинг 18.23. Исходный код на C++

```
//
// Sample to image postprocessing via p-buffer 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    "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    scratchMap;
unsigned    blipMap;
float       angle     = 0;
float       rot       = 0;
bool        useFilter = true;
GlslProgram    program;
PBuffer        pbuffer ( 512, 512, PBuffer :: modeTextureMipmap |
                          PBuffer :: modeAlpha | PBuffer :: modeDepth |
                          PBuffer :: modeTexture2D, true );
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 projection matrix
    glMatrixMode ( GL_MODELVIEW );       // select the modelview matrix
    glPopMatrix  ();                     // restore projection matrix
}
void display ()
{
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    renderToBuffer ();
    if ( !pbuffer.bindToTexture () )
        pbuffer.printLastError ();
    startOrtho ();
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    glBindTexture      ( GL_TEXTURE_2D, pbuffer.getTexture () );
    glActiveTextureARB ( GL_TEXTURE1_ARB );
    glBindTexture      ( GL_TEXTURE_2D, scratchMap );
    glActiveTextureARB ( GL_TEXTURE2_ARB );
    glBindTexture      ( GL_TEXTURE_2D, blipMap );
    glActiveTextureARB ( GL_TEXTURE0_ARB );
    if ( useFilter )
        program.bind ();
```

```
    glBegin ( GL_QUADS );
        glTexCoord2f ( 0, 0 );
        glVertex2f   ( 0, 0 );
        glTexCoord2f ( 1,   0 );
        glVertex2f   ( 511, 0 );
        glTexCoord2f ( 1, 1 );
        glVertex2f   ( 511, 511 );
        glTexCoord2f ( 0, 1 );
        glVertex2f   ( 0, 511 );
    glEnd   ();
    if ( useFilter )
        program.unbind ();
    endOrtho ();
    if ( !pbuffer.unbindFromTexture () )
        pbuffer.printLastError ();
    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 )
```

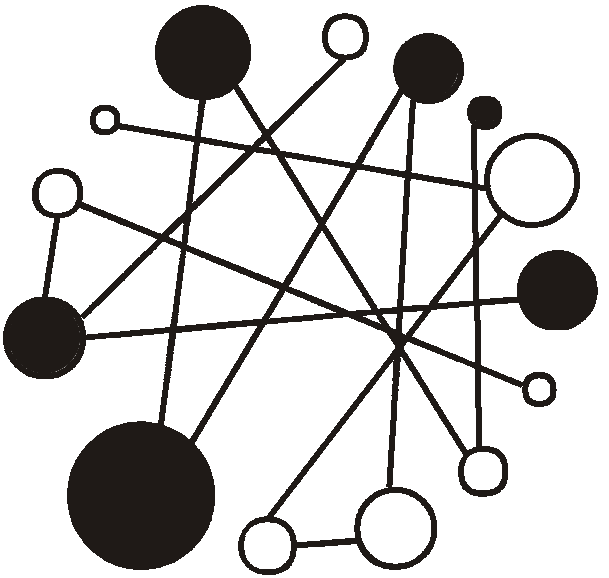
```
        rot += 5;
    else
    if ( key == GLUT_KEY_LEFT )
        rot -= 5;
    glutPostRedisplay ();
}
void    animate ()
{
    angle  = 0.004f * glutGet ( GLUT_ELAPSED_TIME );
    program.bind ();
    program.setUniformFloat ( "time", angle );
    program.unbind ();
    glutPostRedisplay ();
}
void    initPBuffer ()
{
    static    bool    inited = false;
    if ( inited )
        return;
    inited = true;
    if ( !pbuffer.create () )
    {
        printf ( "create error\n" );
        pbuffer.printLastError ();
    }
    pbuffer.makeCurrent ();
    init ();
    reshape ( pbuffer.getWidth (), pbuffer.getHeight () );
    pbuffer.restoreCurrent ();
}
void    renderToBuffer ()
{
    if ( pbuffer.isLost () )
        pbuffer.create ();
    if ( !pbuffer.makeCurrent () )
        printf ( "makeCurrent failed\n" );
    glClearColor ( 0, 0, 1, 1 );
    glClear      ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

```
    reshape ( pbuffer.getWidth (), pbuffer.getHeight () );
    displayBoxes ();
    pbuffer.restoreCurrent ();
}
int main ( int argc, char * argv [] )
{
                        // initialize glut
    glutInit            ( &argc, argv );
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize  ( 512, 512 );
                        // create window
    int    win = glutCreateWindow ( "OpenGL image postprocessing - old
film effect" );
                        // register handlers
    glutDisplayFunc  ( display    );
    glutReshapeFunc  ( reshape    );
    glutKeyboardFunc ( key        );
    glutSpecialFunc  ( specialKey );
    glutIdleFunc     ( animate    );
    init ();
    initExtensions ();
    if ( !pbuffer.checkExtensions () )
    {
        printf ( "Pbuffer extensions not found\n" );
        return 1;
    }
    assertExtensionsSupported ( "GL_ARB_shading_language_100
GL_ARB_shader_objects" );
    if ( !program.loadShaders ( "film.vsh", "film.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" );
    scratchMap = createTexture2D ( true,
                                   "../Textures/Oldfilm/scratches.jpg" );
```

```
    blipMap    = createTexture2D ( true,
                                "../Textures/Oldfilm/blips.jpg"    );
    program.bind ();
    program.setTexture ( "mainTex",    0 );
    program.setTexture ( "scratchTex", 1 );
    program.setTexture ( "blipTex",    2 );
    program.unbind ();
    printf ( "Press F key to turn filtering on/off\n" );
    initPBuffer  ();
    glutMainLoop ();
    return 0;
}
```

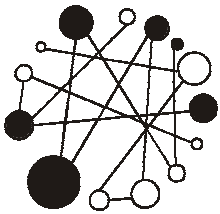
К сожалению, объем книги не позволяет включить и подробно описать все примеры для иллюстрации материала данной главы, находящиеся на прилагаемом компакт-диске. Скорее это лишь первое знакомство с языком GLSL и его возможностями.

Много интересных шейдеров вы можете найти в Интернете; правда, они могут быть не только на GLSL, но и на Cg или HLSL (в основу многих приведенных здесь шейдеров легли примеры из Интернета). Однако все эти языки имеют между собой столько общего, что при необходимости вы легко сможете переписать интересующие примеры на GLSL.



Приложения

Приложение 1



Основы линейной алгебры

Основными понятиями из курса линейной алгебры, которые необходимы для лучшего восприятия материала книги, являются векторы и матрицы из вещественных чисел.

Двумерные векторы и матрицы

Рассмотрим сначала соответствующие понятия для двумерного случая. Упорядоченная пара вещественных чисел (x, y) называется двумерным вектором, множество всех двумерных векторов обозначается через R^2.

Эту пару (x, y) можно записать и в виде столбца $\begin{pmatrix} x \\ y \end{pmatrix}$, и в виде строки из двух чисел $\begin{pmatrix} x & y \end{pmatrix}$. В первом случае мы говорим о том, что у нас есть вектор-столбец, а во втором — вектор-строка.

Пару из двух нулей (0, 0) называют нулевым вектором и обозначают O.

Для двумерных векторов легко вводятся операции умножения на число, сложения и вычитания:

$$\lambda\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \lambda x \\ \lambda y \end{pmatrix},$$

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \pm \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 \pm x_2 \\ y_1 \pm y_2 \end{pmatrix}.$$

Также можно ввести и покомпонентное умножение векторов:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \times \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 \times x_2 \\ y_1 \times y_2 \end{pmatrix}.$$

Важной операцией является скалярное произведение двух векторов (*dot product*).

$$\left(\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}\right) = x_1 \cdot x_2 + y_1 \cdot y_2 .$$

Иногда скалярное произведение векторов v_1 и v_2 обозначают не только как (v_1, v_2), но и как $v_1 \cdot v_2$.

Легко заметить, что скалярный квадрат вектора $v = (x, y)$ всегда неотрицателен ($(v, v) = x^2 + y^2$) и, следовательно, всегда определен квадратный корень из него, называемый длиной или нормой вектора v:

$$|v| = \sqrt{x^2 + y^2} .$$

Длина вектора обозначается как $|v|$ или $\|v\|$.

Далее перечислены основные свойства операций над векторами:

- $u + v = v + u$;
- $(u + v) + w = u + (v + w)$;
- $0 + u = u + 0 = u$;
- $u + (-1) \cdot u = 0$;
- $\alpha(\beta u) = (\alpha\beta) u$;
- $(\alpha + \beta) u = \alpha u + \beta u$;
- $\alpha(u + v) = \alpha u + \alpha v$;
- $1 \cdot u = u$;
- $(u + v,\ w) = (u,\ w) + (v,\ w)$;
- $(\alpha u,\ v) = \alpha(u,\ v)$;
- $(u,\ v) = (v,\ u)$;
- $|\alpha u| = |\alpha| \cdot |u|$.

Справедливы также следующие два неравенства:

- $|(u, v)| \le |u| \cdot |v|$;
- $|u + v| \le |u| + |v|$.

Используя скалярное произведение, можно ввести понятие угла φ между векторами u и v следующим образом:

$$\cos\varphi = \frac{(u,\ v)}{|u| \cdot |v|} .$$

Двумерной матрицей M (матрицей размера 2×2) называется таблица из четырех вещественных чисел вида

$$M = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}.$$

Такую матрицу можно рассматривать как упорядоченный набор из двух векторов-столбцов $\begin{pmatrix} a_{11} \\ a_{21} \end{pmatrix}$ и $\begin{pmatrix} a_{12} \\ a_{22} \end{pmatrix}$, и как набор из двух векторов-строк $(a_{11} \quad a_{12})$ и $(a_{21} \quad a_{22})$.

Главной диагональю матрицы M называют элементы a_{11} и a_{22}.

Для матриц, как и для векторов, вводятся операции умножения на число, сложения и вычитания (покомпонентным образом):

$$\lambda \cdot \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} \lambda a_{11} & \lambda a_{12} \\ \lambda a_{21} & \lambda a_{22} \end{pmatrix},$$

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \pm \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} \pm b_{11} & a_{12} \pm b_{12} \\ a_{21} \pm b_{21} & a_{22} \pm b_{22} \end{pmatrix}.$$

Вводятся также понятия нулевой O и единичной (обозначаемой как I или E) матриц:

$$O = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix},$$

$$I = E = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Кроме описанных операций для матриц вводится операция транспонирования:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}^T = \begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{pmatrix}.$$

Как несложно заметить, эта операция "переворачивает" матрицу относительно ее главной диагонали.

Операцию транспонирования можно обобщить и для векторов. В этом случае она преобразует вектор-столбец в вектор-строку и наоборот:

$$(x \quad y)^T = \begin{pmatrix} x \\ y \end{pmatrix},$$

$$\begin{pmatrix} x \\ y \end{pmatrix}^T = (x \quad y).$$

Таким образом, вектор-столбец отличается от вектора-строки только транспонированием.

Для матриц 2×2 вводится операция умножения (обратите внимание, что эта операция не покомпонентная):

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}.$$

Как можно заметить, элемент (i, j) в произведении матриц A и B задается следующей формулой:

$$c_{ij} = \sum_{k=1}^{2} a_{ik} \cdot b_{kj} .$$

Произведение матриц обладает целым рядом интересных свойств.

- Некоммутативность: $A \cdot B \neq B \cdot A$.
- Ассоциативность: $A \cdot (B \cdot C) = (A \cdot B) \cdot C$.
- Дистрибутивность: $A \cdot (B + C) = A \cdot B + A \cdot C$.
- $A \cdot I = I \cdot A = A$.
- $A \cdot O = O \cdot A = O$.
- $(A \cdot B)^T = B^T \cdot A^T$.

Матрица B называется обратной для матрицы A (и обозначается A^{-1}), если выполняется следующее равенство:

$$A \cdot B = B \cdot A = I .$$

Можно ввести операции умножения матрицы на вектор-столбец и вектора-строки на матрицу следующим образом:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{pmatrix},$$

$$\begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} xa_{11} + ya_{21} & xa_{12} + ya_{22} \end{pmatrix}.$$

Легко заметить, что для матрицы M и вектора-столбца v справедливо следующее соотношение:

$$(Mv)^T = v^T M^T .$$

Преобразования при помощи матриц

На самом деле каждая двумерная матрица задает некоторое линейное преобразование на плоскости.

Рассмотрим матрицу $M = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$ и единичный квадрат на плоскости *Oxy*.

Тогда, если каждую точку этого квадрата представить вектором-столбцом и умножить его на матрицу *M*, то мы получим новую точку на плоскости. Как несложно убедиться, полученное множество преобразованных точек будет представлять собой квадрат размером 2×2 (рис. П1.1).

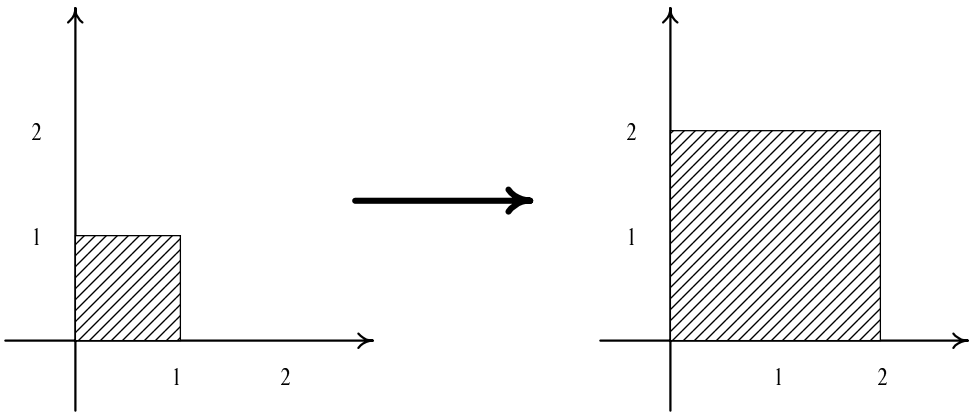


Рис. П1.1. Масштабирование в два раза

Таким образом, эта матрица задает преобразование растяжения (масштабирования) в два раза по каждой из координат.

В общем случае матрица $M = \begin{pmatrix} \lambda & 0 \\ 0 & \mu \end{pmatrix}$, где все элементы главной диагонали больше нуля, задает масштабирование в λ раз по оси *Ox* и в μ раз по оси *Oy*.

При помощи матрицы $M = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$ осуществляется отражение относительно оси *Oy* (рис. П1.2).

Аналогично матрица $M = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ задает отражение относительно оси *Ox*.

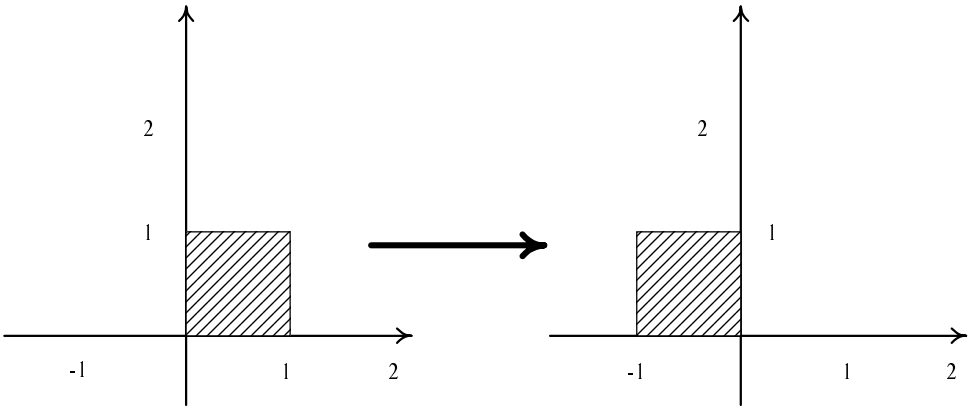


Рис. П1.2. Отражение

Для произвольного угла φ матрица $M = \begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix}$ задает поворот вокруг начала координат на угол φ по часовой стрелке (рис. П1.3).

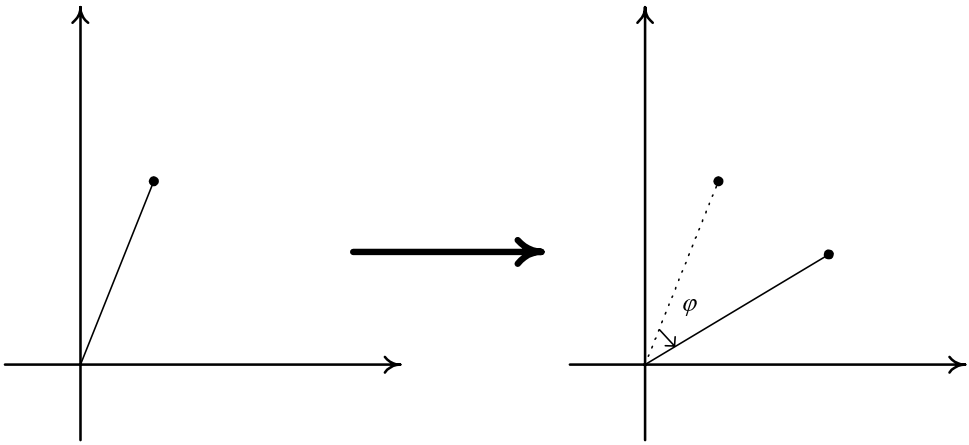


Рис. П1.3. Поворот

Можно убедиться в том, что композиции (последовательному применению) преобразований соответствует преобразование, матрица которого является произведением матриц соответствующих преобразований.

Трехмерные векторы и матрицы

Обобщим теперь рассмотренные ранее понятия на трехмерный случай: трехмерным вектором $v \in R^3$ будем называть упорядоченный набор из трех вещественных чисел x, y и z, записанный или в виде вектора-столбца, или в виде вектора-строки.

Операции умножения на число, сложения и вычитания вводятся полностью аналогично двумерному случаю, т. е. покомпонентно.

Скалярное произведение двух трехмерных векторов вводится следующим образом:

$$\left(\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}, \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix}\right) = x_1 x_2 + y_1 y_2 + z_1 z_2 .$$

Легко убедиться в выполнении всех свойств, ранее сформулированных для двумерных векторов и операций над ними. Понятие длины вектора и угла между трехмерными векторами вводится совершенно аналогично двумерному случаю.

Однако наряду со скалярным произведением для трехмерных векторов вводится также понятие векторного произведения (*cross product*). Векторное произведение двух трехмерных векторов u и v определяется следующим образом:

$$[u,\ v] = u \times v = \begin{pmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{pmatrix}.$$

Можно показать, что векторное произведение задает вектор, перпендикулярный обоим перемножаемым векторам, и удовлетворяет следующим свойствам:

- $u \times v = -v \times u$;
- $(\alpha u + \beta v) \times w = \alpha u \times w + \beta v \times w$;
- $(u \times v) \cdot w = (v \times w) \cdot u = (w \times u) \cdot v = -(v \times u) \cdot w = -(u \times w) \cdot v = -(w \times v) \cdot u$;
- $u \times (v \times w) = (u,\ w) \cdot v - (u,\ v) \cdot w$.

Трехмерная (3×3) матрица определяется как таблица из девяти вещественных чисел следующего вида:

$$M = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}.$$

Операции умножения на число, сложения и вычитания для трехмерных матриц также вводятся покомпонентно. Нулевой матрицей O называется матрица, состоящая из одних нулей, единичной — следующая матрица:

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Операция транспонирования также вводится как отражение относительно главной диагонали, т. е. преобразование, меняющее местами элементы a_{ij} и a_{ji} для всех пар различных индексов i и j.

Умножение трехмерных матриц вводится следующим образом (как можно заметить, отличие от двумерного случая заключается лишь в пределах суммирования):

$$C = A \cdot B,$$

где $c_{ij} = \sum_{k=1}^{3} a_{ik} b_{kj}$.

Также вводятся операции умножения трехмерной матрицы на трехмерный вектор-столбец и трехмерного вектора-строки на трехмерную матрицу:

$$M \cdot v = \begin{pmatrix} a_{11}x + a_{12}y + a_{13}z \\ a_{21}x + a_{22}y + a_{23}z \\ a_{31}x + a_{32}y + a_{33}z \end{pmatrix}.$$

Для введенных операций выполняются все свойства операций для двумерных матриц и векторов.

Каждая трехмерная матрица задает некоторое преобразование в трехмерном пространстве. Ниже приводятся матрицы для основных преобразований.

Матрица поворота вокруг оси Ox на угол φ имеет вид

$$R_x(\varphi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{pmatrix}.$$

Матрица поворота вокруг оси Oy на угол φ имеет вид

$$R_y(\varphi) = \begin{pmatrix} \cos\varphi & 0 & \sin\varphi \\ 0 & 1 & 0 \\ -\sin\varphi & 0 & \cos\varphi \end{pmatrix},$$

а матрица поворота вокруг оси Oz —

$$R_z(\varphi) = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Масштабирование вдоль координатных осей задается при помощи матрицы

$$S = \begin{pmatrix} \lambda & 0 & 0 \\ 0 & \mu & 0 \\ 0 & 0 & \nu \end{pmatrix},$$

где коэффициенты $\lambda,\ \mu,\ \nu > 0$.

Отражение относительно плоскости Oxy задается матрицей

$$R_{xy} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}.$$

Отражение относительно остальных координатных плоскостей задается совершенно аналогично.

Кроме преобразований, задаваемых при помощи матриц (их называют линейными), есть еще одно простое преобразование, которое не может быть представлено умножением на трехмерную матрицу — перенос на вектор a.

Однородные координаты и преобразования

Все преобразования (обычных и текстурных координат, а также нормалей) и в OpenGL, и в Direct3D осуществляются при помощи так называемых однородных координат.

Однородные координаты представляют собой простое и удобное расширение обычных трехмерных координат, позволяющее представлять при помощи матриц широкий класс преобразований пространства, включая перспективное проектирование.

Сопоставим произвольному трехмерному вектору $v = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ семейство четырехмерных векторов вида $h = \begin{pmatrix} xw \\ yw \\ zw \\ z \end{pmatrix}$, где w — произвольный ненулевой числовой множитель. Такие 4-мерные векторы мы и будем называть однородными координатами.

Простейшим способом сопоставления трехмерному вектору однородных координат является добавление четвертой координаты, равной единице (фактически просто берется $w = 1$).

Обратное преобразование несколько сложнее. Для получения по однородным координатам соответствующего трехмерного вектора необходимо разделить все компоненты 4-мерного вектора на его последний элемент. Первые три компонента результата и дадут требуемый трехмерный вектор.

Иногда, для облегчения записи, однородные координаты, соответствующие трехмерному вектору v, записывают как $\begin{pmatrix} v \\ 1 \end{pmatrix}$.

По аналогии с трехмерными матрицами можно ввести матрицу 4×4 как таблицу из шестнадцати вещественных чисел. Если ее нижний правый элемент (a_{44}) равен единице, то такую матрицу называют однородной.

Однородную матрицу можно записать в следующем блочном виде:

$$H = \begin{pmatrix} M & a \\ b^T & 1 \end{pmatrix}.$$

Здесь M — трехмерная матрица, a и b — трехмерные векторы-столбцы.

Рассмотрим, какие преобразования задают однородные матрицы.

Пусть $b = 0$.

Тогда матрица $H = \begin{pmatrix} M & 0 \\ 0^T & 1 \end{pmatrix}$ просто задает то же самое преобразование в однородных координатах, что и матрица M в трехмерном пространстве, поскольку

$$\begin{pmatrix} M & 0 \\ 0^T & 1 \end{pmatrix} \cdot \begin{pmatrix} v \\ 1 \end{pmatrix} = \begin{pmatrix} M \cdot v \\ 1 \end{pmatrix}.$$

Аналогично матрица $H = \begin{pmatrix} I & a \\ 0^T & 1 \end{pmatrix}$ задает перенос на вектор a, поскольку

$$\begin{pmatrix} I & a \\ 0^T & 1 \end{pmatrix} \cdot \begin{pmatrix} v \\ 1 \end{pmatrix} = \begin{pmatrix} v + a \\ 1 \end{pmatrix}.$$

Однородные матрицы, для которых вектор b отличен от нулевого, задают так называемые дробно-линейные преобразования. В частности, с помощью таких матриц задается перспективное преобразование.

Как обычно, суперпозиции преобразований отвечает произведение соответствующих однородных матриц.

В результате все преобразования вершин в конвейере рендеринга могут быть представлены при помощи соответствующих однородных матриц, а значит, для преобразования вершины достаточно ее умножить на однородную матрицу, являющуюся произведением однородных матриц промежуточных преобразований.

Системы координат и переходы между ними

Под системой координат в трехмерном пространстве понимается некоторая точка *org*, задающая точку отсчета (начало системы координат) и три вектора e_1, e_2 и e_3, не лежащие в одной плоскости (рис. П1.4).

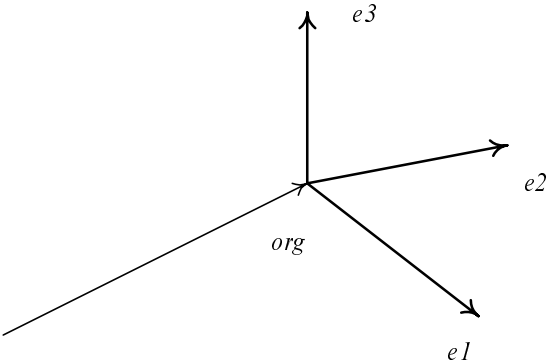


Рис. П1.4. Система координат в трехмерном пространстве

Чаще всего мы имеем дело с так называемыми прямоугольными системами координат, когда все три вектора e_1, e_2 и e_3 попарно перпендикулярны друг другу. В этом случае удобно также считать, что все они являются единичными.

Пусть теперь у нас задана точка p и мы хотим перевести ее в систему координат (org, e_1, e_2, e_3). Тогда ее координатами в этой системе мы будем называть такую тройку чисел p_x, p_y и p_z, что выполняется равенство:

$$p = org + p_x \cdot e_1 + p_y \cdot e_2 + p_z \cdot e_3 .$$

Если записать новые координаты точки как вектор-столбец $p' = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$ и построить матрицу R, используя векторы e_1, e_2 и e_3 в качестве столбцов, то последнее равенство можно переписать в виде:

$$p = org + R \cdot p'.$$

Из этого соотношения с помощью обратной матрицы легко найти искомые координаты p':

$$p' = R^{-1} \cdot (p - org).$$

Однако в случае, когда векторы e_1, e_2 и e_3 попарно перпендикулярны и имеют единичную длину, справедливо следующее свойство: $R^{-1} = R^T$. Фактически матрица R является матрицей поворота вокруг некоторой прямой.

Считая, что условия попарной перпендикулярности и единичной длины векторов e_1, e_2 и e_3 выполнены, формулу перехода в новую систему координат можно переписать при помощи однородной матрицы:

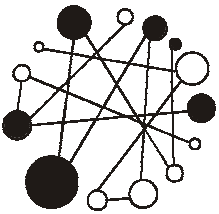
$$p' = \begin{pmatrix} R^T & -org \\ 0^T & 1 \end{pmatrix} \cdot p.$$

Таким образом, для перехода от одной системы координат к другой достаточно умножить старые координаты на матрицу перехода.

Обратите внимание, что не только точка p, но и все векторы org, e_1, e_2 и e_3 задаются в исходной системе координат (из которой мы переводим). Поэтому для перехода в новую систему координат необходимо выразить начало новой системы координат org и направления координатных осей e_1, e_2 и e_3 в текущей системе координат, после чего построить однородную матрицу H для перехода в новую систему координат.

При этом матрица H^{-1} будет осуществлять перевод из новой системы координат в старую.

Приложение 2



Основные модели освещения

Диффузная модель освещения

Простейшей моделью освещения является диффузная или модель Ламберта.

Согласно этой модели падающий в точку свет равномерно рассеивается во все стороны, соответствующие положительному полупространству. Положение наблюдателя никакой роли не играет (рис. П2.1).

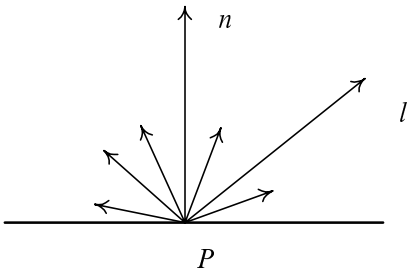


Рис. П2.1. Диффузная модель освещения

При этом видимая наблюдателем освещенность точки определяется только плотностью потока световой энергии, падающей в эту точку. Эта плотность, в свою очередь, определяется как интенсивностью источника света, так и ориентацией самой поверхности в точке.

Рассмотрим произвольный луч света с направляющим вектором *l* и определим, как влияет ориентация освещаемой этим лучом площадки на ее удельную освещенность (рис. П2.2).

Ясно, что для произвольной ориентации площадки общая световая энергия, падающая на нее, постоянна (и определяется мощностью луча). Однако площадь, на которую падает эта световая энергия, зависит от ориентации площадки, а значит, от нее зависит и удельная освещенность в точке.

Из курса геометрии известно, что эта зависимость определяется косинусом угла между нормалью *n* к площадке и вектором *l*. Если считать оба этих век-

тора единичными, то удельная освещенность будет прямо пропорциональна скалярному произведению (n, l).

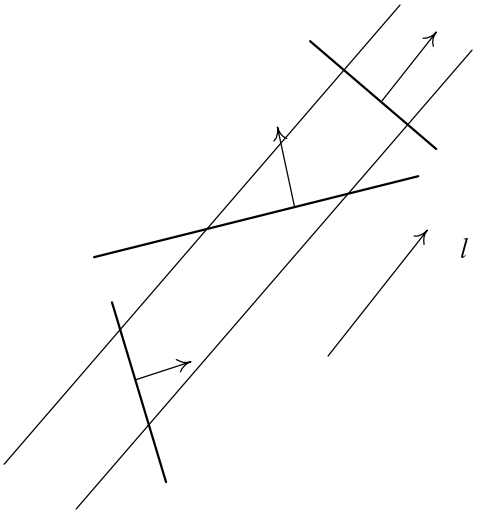


Рис. П2.2. Зависимость плотности световой энергии от ориентации поверхности относительно источника света

Таким образом, видимая наблюдателем освещенность в точке будет описываться следующей формулой:

$$I = C \cdot (n, l) \cdot I_l. \qquad (П2.1)$$

Здесь C — цвет поверхности в точке, а I_l — интенсивность самого источника света.

Обратите внимание, в (П2.1) предполагается, что источник света лежит в том полупространстве, куда показывает вектор нормали. Если это не так, то эта формула даст отрицательное значение освещенности в точке, что неверно. В этом случае освещенность должна равняться нулю (здесь мы исключаем случай преломления).

Еще одним недостатком данной формулы является то, что при ее использовании отдельные части объектов оказываются полностью черными, что в реальности встречается крайне редко (обычно практически всегда присутствует некоторое рассеянное (фоновое) освещение). Для его учета в формулу (П1.1) обычно вводят новый член, отвечающий за фоновую освещенность.

С учетом всех изменений диффузная модель освещенности принимает следующий вид.

$$I = k_a \cdot C \cdot I_a + k_d \cdot C \cdot \max\big((n,\ l), 0\big) \cdot I_l\ , \qquad (П2.2)$$

где I_a — интенсивность фонового (рассеянного) освещения, а k_a и k_d — коэффициенты, определяющие вклад фонового и диффузного освещения в итоговую освещенность точки.

Модель освещения Блинна

Одним из недостатков диффузной модели является полное отсутствие каких-либо бликов на поверхностях объектов, столь часто встречающихся в реальной жизни.

Одной из моделей освещенности, учитывающих блики, является модель Блинна (рис. П2.3). В ней вводится специальный член — $\max\left((n,\ h),0\right)^p$.

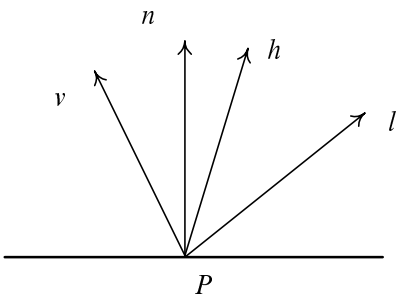


Рис. П2.3. Модель освещения Блинна

Здесь h — средний вектор между направлением на источник света l и наблюдателя v:

$$h = \frac{l + v}{|l + v|}.$$

Степень p характеризует резкость бликов, чем она выше, тем блик меньше и резче.

С учетом этого модель освещенности принимает следующий вид:

$$I = k_a \cdot C \cdot I_a + k_d \cdot C \cdot \max\left((n,\ l),0\right) \cdot I_l + k_s \cdot C_s \cdot \max\left((n,\ h),0\right)^p \cdot I_l, \quad (\text{П}.2.3)$$

где коэффициент k_s отвечает за вклад блика в итоговую освещенность, а C_s определяет цвет блика.

Модель освещения Фонга

Еще одной моделью освещения, учитывающей блики, является модель Фонга. В ней для учета блика также используется скалярное произведение в степени:

$$I = k_a \cdot C \cdot I_a + k_d \cdot C \cdot \max\left((n,\ l),0\right) \cdot I_l + k_s \cdot C_s \cdot \max\left((l,\ r),0\right)^p \cdot I_l, \quad (\text{П2.4})$$

где коэффициент k_s отвечает за вклад блика в итоговую освещенность, C_s определяет цвет блика, а r — это отраженный вектор направления на наблюдателя.

Считая все исходные векторы единичными, отраженный вектор задается следующей формулой:

$$r = 2 \cdot n \cdot (n, v) - v.$$

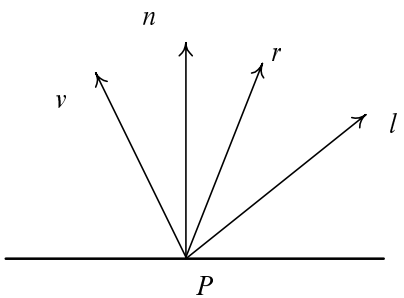


Рис. П2.4. Модель освещения Фонга

Анизотропная модель

Еще одной интересной моделью освещенности является анизотропная модель освещения. В ней свойства поверхности в каждой точке характеризует касательный вектор t, т. е. на поверхности задано поле касательных векторов.

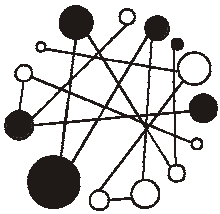
Наиболее распространенной формулой вычисления освещенности в этом случае является следующая:

$$I = k_a \cdot C \cdot I_a + k_d \cdot C \cdot \left(1 - (t,\ l)^2\right)^{p_1} \cdot I_l + k_s \cdot C_s \cdot \left(1 - (t,\ h)^2\right)^{p_2} \cdot I_l. \quad (\text{П2.5})$$

Здесь векторы l и h имеют тот же смысл, что и ранее, а t — это касательный вектор в точке.

Показатели степени p_1 и p_2 задают диффузную и бликовую составляющие.

Приложение 3



Описание содержимого компакт-диска

Папка	Описание	Глава
Bin\Chapter-*NN*	Исполняемые файлы примеров к главе *NN* *, откомпилированные под Windows	*NN*
\Code\	Файлы глобальных параметров *.inc, *make*-файлы и файлы, задающие параметры компиляции rules.linux и rules.win32	1
\Code\3D	Ряд полезных классов (реализации векторов и матриц как классов, ограничивающего параллелепипеда, плоскости)	13
\Code\Chapter-*NN*	Исходные файлы примеров к главе *NN*	*NN*
\Code\glut	Исходные файлы для библиотеки GLUT	1
\Code\libExt	Полный исходный код для библиотеки libExt	1
\Code\libTexture	Исходные файлы для библиотеки libTexture	1, 2, 6, 8
\Code\PBuffer	Реализации класса `PBuffer` (Windows- и Linux-версия)	11
\Code\Program	Вспомогательные файлы для заворачивания вершинных, фрагментных и GLSL-программ в классы C++	15–18
\Code\scripts	Вспомогательные файлы программ на языке Python	18
\Code\Textures	Файлы текстур для различных примеров	8, 14, 18
\Code\utils	Вспомогательные файлы для отдельных примеров	6

* *NN* — номер соответствующей главы.

Перечень рекомендованной литературы и источников в Интернете

Литература

1. Боресков А. В. Графика трехмерной компьютерной игры на основе OPENGL. — М.: Диалог-МИФИ, 2004.
2. Гайдуков С. А. OpenGL. Профессиональное программирование трехмерной графики на C++. — СПб.: БХВ-Петербург, 2004.
3. Шикин Е. В., Боресков А. В. Компьютерная графика. Полигональные модели. — М.: Диалог-МИФИ, 2000.
4. М. Ву. OpenGL. Официальное руководство программиста. — СПб.: ДиаСофтЮП, 2002.
5. OpenGL. Официальный справочник. — СПб.: ДиаСофтЮП, 2002.
6. Ф. Хилл. OpenGL. Программирование компьютерной графики. — СПб.: Питер, 2002.
7. Тихомиров Ю. В. Программирование трехмерной графики. — СПб.: БХВ—Санкт-Петербург, 1998.
8. Краснов М. В. OpenGL. Графика в проектах Delphi. — СПб.: БХВ—Санкт-Петербург, 2000.
9. Э. Эйнджел. Интерактивная компьютерная графика. Вводный курс на базе OpenGL. — М.: Вильямс, 2001.
10. Роджерс Д. Математические основы машинной графики. — М.: Мир, 2001.
11. Ким Г. Д., Ильин В. А. Линейная алгебра и аналитическая геометрия. — М.: Изд-во МГУ, 1998.
12. Никулин Е. А. Компьютерная геометрия и алгоритмы машинной графики. — СПб.: БХВ-Петербург, 2003.

Ресурсы в Интернете

1. **http://www.opengl.org**
2. **http://developer.nvidia.com**
3. **http://www.ati.com/developer**
4. **http://www.steps3d.narod.ru**
5. **http://www.flipcode.com**
6. **http://www.gamedev.ru**
7. **http://www.gamedev.net**
8. **http://nehe.gamedev.net**
9. **http://www.frustum.org**
10. **http://www.shadertech.com**
11. **http://www.clockworkcoders.com/oglsl/tutorials.html**
12. **http://www.3dshaders.com**
13. **http://www.humus.ca**
14. **http://www.paulsprojects.net**

Предметный указатель

(окончание рубрики см. на с. 672)

Р

С

Т

Ф

Ш

Э

Я