
Учебное пособие по OpenGL

Версия 1.0

Дигия, Обучение Qt

28 февраля 2013 г.

Содержание

1 Об этом уроке 1
1.1 Почему вам стоит прочитать это руководство? 1
1.2. Получите исходный код и руководство в разных форматах. 1
1.3 Лицензия. 2

2 Введение 3
2.1 Что такое OpenGL 3
2.2 Рисование в 3D-пространстве. 4
2.3. Краткий обзор линейной алгебры. 5
2.4 Системы координат и концепция системы координат. 8
2.5 Конвейер рендеринга OpenGL. 10
2.6 API OpenGL. 12
2.7 Язык шейдеров OpenGL. 13

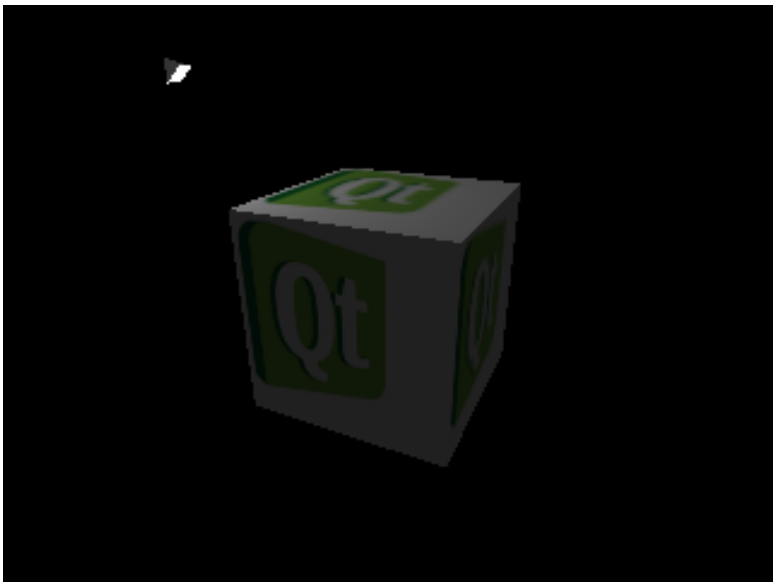
3 Использование OpenGL в вашем приложении Qt 16
3.1 Привет OpenGL. 16
3.2 Рендеринг в 3D. 22
3.3 Раскраска. 26
3.4 Наложение текстур. 28
3.5 Освещение. 32
3.6 Буферный объект. 39

4. Заключение и дополнительная литература 43

Об этом уроке

1.1 Почему вам стоит прочитать это руководство?

Это руководство представляет собой базовое введение в OpenGL и компьютерную 3D-графику. В нем показано, как использовать Qt и связанные с ним классы OpenGL для создания трехмерной графики с использованием программируемого конвейера OpenGL. В руководстве представлено множество примеров, демонстрирующих основные функции программирования OpenGL, такие как рендеринг, наложение текстур, освещение и т. д. К концу руководства вы получите хорошее представление о том, как работает OpenGL, а также сможете писать собственные программы шейдеров.



1.2. Получите исходный код и руководство в разных форматах.

Предоставляется ZIP-файл, содержащий полный исходный код примеров руководства:

Исходный код[1]

Руководство доступно в следующих форматах:

PDF[2]

ePub[3]для читателей электронных книг. Более подробную информацию можно найтиздесь[4].

Qt-справка[5]для Qt Assistant и Qt Creator. В Qt Assistant вДиалог настроек[6]под*Документация*(в раскрывающемся меню для пользователей Mac) нажмите кнопку*Добавлять*кнопка, чтобы добавить это руководство в формате .qch. Мы делаем то же самое в Qt Creator под *Параметры*диалог в*Помощь*раздел. Здесь вы можете добавить это руководство в *Документация*вкладка.

1.3 Лицензия

Что дальше?

Далее будет знакомство с*OpenGL*и основы рисования в 3D.

1 http://releases.qt-project.org/learning/developerguides/qtopengltutorial/opengl_src.zip
2 http://releases.qt-project.org/learning/developerguides/qtopengltutorial/OpenGLTutorial.pdf
3 http://releases.qt-project.org/learning/developerguides/qtopengltutorial/OpenGLTutorial.epub
4 http://en.wikipedia.org/wiki/EPUB#Software_reading_systems
5 http://releases.qt-project.org/learning/developerguides/qtopengltutorial/OpenGLTutorial.qch
6 http://qt-project.org/doc/qt-4.8/assistant-details.html#preferences-dialog

ГЛАВА2

Введение

Это руководство представляет собой базовое введение в OpenGL и компьютерную 3D-графику. Он показывает, как использовать Qt и связанные с ним классы OpenGL для создания 3D-графики.

Мы будем использовать основные функции OpenGL 3.0/2.0 ES и всех последующих версий, а это означает, что мы будем использовать программируемый конвейер рендеринга OpenGL для написания наших собственных шейдеров с помощью языка шейдеров OpenGL (GLSL)/языка шейдеров OpenGL ES (GLSL/ES).).

В первой главе дается введение в компьютерную 3D-графику и API OpenGL, включая язык шейдеров OpenGL (GLSL)/язык шейдеров OpenGL ES (GLSL/ES). Если вы уже знакомы с этой темой и хотите только узнать, как использовать OpenGL в своих программах Qt, вы можете пропустить эту вводную главу и перейти ко второй главе.

Во второй главе мы представляем примеры, в которых используется информация, представленная в первой главе, и показываем, как использовать OpenGL вместе с функциональностью Qt, связанной с OpenGL.

В конце этого руководства вы найдете некоторые ссылки и ссылки, которые могут пригодиться, особенно при работе с примерами. Обратите внимание, что это руководство предназначено для того, чтобы вы начали изучать эту тему, и не может вдаваться в ту же глубину, что и приличная книга, посвященная OpenGL. Также обратите внимание, что классы Qt, связанные с OpenGL, облегчают вашу жизнь, скрывая некоторые детали, с которыми вы могли бы столкнуться, если бы писали свои программы, используя только OpenGL API.

В части примеров мы будем использовать высокоуровневую функциональность Qt, когда это возможно, и лишь кратко назовем различия. Поэтому, если вы хотите получить полное представление о том, как использовать нативный код, вам следует дополнительно обратиться к учебнику или книге, посвященной этой теме.

2.1 Что такое OpenGL

OpenGL — наиболее широко используемый в отрасли API для 2D- и 3D-графики. Он управляется некоммерческим технологическим консорциумом Khronos Group, Inc. Это легко переносимая, масштабируемая, кросс-языковая и кросс-платформенная спецификация, определяющая единый интерфейс для графического ускорителя компьютера. Он гарантирует набор базовых возможностей и позволяет поставщикам реализовывать свои собственные расширения.

OpenGL — это низкоуровневый API, который требует от программиста сообщить ему точные шаги, необходимые для рендеринга сцены. Вы не можете просто описать сцену и отобразить ее на мониторе. Вам предстоит указать геометрические примитивы в трехмерном пространстве, применить эффекты цвета и освещения, а также визуализировать объекты на экране. Хотя это требует некоторых знаний компьютерной графики, это также дает вам большую свободу изобретать собственные алгоритмы и создавать множество новых графических эффектов.

Единственная цель OpenGL — рендеринг компьютерной графики. Он не предоставляет никаких функций для управления окнами или обработки таких событий, как ввод пользователя. Для этого мы используем Qt.

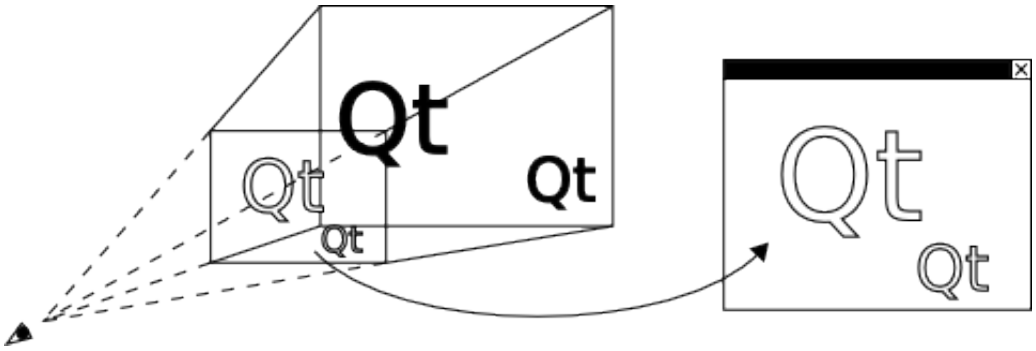
2.2 Рисование в 3D-пространстве

Геометрия трехмерных объектов описывается расположением самых простых строительных блоков (примитивов), таких как отдельные точки, линии или треугольники. Треугольники являются наиболее распространенными, поскольку они используются для аппроксимации поверхности объектов. Каждую поверхность можно разделить на небольшие плоские треугольники. Хотя это хорошо работает с объектами с краями, но гладкие объекты, такие как сферы, будут выглядеть неровными. Конечно, вы можете использовать больше треугольников, чтобы улучшить аппроксимацию, но это происходит за счет производительности, поскольку вашей видеокарте придется обрабатывать больше треугольников. Вместо простого увеличения количества полигонов всегда следует учитывать дополнительные методы, такие как улучшение алгоритма освещения или адаптация уровня детализации.

Чтобы определить пространственные свойства ваших объектов, вы создаете список точек, линий и/или треугольников. Каждый примитив, в свою очередь, определяется положением его углов (вершины/вершин). Таким образом, необходимо иметь базовое представление о том, как определять точки в пространстве и эффективно манипулировать ими. Но через мгновение мы освежим наши знания по линейной алгебре.

Чтобы увидеть объекты, вы должны применить окраску к своим примитивам. Значения цвета часто определяются для каждого примитива (точнее, для каждой вершины) и используются для рисования или заполнения цветом. В более реалистичных приложениях изображения (называемые текстурами) размещаются поверх объектов. Внешний вид можно дополнительно адаптировать в зависимости от свойств материала или освещения. Итак, как нам на самом деле отобразить нашу сцену на экране?

Поскольку экран компьютера является двухмерным устройством, нам необходимо проецировать объекты на плоскость. Затем эта плоскость сопоставляется с областью на нашем экране, называемой окном просмотра*. Чтобы наглядно понять этот процесс, представьте, что вы стоите перед окном и рисуете на стекле очертания предметов, которые видите снаружи, не двигая головой. Рисунок на стекле представляет собой двухмерную проекцию окружающей среды. Хотя техника другая, подобная проекция также происходит в камере при съемке.

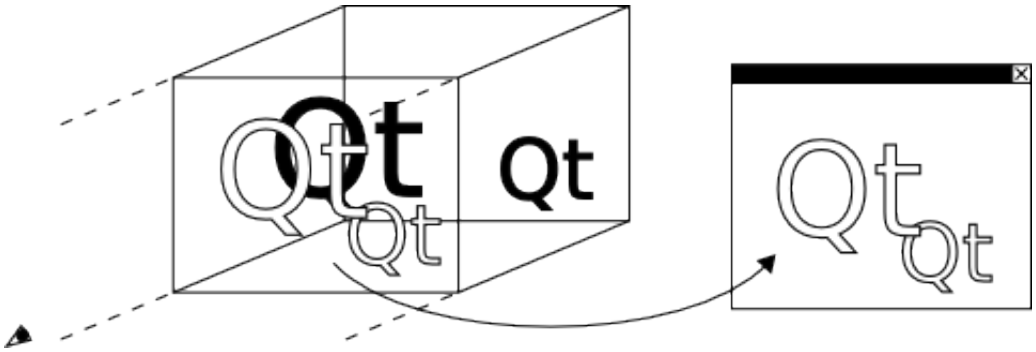


Обрезанная пирамида называется *объем просмотра*. Все, что находится внутри этого объема, проецируется на ближнюю сторону в сторону воображаемого зрителя. Все остальное не нарисовано.

Существует два основных типа проекций: перспективные и ортогональные.

То, что мы только что представили, называется *перспективная проекция*. Он имеет обзорный объем в форме усеченной пирамиды и добавляет иллюзию того, что удаленные объекты кажутся меньше, чем более близкие объекты того же размера. Это в значительной степени способствует реализму и поэтому используется для моделирования, игр и приложений VR (виртуальной реальности).

Другой тип называется *ортографическая проекция*. Ортогональные проекции задаются прямоугольным обзорным объемом. Любые два объекта одинакового размера также имеют одинаковый размер в проекции независимо от их расстояния от зрителя. Это часто используется в инструментах САПР (компьютерного проектирования) или при работе с 2D-графикой.



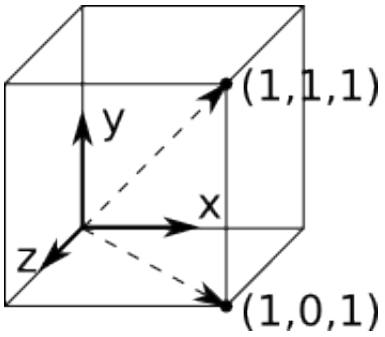
2.3. Краткий обзор линейной алгебры

Поскольку при написании программ OpenGL важно иметь базовое понимание линейной алгебры, в этой главе кратко излагаются наиболее важные концепции. Хотя в основном мы позволяем Qt выполнять математические операции, все равно полезно знать, что происходит в фоновом режиме.

Местоположение трехмерной точки относительно произвольной системы координат определяется ее координатами x, y и z. Этот набор значений также называется *вектор*. Когда он используется для описания примитивов, он называется *вершина*.

$$v = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

Затем объект представляется списком вершин.



Часто вам захочется изменить положение, размер или ориентацию вашего объекта.

Перевести объект так же просто, как добавить постоянный вектор (здесь он называется *d*), который определяет смещение всех вершин ваших объектов (здесь они называются *v*).

$$v_{new} = v_{old} + d = \begin{pmatrix} v_{old,x} \\ v_{old,y} \\ v_{old,z} \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} = \begin{pmatrix} v_{old,x} + d_x \\ v_{old,y} + d_y \\ v_{old,z} + d_z \end{pmatrix}$$

Масштабирование означает умножение вершин на желаемое соотношение (здесь оно называется *c*).

$$v_{new} = s \cdot v_{old} = \begin{pmatrix} s \cdot v_{old,x} \\ s \cdot v_{old,y} \\ s \cdot v_{old,z} \end{pmatrix}$$

Вращение, растяжение, сдвиг или отражение более сложны и достигаются путем умножения вершин на матрицу преобразования (здесь называемую *T*). Матрица — это, по сути, таблица коэффициентов, которые умножаются на вектор, чтобы получить новый вектор, где каждый элемент представляет собой линейную комбинацию элементов умноженного вектора.

$$T = \begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix}$$

$$v_{new} = T \cdot v_{old} = \begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} \cdot \begin{pmatrix} v_{old,x} \\ v_{old,y} \\ v_{old,z} \end{pmatrix} = \begin{pmatrix} t_{11} \cdot v_{old,x} + t_{12} \cdot v_{old,y} + t_{13} \cdot v_{old,z} \\ t_{21} \cdot v_{old,x} + t_{22} \cdot v_{old,y} + t_{23} \cdot v_{old,z} \\ t_{31} \cdot v_{old,x} + t_{32} \cdot v_{old,y} + t_{33} \cdot v_{old,z} \end{pmatrix}$$

Например, это матрицы, вращающие вектор вокруг осей x, y и z системы координат. Произвольные ротации могут быть составлены путем их комбинации.

$$T_{rot,x}(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & -sin(\alpha) \\ 0 & sin(\alpha) & cos(\alpha) \end{pmatrix}$$

$$T_{rot,y}(\alpha) = \begin{pmatrix} cos(\alpha) & 0 & sin(\alpha) \\ 0 & 1 & 0 \\ -sin(\alpha) & 0 & cos(\alpha) \end{pmatrix}$$

$$T_{rot,z}(\alpha) = \begin{pmatrix} cos(\alpha) & -sin(\alpha) & 0 \\ sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Есть также одна матрица, которая вообще не меняет вектор. Это называется*единичная матрица*и состоит из единиц на главной диагонали и нулей в остальных местах.

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Если вы используете матрицу для преобразования вектора, важно, чтобы матрица была записана слева от знака умножения, а вектор — справа. Кроме того, количество столбцов матрицы должно совпадать с количеством компонентов вектора. В противном случае умножение математически неверно, и математические библиотеки могут возвращать неожиданные результаты.

Имейте в виду, что преобразования не коммутативны, т.е. результат конкатенации преобразований зависит от их порядка. Например, имеет значение, вращаете ли вы сначала объект, а затем перемещаете его, или делаете наоборот.

Поскольку удобнее (и даже быстрее для OpenGL) выразить все эти операции как одно умножение матрицы на вектор, мы расширим наш формализм до так называемого*однородные координаты*. Это также позволяет нам легко применять все виды*аффинные преобразования*такие как прогнозы, которые мы обсуждали в главе 1.2. По сути, мы добавляем четвертое измерение, называемое*коэффициент масштабирования*, к нашим вершинам. Это может показаться усложняющим ситуацию, но на самом деле вам не нужно обращать внимание на этот фактор, поскольку по умолчанию он установлен на 1, и вы редко будете менять его самостоятельно. Все, что вам нужно сделать, это объявить свои вершины с дополнительным элементом, имеющим значение 1 (что даже часто подразумевается по умолчанию). (В этой главе мы обозначаем однородные координаты шляпкой на именах переменных.)

$$v = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \rightarrow \hat{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ v_w \end{pmatrix}$$

Тогда преобразование можно записать следующим образом:

$$\hat{v}_{new} = \hat{T} \cdot \hat{v}_{old} = \begin{pmatrix} T & d \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} v_{old} \\ 1 \end{pmatrix} = \begin{pmatrix} t_{11} & t_{12} & t_{13} & d_x \\ t_{21} & t_{22} & t_{23} & d_y \\ t_{31} & t_{32} & t_{33} & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} v_{old,x} \\ v_{old,y} \\ v_{old,z} \\ 1 \end{pmatrix} =$$

$$\begin{pmatrix} v_{new,x} \\ v_{new,y} \\ v_{new,z} \\ 1 \end{pmatrix}$$

Серию преобразований можно записать как серию умножений матриц, а полученное преобразование можно сохранить в одной матрице.

$$\hat{v}_{new} = \hat{T}_3 \cdot \hat{T}_2 \cdot \hat{T}_1 \cdot \hat{v}_{old} = \hat{T}_{resulting} \cdot \hat{v}_{old}$$

2.4 Системы координат и концепция системы координат

Как мы можем использовать знания линейной алгебры, чтобы вывести на экран трехмерную сцену? В этом уроке мы будем использовать наиболее широко используемую концепцию, называемую*концепция рамы*. Этот шаблон позволяет нам легко управлять объектами и зрителями (включая их положение и ориентацию), а также проекцией, которую мы хотим применить.

Представьте себе две системы координат:*А*и*Б*. Система координат*Б*происходит из системы координат*А*посредством перевода и вращения, которое можно описать следующей матрицей:

$$T$$

Тогда для каждой точки, определяемой как

$$p_B$$

в системе координат*Б*, соответствующие координаты точки

$$p_A = T \cdot p_B$$

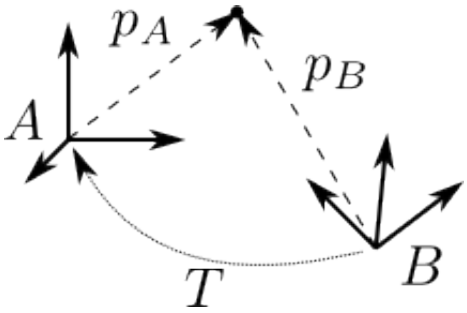
можно рассчитать,

$$p_A$$

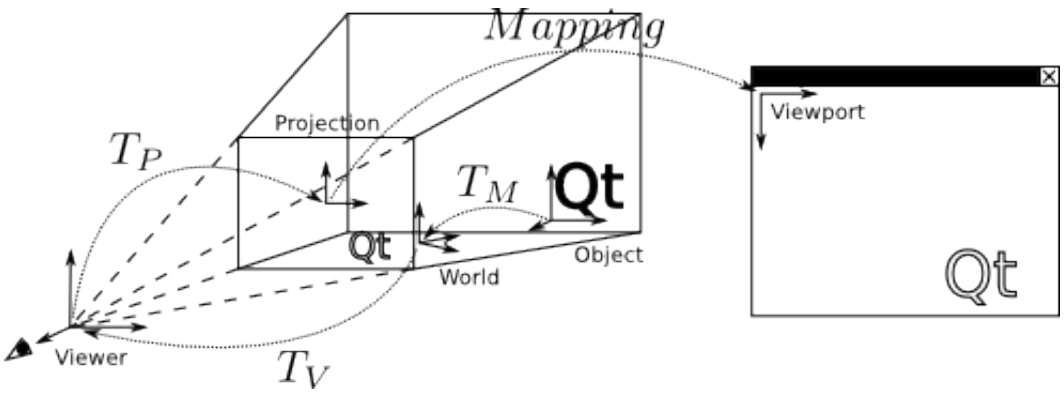
и

$$p_B$$

представляют одну и ту же точку в пространстве, но записаны по-разному.



Что касается концепции фрейма, каждый экземпляр объекта привязан к своей собственной системе координат (также называемой его системой координат).*рамка*). Положение и ориентация каждого объекта затем определяются путем размещения фреймов объектов внутри фрейма мира. То же самое относится и к зрителю (или*камера*) с одной разницей: для простоты мы на самом деле не помещаем кадр зрителя внутрь кадра мира, а делаем наоборот (т.е. помещаем кадр мира внутри кадра зрителя).

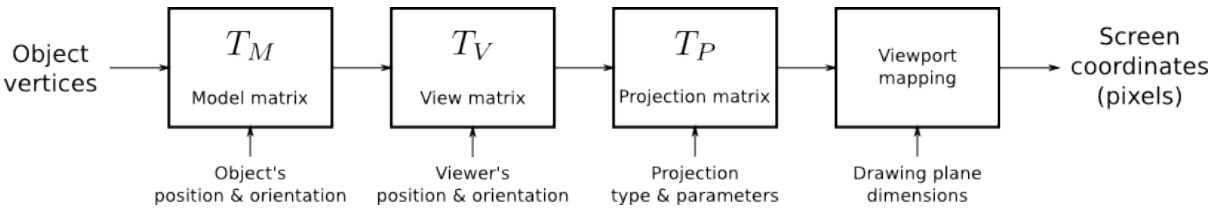


Это означает, что мы определяем положение и вращение каждого экземпляра объекта относительно мировой системы координат. Матрица, определяемая этими параметрами, которая позволяет нам вычислять вершины объекта внутри мировой системы координат, обычно называется *матрица модели*. Впоследствии мы переходим от мировых координат к координатам зрителя (обычно называемым *координаты глаза*) с использованием матрицы, называемой*просмотреть матрицу*точно так же. После этого мы применяем проекцию, которая преобразует вершины объекта из координат зрителя в плоскость проекции. Это делается с помощью матрицы, называемой*матрица проекции*, что дает нормализованные координаты устройства со значениями x, y и z в диапазоне от -1 до +1 (значения -1 и +1 соответствуют позициям на границах объема просмотра). Затем OpenGL сопоставляет все точки объекта на этой плоскости проекции с окном просмотра, отображаемым на экране.

Еще одна часто используемая матрица — это*матрица модель-вид-проекция*. Это объединение вышеупомянутых трех матриц.*матрица модель-вид-проекция*обычно передается в*вершинный шейдер*, который умножает эту матрицу на вершины объекта, чтобы вычислить проецируемую форму. Вы узнаете о шейдерах в следующей главе.

Определение этих матриц имеет различные преимущества:

- На этапе проектирования модель каждого объекта (т.е. его набор вершин) может быть указана относительно произвольной системы координат (например, его центральной точки).
- Процесс трансформации разделен на небольшие шаги, которые весьма показательны.
- Все используемые матрицы преобразования могут быть рассчитаны, сохранены и эффективно объединены.



На рисунке выше показаны шаги, необходимые для получения правильных экранных координат из вершин объекта. Различные виды преобразований применяются в определенном порядке. Вы добавляете несколько вершин объекта и после некоторой обработки чисел получаете соответствующие координаты экрана. На этом рисунке вы также можете легко понять, почему эта часть 3D-программирования называется *конвейер трансформации*.

2.5 Конвейер рендеринга OpenGL

Конвейер рендеринга OpenGL — это модель высокого уровня, описывающая основные шаги, которые OpenGL выполняет для рендеринга изображения на экране. Как слово *трубопровод* предполагает, что все операции применяются в определенном порядке. То есть конвейер рендеринга имеет состояние, которое принимает некоторые входные данные и возвращает изображение на экран.

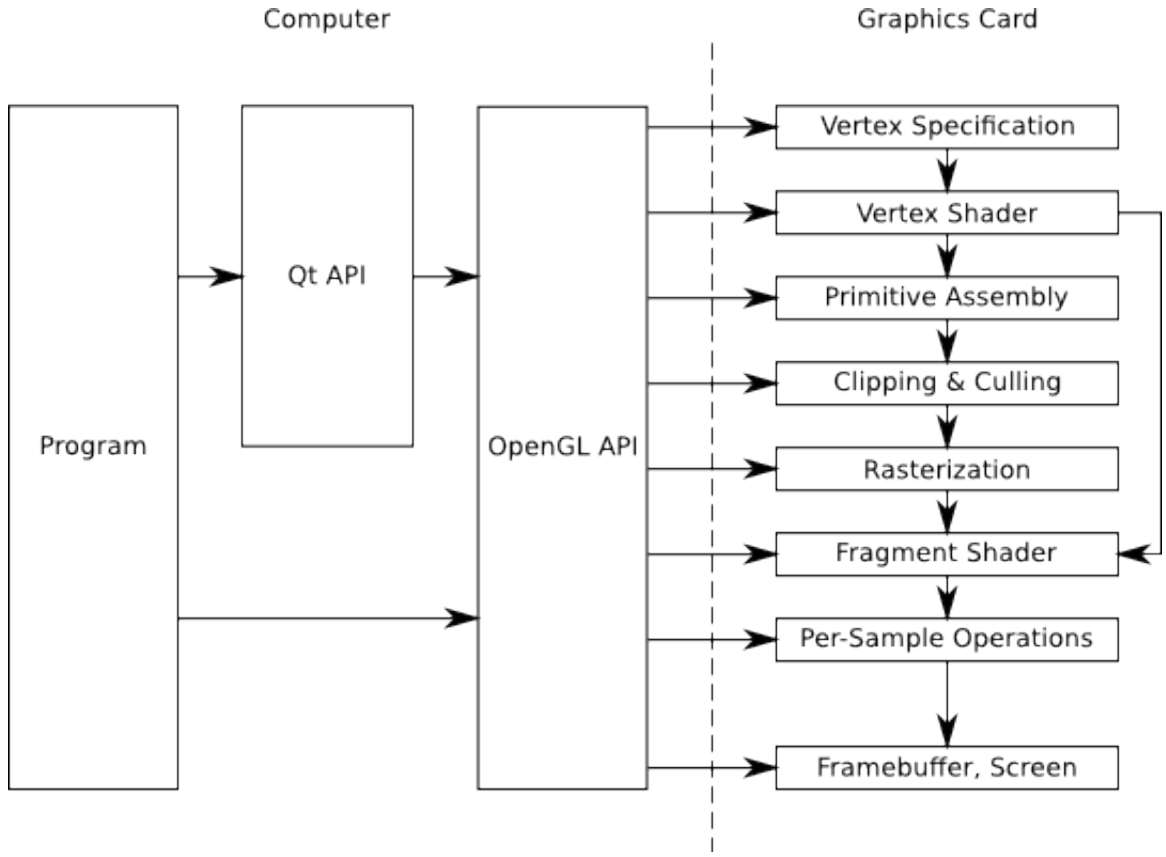
Состояние конвейера рендеринга влияет на поведение его функций. Поскольку нецелесообразно устанавливать параметры каждый раз, когда мы хотим что-то нарисовать, мы можем установить параметры заранее. Эти параметры затем используются во всех последующих вызовах функций. Например, после того как вы определили цвет фона, этот цвет будет использоваться для очистки экрана, пока вы не измените его на что-то другое. Вы также можете включать и выключать отдельные функции, такие как тестирование глубины или мультисэмплирование. Таким образом, чтобы нарисовать наложенное изображение поверх экрана, вы должны сначала нарисовать сцену с включенным тестированием глубины, затем отключить тестирование глубины и нарисовать элементы наложения, которые затем всегда будут отображаться поверх экрана, независимо от их расстояния от экрана. зритель.

Входные данные для конвейера могут предоставляться в виде отдельных значений или массивов. В большинстве случаев эти значения будут представлять позиции вершин, нормали поверхности, текстуры, координаты текстуры или значения цвета.

Выходными данными конвейера рендеринга является изображение, отображаемое на экране или записываемое в память. Такой сегмент памяти тогда называется кадровым буфером.

На рисунке ниже показана упрощенная версия конвейера. Элементы, не относящиеся к этому руководству, были опущены (например, тесселяция, затенение геометрии и обратная связь преобразования).

Основная программа, находящаяся в памяти компьютера, выполняется процессором и отображается в левом столбце. Действия, выполняемые на видеокарте, перечислены в столбце справа.



Видеокарта имеет собственную память и графический процессор, как небольшой мощный компьютер, специализирующийся на обработке 3D-данных. Программы, работающие на графическом процессоре, называются шейдерами. И главный компьютер, и видеокарта могут работать независимо, и вам следует держать их обоих занятыми одновременно, чтобы в полной мере воспользоваться преимуществами аппаратного ускорения.

В течение*спецификация вершины*, настроен упорядоченный список вершин, который передается на следующий шаг. Эти данные могут либо отправляться программой, выполняющейся на ЦП одну вершину за другой, либо считываться из памяти графического процессора напрямую с использованием буферных объектов. Однако следует избегать многократного получения данных через системную шину, поскольку так видеокарта быстрее получит доступ к собственной памяти.

The *вершинный шейдер*обрабатывает данные по каждой вершине. Он получает этот поток вершин вместе с дополнительными атрибутами, такими как связанные координаты текстуры или значения цвета, а также данные, такие как матрица модель-вид-проекция. Его типичная задача — преобразовать вершины и применить матрицу проекции. Помимо интерфейса к следующему этапу, вершинный шейдер также может напрямую передавать данные во фрагментный шейдер.

В течение*примитивная сборка*На этом этапе проецируемые вершины объединяются в примитивы. Этими примитивами могут быть треугольники, линии, точечные спрайты или более сложные объекты, такие как квадрики. Пользователь решает, какой тип примитива следует использовать при вызове функции рисования. Например, если пользователь хочет нарисовать треугольники, OpenGL берет группы из трех вершин и преобразует их все в треугольники.

В течение*обрезка и выбраковка*На этапе удаляются примитивы, которые лежат за пределами просматриваемого объема и поэтому в любом случае не видны. Кроме того, если включено отсечение граней, каждый примитив, который не показывает свою лицевую сторону (а вместо этого показывает обратную сторону), удаляется. Этот шаг эффективно способствует повышению производительности.

The *растеризация*этап дает так называемые*фрагменты*. Эти фрагменты соответствуют пикселям на экране. В зависимости от выбора пользователя для каждого примитива может быть создан набор фрагментов. Вы можете либо заполнить весь примитив (обычно цветными) фрагментами, либо создать только его контуры (например, для визуализации каркасной модели).

Затем каждый фрагмент обрабатывается*фрагментный шейдер*. Наиболее важным результатом фрагментного шейдера является значение цвета фрагмента. На этом этапе обычно применяются текстуры и освещение. Как программа, работающая на ЦП, так и вершинный шейдер могут передавать ему данные. Очевидно, он также имеет доступ к буферу текстур. Поскольку между несколькими вершинами обычно находится много фрагментов, значения, отправляемые вершинным шейдером, обычно интерполируются. Когда это возможно, интенсивные вычисления следует реализовывать в вершинах, а не во фрагментном шейдере, поскольку обычно фрагментов для вычисления гораздо больше, чем вершин.

Заключительный этап,*операции над выборкой*, применяет несколько тестов, чтобы решить, какие фрагменты действительно следует записать в фреймбуфер (тест глубины, маскирование и т. д.). После этого происходит смешивание и итоговое изображение сохраняется во фреймбуфере.

2.6 API OpenGL

В этой главе будут объяснены соглашения, используемые в OpenGL. Хотя мы попытаемся использовать абстракцию Qt для API OpenGL везде, где это возможно, нам все равно придется вызывать некоторые функции OpenGL напрямую. Примеры познакомят вас с необходимыми функциями.

API OpenGL использует собственные типы данных для улучшения переносимости и читаемости. Эти типы гарантированно имеют минимальную дальность и точность на каждой платформе.

Тип	Описание
ГЛенум	Указывает, что ожидается одно из определений препроцессора OpenGL.
GLлогическое значение	Используется для логических значений.
Глбитфилд	Используется для битовых полей.
Глвоид	sed для передачи указателей.
ГЛбайт	1-байтовое целое число со знаком.
GLshort	GLshort 2-байтовое целое число со знаком.
Глинт	4-байтовое целое число со знаком.
ГЛубайт	1-байтовое целое число без знака.
ГЛушорт	2-байтовое целое число без знака.
ГЛуинт	4-байтовое целое число без знака.
GLsizei	Используется для размеров.
ГЛфлоат	Число с плавающей запятой одинарной точности.
GLclampf	Число с плавающей запятой одинарной точности в диапазоне от 0 до 1.
GLдабл	Число двойной точности с плавающей запятой.
GLclampd	Число двойной точности с плавающей запятой в диапазоне от 0 до 1.

Различные определения препроцессора OpenGL имеют префикс GL_*. Его функции начинаются с *гл*. Например, функция, запускающая процесс рендеринга, объявляется как void glDrawArrays(режим GLenum, сначала GLint, счетчик GLsizei)*.

2.7 Язык шейдеров OpenGL

Как мы уже узнали, программирование шейдеров является одним из основных требований при использовании OpenGL. Шейдерные программы написаны на языке высокого уровня, называемом*Язык шейдеров OpenGL (GLSL)*, язык, очень похожий на C. Чтобы установить программу шейдера, исходный код шейдера необходимо отправить на видеокарту в виде строки, где программу затем необходимо скомпилировать и скомпоновать.

Язык определяет различные типы, соответствующие его потребностям.

Тип	Описание
пустота	Нет*возврат функции*ценность или*пустой параметр*список.
плавать	Значение с плавающей запятой.
интервал	Целое число со знаком.
логическое значение	Логическое значение.
век2, век3, век4	Вектор с плавающей запятой.
ивек2, ивек3, ивек4	Целочисленный вектор со знаком.
бвек2, бвек3, бвек4	Булев вектор.
мат2, мат3, мат4	Матрица с плавающей запятой 2x2, 3x3, 4x4.
сэмплер2D	Доступ к 2D-текстуре.
сэмплерКуб	Доступ к текстуре, отображенной в кубе.

Все эти типы можно комбинировать с помощью C-подобной структуры или массива.

Для доступа к элементам вектора или матрицы можно использовать квадратные скобки «[]» (например, вектор[индекс] = значение* и*матрица[столбец][строка] = значение;*). В дополнение к этому, именованные компоненты вектора доступны с помощью оператора выбора поля «». (например,*вектор.x = xValue*и *вектор.xy = vec2(xValue, yValue)*). Имена*(x, y, r, ш)*используются для позиций. *(р, г, б, а)*и*(с, т, п, ц)* используются для адресации значений цвета и координат текстуры соответственно.

Чтобы определить связь между различными шейдерами, а также между шейдерами и приложением, GLSL предоставляет переменным дополнительную функциональность за счет использования квалификаторов хранилища. Эти квалификаторы хранилища должны быть записаны перед именем типа во время объявления.

Стор-возраст Качественно-огонь	Описание
никто	(по умолчанию) Обычная переменная
константа	Константа времени компиляции
в-дань	Связь между вершинным шейдером и OpenGL для данных по вершинам. Поскольку вершинный шейдер выполняется один раз для каждой вершины, это значение, доступное только для чтения, сохраняет новое значение при каждом запуске. Например, он используется для передачи вершин в вершинный шейдер.
униформа	Связь между шейдером и OpenGL для рендеринга данных. Это значение, доступное только для чтения, не меняется на протяжении всего процесса рендеринга. Он используется для передачи матрицы модель-вид-проекция, например, поскольку этот параметр не меняется для одного объекта.
меняющийся	Связь между вершинным шейдером и фрагментным шейдером для интерполированных данных. Эта переменная используется для передачи значений, рассчитанных в вершинном шейдере, во фрагментный шейдер. Чтобы это работало, переменные должны иметь одно и то же имя в обоих шейдерах. Поскольку между несколькими вершинами обычно находится много фрагментов, данные, рассчитанные вершинным шейдером, (по умолчанию) интерполируются. Такие переменные часто используются в качестве координат текстуры или расчетов освещения.

Чтобы отправить данные из вершинного шейдера во фрагментный шейдер,*вне*переменная вершинного шейдера и*в* переменная фрагментного шейдера должна иметь одно и то же имя. Поскольку между несколькими вершинами обычно находится много фрагментов, данные, рассчитанные вершинным шейдером, по умолчанию интерполируются с учетом перспективы. Чтобы обеспечить такое поведение, перед этим можно написать дополнительный квалификатор Smooth*. *в*. Чтобы использовать линейную интерполяцию,*бесперспективный*квалификатор может быть установлен. Интерполяцию можно полностью отключить, используя*этв*, что приводит к использованию значения, выводимого первой вершиной примитива, для всех фрагментов между примитивами.

Такие переменные обычно называют*вариации*, из-за этой интерполяции и из-за того, что в более ранних версиях OpenGL эта связь между шейдерами достигалась с использованием квалификатора переменной, называемогovariing* вместо*ви вне*.

Для связи с конвейером используются несколько встроенных переменных. Мы будем использовать следующее:

Вари-способный Имя	Описание
век4 gl_Positio	На этапе растеризации необходимо знать положение преобразованного *н*вершина. Поэтому вершинному шейдеру необходимо установить для этой переменной значение расчетное значение.
век4 gl_FragC	*олеовр*в конечном итоге будет записан в буфер кадра. Это значение может быть установлено с помощью фрагментный шейдер.

При использовании нескольких квалификаторов переменных порядок следующий: <квалификатор хранилища> <квалификатор точности> <тип> <имя>*.

Как и в C, точкой входа в каждую программу GLSL является функция main()*, но вы также можете объявлять свои собственные функции. Функции в GLSL работают совсем иначе, чем в C. У них нет возвращаемого значения. Вместо этого значения возвращаются с использованием соглашения о вызовах, называемого*возврат стоимости*. Для этой цели GLSL использует квалификаторы параметров, которые должны быть записаны перед типом переменной во время объявления функции. Эти квалификаторы определяют, происходит ли обмен значениями между функцией и ее вызывающей стороной и когда это происходит.

Параметр квалификатор	Описание
в	(по умолчанию) При вводе переменная инициализируется значением, переданным вызывающей стороной.
вне	При возврате значение этой переменной записывается в переменную, переданную вызывающей стороной. Переменная не инициализируется.
входной	Сочетание входа и выхода. Переменная инициализируется и возвращается.

На самом деле существует еще много квалификаторов, но их перечисление выходит за рамки данного руководства.

Язык также предлагает структуры управления, такие как if*,*выключатель*,*для*,*пока*, и*делать пока*, включая*перерыв*и*возвращаться*. Кроме того, во фрагментном шейдере вы можете вызвать *отказаться*чтобы выйти из фрагментного шейдера и игнорировать этот фрагмент остальной частью конвейера.

GLSL также использует несколько директив препроцессора. Самый заметный из них, который вам следует использовать во всех ваших программах, — это #version*, за которым следуют три цифры языковой версии, которую вы хотите использовать (например,*#версия 330*для версии 3.3). По умолчанию OpenGL предполагает версию 1.1, что не всегда может быть тем, что вам нужно.

Хотя GLSL очень похож на C, существуют некоторые ограничения, о которых вам следует знать:

Функции не могут быть рекурсивными.

Циклы for должны иметь количество итераций, известное во время компиляции.

Никаких указателей нет.

Индексирование массива возможно только с постоянными индексами.

Приведение типов возможно только с использованием конструкторов (например,*myFloat = поплавок (myInt);*).

Примечание:Сцена, которую вы хотите визуализировать, может быть настолько сложной, что в ней будут тысячи вершин и миллионы фрагментов. Вот почему современные видеокарты оснащены несколькими блоками потоковой обработки, каждый из которых одновременно выполняет один вершинный или фрагментный шейдер. Поскольку все вершины и фрагменты обрабатываются параллельно, шейдер не может запросить свойства другой вершины или фрагмента.

Использование OpenGL в вашем приложении Qt

Qt предоставляет виджет под названием*QGL-виджет*для рендеринга графики OpenGL, что позволяет легко интегрировать OpenGL в ваше приложение Qt. Он является подклассом и используется как любой другой QWidget и является кроссплатформенным. Обычно вы переопределяете следующие три виртуальных метода:

> *QGLWidget::initializeGL()*-устанавливает контекст рендеринга OpenGL. Он вызывается один раз перед*QGLWidget::resizeGL()*или*QGLWidget::paintGL()*функция вызывается впервые.*QGLWidget::resizeGL()*-вызывается всякий раз, когда *QGL-виджет*изменяется размер, и после инициализации. Этот метод обычно используется для настройки области просмотра и проекции.*QGLWidget::paintGL()*-визуализирует сцену OpenGL. Это сравнимо с*QWidget::paint()*.

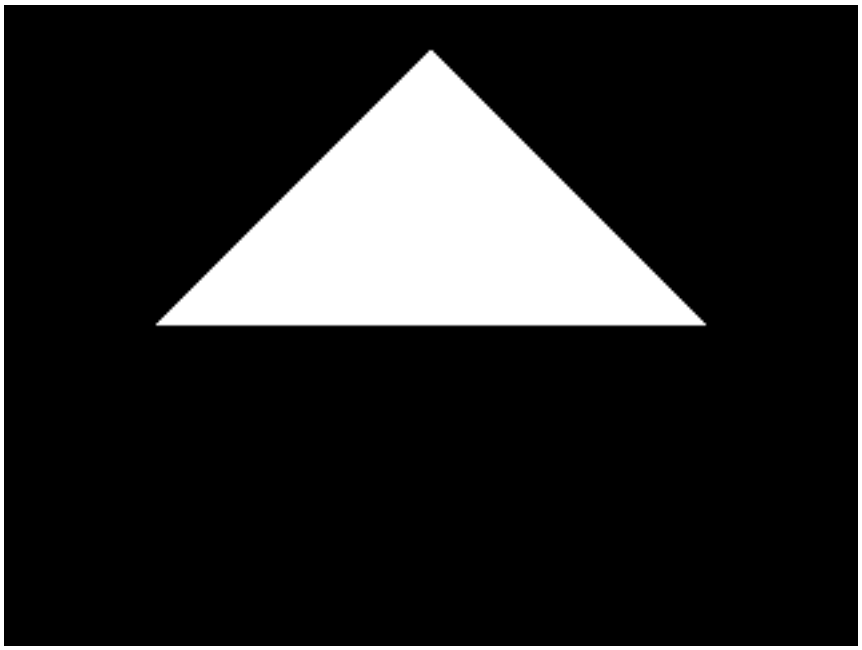
Qt также предлагает кроссплатформенную абстракцию для шейдерных программ, называемую *QGLShaderПрограмма*. Этот класс облегчает процесс компиляции и связывания программ шейдеров, а также переключение между различными шейдерами.

Примечание:Возможно, вам придется адаптировать версии, установленные в примерах исходных кодов, к версиям, поддерживаемым вашей системой.

3.1 Привет OpenGL

Мы начинаем с небольшого*Привет, мир*пример, в котором наша видеокарта будет отображать простой треугольник. Для этого мы создадим подкласс*QGL-виджет*чтобы получить контекст рендеринга OpenGL и написать простой вершинный и фрагментный шейдер.

Этот пример подтверждает, правильно ли мы настроили среду разработки.



Примечание:Исходный код, относящийся к этому разделу, находится в*примеры/привет-opengl/*каталог.

Прежде всего, нам нужно указать qmake использовать*QtOpenGL*модуль. Поэтому мы добавляем:

```
QT+=OpenGL
```

The *основной()*функция служит только для создания экземпляра и показа нашего*QGL-виджет* подкласс.

```
интервалосновной(интерваларгк,голец**аргв) {

    QApplication a(argc, argv);

    ГлВиджет w;
    ш.шоу();

    возвращатьсяa.exec();
}
```

Наш класс виджетов OpenGL определяется следующим образом:

Мы хотим, чтобы виджет был подклассом*QGL-виджет*. Поскольку позже мы можем использовать сигналы и слоты, мы вызываем метод*Q_OBJECT*макрос. Кроме того, мы переопределяем*QWidget::minumSizeHint()*и *QWidget::sizeHint()*чтобы установить разумные размеры по умолчанию.

Для вызова обычных команд рендеринга OpenGL мы переопределяем три виртуальные функции *GLWidget::initializeGL()*,*QGLWidget::resizeGL()*, и*QGLWidget::paintGL()*.

Нам также нужны некоторые переменные-члены.*pMatrix*это*QMatrix4x4*который сохраняет проекционную часть конвейера трансформации. Для управления шейдерами мы используем*QGLShaderПрограмма*названный, *программа шейдера.вершины*это*QVector*сделано из*QVector3Ds*который хранит вершины треугольника. Хотя вершинный шейдер ожидает, что мы отправим однородные координаты, мы можем использовать 3D-векторы, поскольку конвейер OpenGL автоматически устанавливает для четвертой координаты значение по умолчанию, равное 1.

```
сортГлвиджет:общественныйQGLWidget {

    Q_OBJECT

общественный:
    GlWidget(QWidget*родитель"="0); ~
    ГлВиджет();
    QSize sizeHint()константа;

защищенный:
    пустотаинициализироватьGL();
    пустотаизменить размерGL(интервалширина,интервал
    высота); пустотакраскаГЛ();

частный:
    QMatrix4x4 pMatrix;
    QGLShaderProgram программа шейдера;
    QVector<QVector3D>вершины;
};
```

Теперь, когда мы определили наш виджет, мы можем наконец поговорить о реализации.

Вызов списка инициализаторов конструктора *QGLWidget's* конструктор, передающий *QGLФормат* объект. Это можно использовать для установки возможностей контекста рендеринга OpenGL, таких как двойная буферизация или множественная выборка. Нас устраивают значения по умолчанию, поэтому мы могли бы также опустить *QLFormat*. Qt пытается получить контекст рендеринга как можно ближе к тому, что мы хотим.

Затем мы переопределяем *QWidget::sizeHint()* чтобы установить разумный размер по умолчанию для виджета.

```
Глвиджет::GlWidget(QWidget*родитель)
    :QGLWidget(QGLFormat(/* Дополнительные параметры формата */), родитель)
{
}

Глвиджет::~ГлВиджет() {

}

QSize GlWidget::размерПодсказка()константа {

    возвращатьсяQSize(640,480);
}
```

The *QGLWidget::initializeGL()* метод вызывается один раз при создании контекста OpenGL. Мы используем эту функцию для установки поведения контекста рендеринга и создания программ шейдеров.

Если мы хотим визуализировать 3D-изображения, нам нужно включить тестирование глубины. Это один из тестов, который можно выполнить на этапе операций с каждым образцом. Это приведет к тому, что OpenGL будет отображать только ближайшие к камере фрагменты, когда примитивы перекрываются. Хотя нам не нужна эта возможность, поскольку мы хотим показать только плоский треугольник, мы можем использовать эту настройку в других наших примерах. Если вы пропустили этот оператор, вы можете увидеть объекты сзади, проходящие сквозь объекты спереди, в зависимости от порядка отображения примитивов. Деактивация этой возможности полезна, если вы хотите нарисовать наложенное изображение поверх экрана.

В качестве простого способа значительно улучшить производительность 3D-приложения мы также включаем

выбраковка лица. Это говорит OpenGL отображать только те примитивы, которые показывают их лицевую сторону. Лицевая сторона определяется порядком вершин треугольника. Вы можете определить, какую сторону треугольника вы видите, посмотрев на его углы. Если углы треугольника указаны в порядке против часовой стрелки, это означает, что передняя часть треугольника — это сторона, обращенная к вам. Для всех треугольников, не обращенных в камеру, этап обработки фрагмента можно опустить.

Затем мы устанавливаем цвет фона, используя *QGLWidget::qglClearColor()*. Это функция, которая вызывает OpenGL.*glClearColor(GLclampf красный, GLclampf зеленый, GLclampf синий, GLclampf apha)*но имеет то преимущество, что позволяет передавать любой цвет, который понимает Qt. Указанный цвет затем будет использоваться во всех последующих вызовах*glClear (маска GLbitfield)*.

В следующем разделе мы настраиваем шейдеры. Мы передаем исходные коды шейдеров в QGLShaderProgram, компилируем и компонуем их, а также привязываем программу к текущему контексту рендеринга OpenGL.

Шейдерные программы должны поставляться в виде исходных кодов. Мы можем использовать *QGLShaderProgram::addShaderFromSourceFile()*чтобы позволить Qt обрабатывать компиляцию. Эта функция компилирует исходный код как указанный тип шейдера и добавляет его в программу шейдера. В случае возникновения ошибки функция возвращает*ЛОЖЬ*и мы можем получить доступ к ошибкам компиляции и предупреждениям, используя *QGLShaderProgram::log()*. Ошибки будут автоматически выведены в стандартный вывод ошибок, если мы запустим программу в режиме отладки.

После компиляции нам еще нужно связать программы с помощью*QGLShaderProgram::link()*. Мы можем снова проверить наличие ошибок и получить доступ к ошибкам и предупреждениям, используя*QGLShaderProgram::log()*.

Затем шейдеры готовы к привязке к контексту рендеринга с помощью*QGLShaderProgram::bind()*. Привязка программы к контексту означает включение ее в графический конвейер. После этого каждая вершина, передаваемая в графический конвейер, будет обрабатываться этими шейдерами, пока мы не вызовем*QGLShaderProgram::release()*отключить их, иначе будет привязана другая шейдерная программа.

Привязку и выпуск программы можно выполнять несколько раз в процессе рендеринга, а это значит, что для разных объектов сцены можно использовать несколько вершинных и фрагментных шейдеров. Поэтому мы будем использовать эти функции в*QGLWidget::paintGL()*функция.

И последнее, но не менее важное: мы настраиваем вершины треугольников. Обратите внимание, что мы определили треугольник, передняя сторона которого направлена в положительное направление z. Включив отсеивание лиц, мы сможем увидеть этот объект, если посмотрим на него с позиций наблюдателя со значением az, превышающим значение z этого объекта.

```
пустотаГлвиджет::инициализироватьGL() {

    glEnable (GL_DEPTH_TEST);
    glEnable (GL_CULL_FACE);

    qglClearColor(QColor(Qt::черный));

    ShaderProgram.addShaderFromSourceFile(QGLShader::Вертекс,":/vertexShader.vsh");
    ShaderProgram.addShaderFromSourceFile(QGLShader::Фрагмент,":/fragmentShader.fsh");
    шейдерПрограмма.ссылка();

    вершины<<QVector3D(1,0,-2)<<QVector3D(0,1,-2)<<QVector3D(-1,0,-2);
}
```

Теперь давайте посмотрим на шейдеры, которые мы будем использовать в этом примере.

Вершинный шейдер вычисляет окончательную проекцию каждой вершины только путем умножения вершины на матрицу модель-вид-проекция.

Ему необходимо прочитать две входные переменные. Первым входом является матрица модель-представление-проекция. Это матрица 4x4, которая меняется один раз для каждого объекта и поэтому объявляется как *униформа mat4*. Мы назвали это *mvpMatrix*. Вторая переменная — это фактическая вершина, которую обрабатывает шейдер. Поскольку шейдер считывает новое значение каждый раз при выполнении, переменную вершины необходимо объявить как *атрибут vec4*. Мы назвали эту переменную *вершина*.

в *основной()* функции, мы просто вычисляем результирующую позицию, которая отправляется на этап растеризации с помощью встроенного матричного векторного умножения.

```
униформа mat4 mvpMatrix;

в вершине vec4;

пустота основной(пустота)
{
    gl_Position = mvpMatrix * вершина;
}
```

Фрагментный шейдер просто отображает цветной пиксель для каждого фрагмента, над которым он выполняется.

Выходными данными фрагментного шейдера является значение, записанное в буфер кадра. Мы назвали эту переменную *фрагментЦвет*. Это пример *век4* с одним элементом для значения красного, зеленого и синего цвета и одним элементом для значения альфа.

Мы хотим использовать один и тот же простой цвет для каждого пикселя. Поэтому мы объявляем входную переменную с именем *цвет*, который представляет собой *униформа vec4*.

The *основной()* функция затем устанавливает встроенный *hl_FragColor* выходную переменную к этому значению.

```
равномерный цвет vec4;

из vec4 fragColor;

пустота основной(пустота)
{
    фрагментЦвет = цвет;
}
```

Переопределенный *QGLWidget::resizeGL()* Метод вызывается всякий раз, когда изменяется размер виджета. Вот почему мы используем эту функцию для настройки матрицы проекции и области просмотра.

После того, как мы проверили высоту виджета, чтобы предотвратить деление на ноль, мы установили для нее матрицу, которая выполняет перспективную проекцию. К счастью, нам не нужно рассчитывать это самостоятельно. Мы можем использовать один из многих полезных методов *QMatrix4x4*, а именно *QMatrix4x4::perspective()*, который делает именно то, что нам нужно. Этот метод умножает *QMatrix4x4* экземпляр с матрицей проекции, заданной углом поля зрения, его соотношением сторон и областями отсечения ближней и дальней плоскостей. Матрица, которую мы получаем с помощью этой функции, напоминает проекцию камеры, которая находится в начале мировой системы координат и смотрит в сторону отрицательного направления z мира, при этом ось x мира направлена вправо, а ось y направлена вверх. Тот факт, что эта функция изменяет свой экземпляр, объясняет необходимость сначала инициализировать ее единичной матрицей (матрицей, которая не меняет вектор при применении в качестве преобразования).

Далее мы настраиваем область просмотра OpenGL. Область просмотра определяет область виджета, на которую отображается результат проекции. Это отображение преобразует нормализованные координаты на пленке вышеупомянутой камеры в координаты пикселей внутри *QGL-виджет*. Чтобы избежать искажений, соотношение сторон окна просмотра должно совпадать с соотношением сторон проекции.

```
пустотаГлвиджет::изменить размерGL(интервалширина,интервалвысота) {

    если(высота==0) {
        высота"="1;
    }

    pMatrix.setToIdentity();
    pMatrix.perspective(60,0, (плавать) ширина/(плавать) высота,0,001,1000);

    glViewport(0,0, ширина высота);
}
```

Наконец, у нас есть OpenGL, рисующий треугольник в *QGLWidget::paintGL()* метод.

Первое, что мы делаем, это очищаем экран, используя *glClear (маска GLbitfield)*. Если эта функция OpenGL вызывается с помощью *GL_COLOR_BUFFER_BIT* set, он заполняет цветовой буфер цветом, установленным *glClearColor(GLclampf красный, GLclampf зеленый, GLclampf синий, GLclampf aplha)*. Установка *GL_DEPTH_BUFFER_BIT* сообщает OpenGL очистить буфер глубины, который используется для проверки глубины и хранит расстояние отрисованных пикселей. Обычно нам необходимо очистить оба буфера, поэтому мы устанавливаем оба бита.

Как мы уже знаем, матрица модель-представление-проекция, используемая вершинным шейдером, представляет собой объединение матрицы модели, матрицы представления и матрицы проекции. Как и в случае с матрицей проекции, мы также используем *QMatrix4x4* класс для обработки двух других преобразований. Хотя мы не хотим использовать их в этом базовом примере, мы уже представляем их здесь, чтобы прояснить их использование. Мы используем их для расчета матрицы модель-представление-проекция, но оставляем их инициализированными единичной матрицей. Это означает, что мы не перемещаем и не вращаем рамку треугольника, а также оставляем без изменений камеру, расположенную в начале мировой системы координат.

Теперь рендеринг можно запустить вызовом функции OpenGL. *glDrawArrays (режим GLenum, сначала GLint, счетчик GLsizei)*. Но прежде чем мы сможем это сделать, нам нужно связать шейдеры и передать им всю необходимую униформу и атрибуты.

В собственном OpenGL программисту сначала придется запросить идентификатор (называемый *расположение*) каждой входной переменной, используя дословное имя переменной, как оно введено в исходном коде шейдера, а затем установите ее значение, используя этот идентификатор и тип, понятный OpenGL. *QGLShaderПрограмма* вместо этого предлагает для этой цели огромный набор перегруженных функций, которые позволяют обращаться к входной переменной, используя либо ее *расположение* или его название. Эти функции также могут автоматически преобразовывать тип переменной из типов Qt в типы OpenGL.

Мы устанавливаем одинаковые значения для обоих шейдеров, используя *QGLShaderProgram::setUniformValue()* передав его имя. Форма вершинного шейдера *Матрица* рассчитывается путем умножения трех его составляющих. Цвет треугольника задается с помощью *QColor* экземпляр, который автоматически преобразуется в *век4* для нас.

Чтобы сообщить OpenGL, где найти поток вершин, мы вызываем QGLShaderProgram::setAttributeArray() и передаем указатель QVector::constData(). Установка массивов атрибутов работает так же, как установка универсальных значений, но есть одно отличие: мы должны явно

включите массив атрибутов с помощью QGLShaderProgram::enableAttributeArray(). Если мы этого не сделаем, OpenGL будет считать, что мы присвоили одно значение вместо массива.

Наконец мы звоним*glDrawArrays (режим GLenum, сначала GLint, счетчик GLsizei)*сделать рендеринг. Он используется для начала рендеринга последовательности геометрических примитивов с использованием текущей конфигурации. Мы проходим*GL_TRIANGLES*в качестве первого параметра, сообщающего OpenGL, что каждая из трех вершин образует треугольник. Второй параметр определяет начальный индекс в массивах атрибутов, а третий параметр — это количество отображаемых индексов.

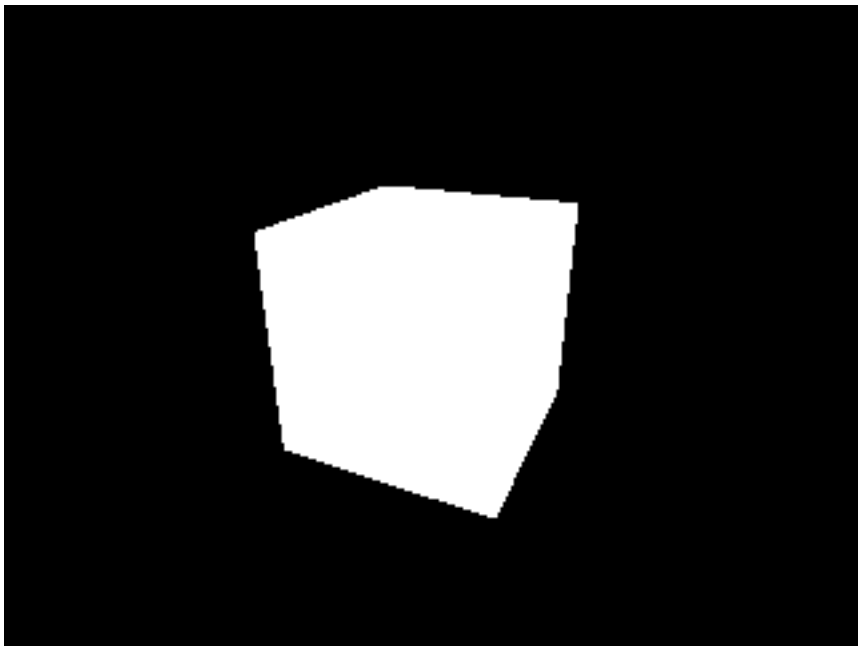
Обратите внимание: если вы позже захотите нарисовать более одного объекта, вам нужно будет только повторить все шаги (за исключением очистки экрана, конечно), которые вы выполнили в этом методе, для каждого нового объекта.

```
пустотаГлвиджет::краскаGL() {

    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

    QMatrix4x4 mMatrix;
    QMatrix4x4 vMatrix;

    программа шейдера.bind();

    ShaderProgram.setUniformValue("мвпматрица", pМатрица*vMatrix*мМатрикс);

    ShaderProgram.setUniformValue("цвет", QColor(Qt::белый));

    ShaderProgram.setAttributeArray("вершина", вершины.constData());
    ShaderProgram.enableAttributeArray("вершина");

    glDrawArrays(GL_TRIANGLES,0, вершины.размер());

    ShaderProgram.disableAttributeArray("вершина");

    шейдерПрограмма.релиз();
}
```

После компиляции и запуска этой программы вы должны увидеть белый треугольник на черном фоне.

3.2 Рендеринг в 3D

Белый треугольник на черном фоне не очень интересен и к тому же не 3D, но теперь, когда у нас есть работающая основа, мы можем расширить его, чтобы создать настоящее 3D-приложение. В этом примере мы отрендерим более сложный объект и реализуем функционал интерактивного исследования нашей сцены.



Примечание:Исходный код, относящийся к этому разделу, находится в*примеры/рендеринг в 3d/*каталог.

Так же, как и с любым*QВиджет*подкласс, мы можем использовать систему событий Qt для обработки пользовательского ввода. Мы хотим иметь возможность рассматривать сцену так же, как если бы мы изучали земной шар. Перетаскивая указатель мыши по виджету, мы хотим изменить угол, под которым мы смотрим. Расстояние до сцены изменится, если мы повернём колесо прокрутки мыши.

Для этой функциональности мы переопределяем *QWidget::mousePressEvent(), QWidget::mouseMoveEvent()*, и*QWidget::wheelEvent()*. Новые переменные-члены*альфа, бета*, и *расстояние*сохранить параметры точки обзора и*последняя позиция мыши*помогает нам отслеживать движение мыши.

```
сортГлвиджет:общественныйQGLWidget {

    . . .

защищенный:
    . . .

    пустота mousePressEvent(QMouseEvent *событие);
    пустота mouseMoveEvent(QMouseEventwheelEvent(QWheelEvent)
    пустота *событие);

частный:
    . . .

    двойной альфа;
    двойной бета;
    двойной расстояние;
    QPoint LastMousePosition;
};
```

Самым важным нововведением в этом примере является использование матрицы представления. Опять же, мы не рассчитываем эту матрицу сами, а используем*QMatrix4x4::lookAt()*функция для получения

эта матрица. Эта функция принимает положение зрителя, точку, на которую смотрит зритель, и вектор, определяющий направление вверх. Мы хотим, чтобы зритель посмотрел на происхождение мира и начал с позиции, расположенной на определенном расстоянии (*расстояние*) вдоль оси z, причем направление вверх соответствует оси y. Затем мы вращаем эти две вершины, используя матрицу преобразования. Сначала мы вращаем их (и их систему координат) на*альфа*угол вокруг их новой повернутой оси X, которая наклоняет камеру. Обратите внимание, что вы также можете проиллюстрировать преобразование наоборот: сначала мы поворачиваем вершины на*бета*угол вокруг оси X мира, а затем мы вращаем их на *альфа*угол вокруг мировой оси Y.

```
пустотаГлвиджет::краскаGL() {

    . . .
    Камера QMatrix4x4Трансформация;
    cameraTransformation.rotate(альфа,0,1,0);
    cameraTransformation.rotate(бета,1,0,0);

    QVector3D-камераПоложение"="камераТрансформация*QVector3D(0,0, расстояние); QVector3D-
    камераUpDirection"="камераТрансформация*QVector3D(0,1,0);

    vMatrix.lookAt(cameraPosition, QVector3D(0,0,0), cameraUpDirection); . . .

}
```

Эти три параметра необходимо инициализировать в конструкторе, и, чтобы учесть ввод пользователя, мы затем изменяем их в соответствующих обработчиках событий.

```
Глвиджет::GlWidget(QWidget*родитель)
    :QGLWidget(QGLFormat(/* Дополнительные параметры формата */), родитель)
{
    альфа"="25;
    бета"="25;
    расстояние"="2,5;
}
```

в*QWidget::mousePressEvent()*, мы сохраняем исходное положение указателя мыши, чтобы иметь возможность отслеживать движение. в*QGLWidget::mouseMoveEvent()*, рассчитываем изменение указателей и адаптируем углы*альфа*и*бета*. Поскольку параметры точки обзора изменились, мы вызываем *QGLWidget::updateGL()*чтобы вызвать обновление контекста рендеринга.

```
пустотаГлвиджет::mousePressEvent (QMouseEvent*событие) {

    последняя позиция мыши"="событие->позиция();

    событие->принимать();
}

пустотаГлвиджет::mouseMoveEvent(QMouseEvent*событие) {

    интервалдельтаX"="событие->Икс()-LastMousePosition.x();
    интервалдельтаY"="событие->y()-LastMousePosition.y();

    если(событие->кнопки()&Qt::Левая кнопка) {
        альфа"="дельтаX;
        пока(альфа<0) {
            альфа+=360;
        }
```

```
        пока(альфа>=360) {
            альфа"="360;
        }

        бета"="дельтаY;
        если(бета<-90) {
            бета"="90;
        }
        если(бета>90) {
            бета"="90;
        }

        обновитьГЛ();
    }

    последняя позиция мыши"="событие->позиция();

    событие->принимать();
}
```

в*QGLWidget::wheelEvent()*, мы либо увеличиваем, либо уменьшаем расстояние до зрителя на 10% и снова обновляем рендеринг.

```
пустотаГлвиджет::колесоEvent(QWheelEvent*событие) {

    интервалдельта"="событие->дельта();

    если(событие->ориентация()==Qt::Вертикально) {
        если(дельта<0) {
            расстояние"="1.1; }еще
        если(дельта>0) {
            расстояние"="0,9;
        }

        обновитьГЛ();
    }

    событие->принимать();
}
```

Чтобы закончить этот пример, нам нужно всего лишь изменить список вершин, чтобы сформировать куб.

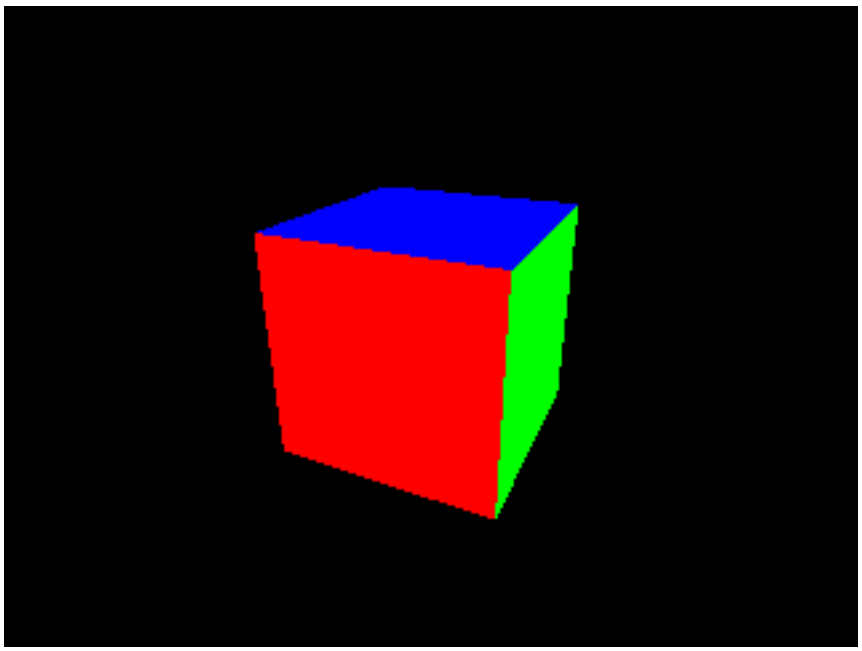
```
пустотаГлвиджет::инициализироватьGL() {

    . . .
    вершины<<QVector3D(-0,5,-0,5,          0,5)<<QVector3D(0,5,-0,5, 0,5)<<          0,5)<<QVector3D( 0,5)<<
            <<QVector3D(0,5,0,5,           QVector3D(-0,5,0,5,                       QVector3D(
            <<QVector3D(0,5,-0,5,-0,5)<<QVector3D(-0,5,-0,5,-0,5)<<QVector3D( <<QVector3D(-0,5,0,5,-0,5)<<
            QVector3D(0,5,0,5,-0,5)<<QVector3D( <<QVector3D(-0,5,-0,5,-0,5)<<QVector3D(-0,5,-0,5, <<
            QVector3D(-0,5,0,5, <<QVector3D(0,5,-0,5, <<QVector3D(0,5, <<QVector3D(-0,5,<<QVector3D(0,5,
                                                0,5)<<QVector3D(-0,5,0,5,-0,5)<<QVector3D( 0,5)<<QVector3D(0,5,
                                                0,5,-0,5)<<QVector3D( 0,5,-0,5)<<QVector3D(0,5, 0,5,0,5)<<
                                        QVector3D(0,5, 0,5,-0,5)<<QVector3D(-0,5,0,5,      0,5)<<QVector3D( 0,5)<<
                                                                                0,5,       QVector3D(
                                                                                0,5,-0,5)<<QVector3D(
            <<QVector3D(-0,5,-0,5,-0,5)<<QVector3D(0,5,-0,5,-0,5)<<QVector3D( <<QVector3D(0,5,-0,5,0,5)<<
            QVector3D(-0,5,-0,5,0,5)<<QVector3D(
```

}

Если вы теперь скомпилируете и запустите эту программу, вы увидите белый куб, который можно вращать с помощью мыши. Поскольку каждая из шести сторон окрашена в один и тот же цвет плоскости, глубина не видна. Мы поработаем над этим в следующем примере.

3.3 Раскраска

В этом примере мы хотим раскрасить каждую сторону куба в разные цвета, чтобы усилить иллюзию трехмерности. Чтобы заархивировать это, мы расширим наши шейдеры таким образом, чтобы мы могли указывать один цвет для каждой вершины и использовать интерполяцию вариаций для генерации цветов фрагмента. В этом примере показано, как передавать данные из вершинного шейдера во фрагментный шейдер.



Примечание:Исходный код, относящийся к этому разделу, находится в папке*примеры/раскраски/*каталог

Чтобы сообщить шейдерам о цветах, мы указываем значение цвета для каждой вершины как массив атрибутов вершинного шейдера. Таким образом, при каждом запуске шейдера он будет считывать новое значение как для атрибута вершины, так и для атрибута цвета.

Поскольку цвет фрагмента в конечном итоге должен быть установлен во фрагментном шейдере, а не в вершинном шейдере, мы передаем ему значение цвета. Для этого нам нужно объявить переменную с одинаковым именем в обоих шейдерах. Мы назвали это переменным*варьирующийсяЦвет*. Если теперь фрагментный шейдер запускается для каждого фрагмента между тремя вершинами треугольника, значение, считываемое шейдером, вычисляется путем интерполяции значений трех углов. Это означает, что если мы укажем один и тот же цвет для трех вершин треугольника, OpenGL нарисует треугольник плоского цвета. Если мы укажем разные цвета, OpenGL будет плавно смешивать эти значения.

В основной функции вершинного шейдера нам нужно только установить переменную значения цвета.

```
униформа mat4 mvpMatrix;

в вершине vec4;
в цвете vec4;

out vec4 варьирующийся цвет;

пустотаосновной(пустота)
{
    варьирующийсяЦвет"="цвет;
    gl_Position"="mvpMatrix*вершина;
}
```

В основной функции фрагментного шейдера мы устанавливаем*gl_FragColor*переменная к полученному цвету.

```
в vec4 варьирующийся цвет;

из vec4 fragColor;

пустотаосновной(пустота)
{
    фрагментЦвет"="варьирующийся цвет;
}
```

Конечно, нам все еще нужно использовать новую структуру для хранения значений цвета и отправки их шейдерам в файле.*QGLWidget::paintGL()*метод. Но это должно быть очень просто, поскольку мы уже сделали все это для*вершины*массив атрибутов таким же образом.

```
// glwidget.h

сортГлвиджет:общественныйQGLWidget {

    . . .

    частный:
        . . .

        QVector<QVector3D>цвета;
};

// glwidget.cpp

пустотаГлвиджет::краскаGL() {

    . . .

    программа шейдера.bind();

    ShaderProgram.setUniformValue("мвпматрица", pМатрица*vMatrix*мМатрикс);

    ShaderProgram.setAttributeArray("вершина", вершины.constData());
    ShaderProgram.enableAttributeArray("вершина");

    ShaderProgram.setAttributeArray("цвет", цвета.constData());
    ShaderProgram.enableAttributeArray("цвет");

    glDrawArrays(GL_TRIANGLES,0, вершины.размер());
```

```
        ShaderProgram.disableAttributeArray("вершина");

        ShaderProgram.disableAttributeArray("цвет");

        шейдерПрограмма.релиз();
}
```

Есть только одно небольшое неудобство при переключении с цветовой формы на массив атрибутов цвета. К сожалению *QGLShaderProgram::setAttributeArray()* не поддерживает *QColor* type, поэтому нам нужно сохранить цвета как *QVector3D* (или *QVector4D*, если вы хотите установить значение альфа для изменения непрозрачности). Допустимые значения цвета варьируются от 0 до 1. Поскольку мы хотим окрасить каждую грань куба в простой цвет, мы устанавливаем одинаковое значение цвета вершин каждой грани.

```
пустотаГлвиджет::инициализироватьGL() {

        . . .

        цвета<<QVector3D(1,0,0)<<QVector3D(1,0,0)<<QVector3D(1,0,0)//Передний
                <<QVector3D(1,0,0)<<QVector3D(1,0,0)<<QVector3D(1,0,0)
                <<QVector3D(1,0,0)<<QVector3D(1,0,0)<<QVector3D(1,0,0)//Назад <<QVector3D(1,0,0)<<
                QVector3D(1,0,0)<<QVector3D(1,0,0)
                <<QVector3D(0,1,0)<<QVector3D(0,1,0)<<QVector3D(0,1,0)//Левый <<QVector3D(0,1,0)<<
                QVector3D(0,1,0)<<QVector3D(0,1,0)
                <<QVector3D(0,1,0)<<QVector3D(0,1,0)<<QVector3D(0,1,0)//Верно <<QVector3D(0,1,0)<<
                QVector3D(0,1,0)<<QVector3D(0,1,0) <<QVector3D(0,0,1)<<QVector3D(0,0,1)<<QVector3D(0,0,1)//
                Вершина <<QVector3D(0,0,1)<<QVector3D(0,0,1)<<QVector3D(0,0,1)

                <<QVector3D(0,0,1)<<QVector3D(0,0,1)<<QVector3D(0,0,1)//Нижний <<QVector3D(0,0,1)<<
                QVector3D(0,0,1)<<QVector3D(0,0,1);
}
```

Теперь у нашего куба шесть сторон окрашены в другой цвет.

3.4 Наложение текстур

Наложение текстур — очень важная концепция в компьютерной 3D-графике. Это нанесение изображений поверх поверхностей модели, необходимое для создания красивой 3D-сцены.

С текстурами можно делать больше, чем просто накладывать их на поверхность. По сути, текстура — это двумерный массив, содержащий значения цвета, поэтому вы можете передавать в шейдеры не только цвета, но и массив любых данных, которые вам нужны. Однако в этом примере мы будем использовать классическую 2D-текстуру для отображения изображения поверх куба, который мы создали в предыдущих примерах.

Примечание:Исходный код, относящийся к этому разделу, находится в папке*примеры/текстурное отображение/* каталог

Чтобы сопоставить текстуру с примитивом, нам нужно указать так называемые координаты текстуры*, которые сообщают OpenGL, какая координата изображения к какой вершине должна быть прикреплена. Координаты текстуры являются экземплярами*век2*которые нормализованы к диапазону от 0 до 1. Начало системы координат текстуры находится в левом нижнем углу изображения, причем первая ось указывает на правую сторону, а вторая ось — вверх (т. е. нижний левый угол изображения). изображение находится на*(0, 0)*и правый верхний угол находится на*(1, 1)*). Также допускаются значения координат выше 1, что приводит к обтеканию текстуры по умолчанию.

Сами текстуры представляют собой объекты OpenGL, хранящиеся в памяти видеокарты. Они созданы с использованием*glGenTextures (GLsizei n, текстура GLuint)*и снова удалил с помощью вызова *glDelereTextures(GLsizei n, const GLuint *texture)*. Для идентификации текстур каждой текстуре при создании присваивается идентификатор текстуры. Как и в случае с шейдерными программами, они должны быть привязаны к*glBindTexture (цель GLenum, текстура GLuint)*прежде чем их можно будет настроить и заполнить данными. Мы можем использовать Qt*QGLWidget::bindTexture()*для создания объекта текстуры. Обычно нам нужно убедиться, что данные изображения имеют определенный формат в соответствии с конфигурацией объекта текстуры, но, к счастью,*QGLWidget::bindTexture()*может позаботиться об этом.

OpenGL позволяет нам иметь несколько текстур, доступных шейдерам одновременно. Для этой цели OpenGL использует так называемые текстурные блоки*. Поэтому, прежде чем мы сможем использовать текстуру, нам нужно привязать ее к одному из текстурных блоков, определенных перечислением.*GL_TEXTURE*i(где я от 0 до *GL_MAX_COMBINED_TEXTURE_UNITS -1*). Для этого мы вызываем*glActiveTexture (текстура GLenum)*и свяжите текстуру, используя*glBindTexture (цель GLenum, текстура GLuint)*. Чтобы добавить новые текстуры или изменить существующие, нам нужно вызвать*glBindTexture (цель GLenum, текстура GLuint)*, перезаписывает текущий активный текстурный блок. Поэтому вам следует установить недопустимый активный текстурный блок после его установки, вызвав*глАктивететекстуре (0)*. Таким образом, одновременно можно настроить несколько текстурных блоков. Обратите внимание, что текстурные блоки должны использоваться в порядке возрастания, начиная с *GL_TEXTURE0*.

Чтобы получить доступ к текстуре в шейдере для ее фактической визуализации, мы используем метод *текстура2D (сэмплер sampler2D, координата vec2)* функция для запроса значения цвета в определенной координате текстуры. Эта функция считывает два параметра. Первый параметр имеет тип *сэмплер2D* и это относится к текстурному блоку. Второй параметр — это координата текстуры, к которой мы хотим получить доступ. Чтение из текстурного блока *я* обозначается перечислением *GL_TEXTUREi*, нам нужно пройти *ГЛуинт я* как единое значение.

Благодаря всей этой теории мы теперь можем сделать наш куб текстурированным.

Мы заменяем *век4* атрибут цвета и соответствующее изменение в зависимости от *век2* переменную для координат текстуры и передайте это значение фрагментному шейдеру.

```
униформа mat4 mvpMatrix;

в вершине vec4;
в координате текстуры vec2;

out vec2variableTextureCoordinate;

пустота основной(пустота)
{
    варьирующаясятекстуракоордината"="координата
    текстуры; gl_Position"="mvpMatrix*вершина;
}
```

Во фрагментном шейдере мы используем *текстура2D (сэмплер sampler2D, координата vec2)* чтобы найти правильное значение цвета. Униформа *текстура* типа *сэмплер2D* выбирает текстурный блок, и мы используем интерполированные значения, поступающие из вершинного шейдера, в качестве координат текстуры.

```
однородная текстура sampler2D;

в vec2variableTextureCoordinate;

из vec4 fragColor;

пустота основной(пустота)
{
    фрагментЦвет"="текстура2D (текстура, варьирующаяся координата текстуры);
}
```

в *Глвиджет* объявление класса, мы заменяем ранее использованное *цвета* член с *QVector* сделано из *QVector2Ds* для координат текстуры и добавьте переменную-член для хранения идентификатора объекта текстуры.

```
сорт Глвиджет:общественный QGLWidget {

    . . .

    частный:
        . . .

        QVector<QVector2D> текстурные координаты;
        ГЛуинтовая текстура;
        . . .

};
```

в*QGLWidget::initializeGL()*При повторной реализации мы устанавливаем координаты текстуры, а также создаем объект текстуры. Каждая сторона будет покрыта полным квадратным изображением, содержащимся в нашем файле ресурсов.

```
пустотаГлвиджет::инициализироватьGL() {

    . . .
    текстураКоординаты<<QVector2D(0,0)<<QVector2D(1,0)<<QVector2D(1,1)//Передний
                       <<QVector2D(1,1)<<QVector2D(0,1)<<QVector2D(0,0)
                       <<QVector2D(0,0)<<QVector2D(1,0)<<QVector2D(1,1)//Назад <<QVector2D(1,1)<<
                       QVector2D(0,1)<<QVector2D(0,0)
                       <<QVector2D(0,0)<<QVector2D(1,0)<<QVector2D(1,1)///Левый <<QVector2D(1,1)<<
                       QVector2D(0,1)<<QVector2D(0,0)
                       <<QVector2D(0,0)<<QVector2D(1,0)<<QVector2D(1,1)//Верно <<QVector2D(1,1)<<
                       QVector2D(0,1)<<QVector2D(0,0) <<QVector2D(0,0)<<QVector2D(1,0)<<QVector2D(1,1)
                       //Вершина <<QVector2D(1,1)<<QVector2D(0,1)<<QVector2D(0,0)

                       <<QVector2D(0,0)<<QVector2D(1,0)<<QVector2D(1,1)//Ботто <<QVector2D(1,1)<<
                       QVector2D(0,1)<<QVector2D(0,0);

    текстура"="bindTexture(QPixmap(":/texture.png"));
}
```

в*QGLWidget::paintGL()*мы устанавливаем униформу sampler2D фрагментного шейдера в качестве первого текстурного блока. Затем мы активируем этот модуль, привязываем к нему наш объект текстуры и после этого снова деактивируем его, чтобы предотвратить случайную перезапись этой настройки. И вместо передачи массива атрибутов цвета мы передаем массив, содержащий координаты текстуры.

```
пустотаГлвиджет::краскаGL() {

    . . .

    программа шейдера.bind();

    ShaderProgram.setUniformValue("мвпматрица", рМатрица*vMatrix*мМатрикс);

    ShaderProgram.setUniformValue("текстура",0);

    //glActiveTexture(GL_TEXTURE0); glBindTexture
    (GL_TEXTURE_2D, текстура); //glActiveTexture(0);


    ShaderProgram.setAttributeArray("вершина", вершины.constData());
    ShaderProgram.enableAttributeArray("вершина");

    ShaderProgram.setAttributeArray("Координата текстуры"
    ,textureCoordinates.constData())shaderProgram.enableAttributeArray("Координата текстуры");

    glDrawArrays(GL_TRIANGLES,0, вершины.размер());

    ShaderProgram.disableAttributeArray("вершина");

    ShaderProgram.disableAttributeArray("Координата текстуры");

    шейдерПрограмма.релиз();
}
```

Наш куб теперь текстурирован.

Примечание:Заголовочный файл Windows OpenGL включает в себя функциональные возможности только до версии OpenGL 1.1 и предполагает, что программист самостоятельно получит дополнительные функциональные возможности. Сюда входят вызовы функций API OpenGL, а также перечисления. Причина в том, что, поскольку существуют разные библиотеки OpenGL, программист должен запрашивать точки входа в функции библиотеки во время выполнения.

Qt определяет только функциональность, необходимую для его собственных классов, связанных с OpenGL. *glActiveTexture (текстура GLenum)*так же хорошо как*GL_TEXTUREi*перечисления не принадлежат к этому подмножеству.

Существует несколько служебных библиотек, упрощающих определение этих функций (например, GLEW, GLEE и т. д.). Мы определим*glActiveTexture (текстура GLenum)*и*GL_TEXTUREi*вручную.

Сначала мы включаем*glext.h*заголовочный файл, чтобы установить недостающие перечисления и несколько определений типов, которые помогут нам сделать код читабельным (поскольку версия, поставляемая с вашим компилятором, может быть устаревшей, вам может потребоваться получить последнюю версию с сайтадомашняя страница OpenGL[1]). Далее мы объявляем указатель на функцию, которую будем использовать для вызова*glActiveTexture (текстура GLenum)*используя включенные определения типов. Чтобы не запутать компоновщик, мы используем другое имя, чем*glActiveTexture*и определите макрос препроцессора для замены вызовов*glActiveTexture (текстура GLenum)*с нашей собственной функцией:

```
# ifdef WIN32
    # включаем <GL/glext.h>
    PFNGLACTIVETEXTUREPROC pGlActiveTexture"="НУЛЕВОЙ;
    # определяем glActiveTexture pGlActiveTexture
# окончание//WIN32
```

в*GlWidget::initializeGL()*функцию, мы запрашиваем этот указатель, используя*PROC WINAPI wglGet-ProcAddress(LPCSTR lpszProc)*. Эта функция считывает имя функции OpenGL API и возвращает указатель, который нам нужно привести к правильному типу:

```
пустотаГлвиджет::инициализироватьGL() {

    . . .

    # ifdef WIN32
        glActiveTexture"="(PFNGLACTIVETEXTUREPROC) wglGetProcAddress((LPCSTR)"glActiveT
    # окончание
    . . .
}
```

*глактиветекстуре()*и*GL_TEXTUREi*затем можно использовать в Windows.

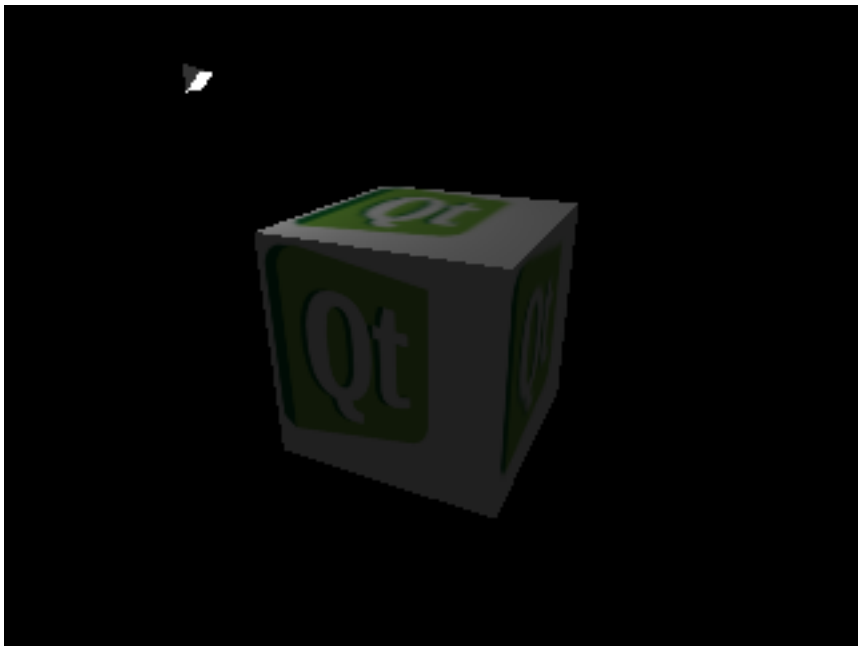
3.5 Освещение

Возможность писать собственные программы шейдеров дает вам возможность настроить тот эффект освещения, который лучше всего соответствует вашим потребностям. Это может варьироваться от очень простых и экономящих время подходов до высококачественных алгоритмов трассировки лучей.

[1] http://www.opengl.org

В этой главе мы реализуем технику, называемую затенением Фонга*, которая является популярным методом затенения базовой линии для многих приложений рендеринга. Для каждого пикселя на поверхности объекта мы рассчитаем интенсивность цвета на основе положения и цвета источника света, а также текстуры объекта и свойств его материала.

Чтобы показать результаты, мы отобразим куб с кружащим над ним источником света. Источник света будет отмечен пирамидой, которую мы визуализируем с помощью повершинного шейдера цвета одного из предыдущих примеров. Итак, в этом примере вы также увидите, как визуализировать сцену с несколькими объектами и различными программами шейдеров.



Примечание:Исходный код, относящийся к этому разделу, находится в папке*примеры/освещение/*каталог

Поскольку мы используем два разных объекта и две разные шейдерные программы, мы добавили к именам префиксы. Куб визуализируется с помощью*освещениеShaderПрограмма*, для чего нам нужно дополнительное хранилище, которое сохраняет нормаль к поверхности каждой вершины (т.е. вектор, перпендикулярный поверхности и имеющий размер 1). Прожектор, с другой стороны, визуализируется с помощью *раскраскаShaderProgram*, который состоит из шейдера, который мы разработали ранее в этом уроке.

Чтобы отслеживать положение источника света, мы ввели новую переменную-член, фиксирующую его вращение. Это значение периодически увеличивается в*таймаут()*слот.

```
сортГлвиджет:общественныйQGLWidget {

    . . .

    частный:
        QGLShaderProgram освещениеShaderProgram;
        QVector<QVector3D>кубВертицес; QVector<QVector3D
        >кубНормалс; QVector<QVector2D>координаты
        кубической текстуры; GLuint кубическая текстура;

        QGLShaderProgram QVector<QVector3D>
        прожекторВершины;
```

```
        QVector<QVector3D>Цвета прожектора;
        двойнойсветУгол;
    . . .

    частныйQ_СЛОТЫ:
        пустотатаймаут();
};
```

Модель отражения Фонга предполагает, что свет, отраженный от объекта (то есть то, что вы на самом деле видите), состоит из трех компонентов: диффузное отражение шероховатых поверхностей, зеркальные блики глянцевых поверхностей и окружающий термин, который суммирует небольшое количество света, которое попадает разбросаны по всей сцене.

Для каждого источника света в сцене мы определяем$я_д$и$я_с$как интенсивности (значения RGB) диффузного и зеркального компонентов.$я_а$определяется как компонент окружающего освещения.

Для каждого вида поверхности (глянцевой или ровной) мы определяем следующие параметры:$к_д$и $к_с$ установить соотношение отражения диффузной и зеркальной составляющей,$к_а$задает коэффициент отражения окружающего термина соответственно иα— константа блеска, которая управляет размером зеркальных бликов.

Уравнение для расчета освещенности каждой точки поверхности (фрагмента):

$$I_p = k_a i_a + \sum_{m \in lights} (k_d(\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s(\hat{R}_m \cdot \hat{V})^\alpha i_{m,s})$$

$\hat{л}_м$– нормированный вектор направления, направленный от фрагмента к источнику света,$\hat{H}$– нормаль к поверхности этого фрагмента,$P_м$- направление света, отраженного в этой точке, и$\hat{B}$ указывает из фрагмента в сторону зрителя сцены.

Чтобы получить упомянутые выше векторы, мы вычисляем их для каждой вершины в вершинном шейдере и сообщаем OpenGL передать их как интерполированные значения во фрагментный шейдер. Во фрагментном шейдере мы окончательно устанавливаем освещенность каждой точки и совмещаем ее со значением цвета текстуры.

Таким образом, в дополнение к передаче позиций вершин и матрицы модель-вид-проекция для получения положения фрагмента нам также необходимо передать нормаль к поверхности каждой вершины. Чтобы вычислить преобразованный куб$\hat{л}_м$и$\hat{B}$, нам нужно знать часть преобразования, связанную с представлением модели. Чтобы вычислить преобразованное$\hat{H}$, нам нужно применить матрицу, преобразующую нормали к поверхности. Эта дополнительная матрица необходима, потому что мы хотим, чтобы нормали вращались только в соответствии с матрицей представления модели, но не переносились.

Это исходный код вершинного шейдера:

```
униформа mat4 mvpMatrix;
униформа mat4 mvMatrix;
униформа mat3 NormalMatrix;
униформа vec3 LightPosition;

в вершине vec4;
в vec3 нормально;
в координате текстуры vec2;

из vec3 варьируется Нормально; из
vec3variableLightDirection;
```

```
из vec3variableViewerDirection; out
vec2variableTextureCoordinate;

пустотаосновной(пустота)
{
    vec4 глазВертекс"="MBМатрикс*вершина; глазВертекс"="глазВертекс.w;
    варьируетсяНормальный"="нормальныйМатрица*нормальный;
    переменное направление света"="светПозиция-глазВертекс.xyz;
    варьирующееся направление просмотра"="глазВертекс.xyz;
    варьирующаясятекстуракоордината"="координата текстуры; gl_Position"="
    mvpMatrix*вершина;

}
```

Фрагментный шейдер снабжен свойствами источника света и материала, а также геометрическими данными, рассчитанными вершинным шейдером. Затем он устанавливает значение цвета фрагмента в соответствии с приведенной выше формулой.

Это исходный код фрагментных шейдеров:

```
единый фоновый цвет vec4;
однородный vec4 диффузный
цвет; однородный vec4
specularColor; униформа плавать окружающееОтражение;
униформа   плавать диффузноеОтражение;
униформа   плавать зеркальное отражение;
униформа   плавать блеск;
однородная текстура sampler2D;

в vec3 варьируется Нормально;
в vec3 варьирующееся LightDirection; в
vec3 варьирующееся ViewerDirection; в
vec2variableTextureCoordinate;

из vec4 fragColor;

пустотаосновной(пустота)
{
    vec3 нормальный"="нормализовать (варьирующийся нормальный);
    vec3 LightDirection"="нормализовать (variingLightDirection); vec3
    ViewerНаправление"="нормализовать (variingViewerDirection); vec4
    AmbientИллюминация"="окружающееОтражение*окружающий цвет;
    vec4 диффузное освещение"="диффузноеОтражение*Макс(0,0, точка(lightDirection, нормальный)) vec4
    specularIllumination"="зеркальное отражение*мощность(макс(0,0,
                                                точка(-отражать(lightDirection),
                                                блеск)*SpecularCol
    фрагментЦвет"="текстура2D(текстура,
                        варьирующаяся координата текстуры)*(окружающее освещение+диффузныйИллю )
          + зеркальное освещение;
}
```

в *GLWidget::initiliazeGL()* Методом мы настраиваем оба шейдера и готовим массивы атрибутов куба и прожектора. Единственное, что здесь новое, это *QVector* сделано из *QVector3Ds* который хранит нормаль к поверхности каждой вершины куба.

```
пустотаГлвиджет::инициализироватьGL() {

    . . .

    освещениеShaderProgram.addShaderFromSourceFile(QGLShader::Вертекс,":/lightingVertexSh
    освещениеShaderProgram.addShaderFromSourceFile(QGLShader::Фрагмент,":/lightingFragme
    освещениеShaderProgram.link();

    кубВершины<<QVector3D(-0,5,-0,5,                0,5)<<QVector3D(0,5,-0,5, 0,5)<<        0,5)<<QVecto
                <<QVector3D(0,5,0,5,                QVector3D(-0,5,0,5,                     0,5)<<QVecto
                <<QVector3D(0,5,-0,5,-0,5)<<QVector3D(-0,5,-0,5,-0,5)<<QVecto <<QVector3D(-0,5,0,5,-0,5)<<
                QVector3D(0,5,0,5,-0,5)<<QVecto <<QVector3D(-0,5,-0,5,-0,5)<<QVector3D(-0,5,-0,5, <<
                QVector3D(-0,5,0,5, <<QVector3D(0,5,-0,5, <<QVector3D(0,5, <<QVector3D(-0,5,-0,5)<<QVecto3D(
                0,5,                0,5)<<QVector3D(-0,5,0,5,-0,5)<<QVecto 0,5)<<QVector3D(0,5
                                    ,-0,5,-0,5)<<QVecto 0,5,-0,5)<<QVector3D(0,5, 0,5,0,5)<<
                                    QVector3D(0,5, 0,5,-0,5)<<QVector3D(-0,5,0,5,       0,5)<<QVecto
                                                                0,5,    0,5)<<QVecto
                                                                0,5,-0,5)<<QVecto
                <<QVector3D(-0,5,-0,5,-0,5)<<QVector3D(0,5,-0,5,-0,5)<<QVecto <<QVector3D(0,5,-0,5,
                                    0,5)<<QVector3D(-0,5,-0,5, 1)<<         0,5)<<QVecto
    кубНормалы<<QVector3D(0,        0,      QVector3D(0, 1)<<QVector3D(0,   1)<<QVector3D(0, 1)<<      0,
                <<QVector3D(0, <<   0,                      0,      QVector3D(0,                    0,
                QVector3D(0, <<     0,-1)<<QVector3D(0, 0,-1)<<     0,-1)<<QVector3D(0, 0,-1)<<     0,-
                QVector3D(0, <<     QVector3D(0, 0,                 QVector3D(0, 0,                 0,-
                QVector3D(-1, <<        0)<<QVector3D(-1, 0)<<          0)<<QVector3D(-1, 0)<<      0,
                QVector3D(-1, <<    0,  QVector3D(-1, 0)<<QVector3D(    QVector3D(-1, 0)<<QVector3D(
                QVector3D(1, <<     0,  1, 0)<<QVector3D(1, 0)<<    0,  1, 0)<<QVector3D(1, 0)<<    0,
                QVector3D(1, <<     0,  QVector3D(0, 0)<<QVector3D(0,   QVector3D(0, 0)<<QVector3D(0,
                QVector3D(0, <<     1,  , 0)<<QVector3D(0,-1, 0)<<  1,  , 0)<<QVector3D(0,-1, 0)<<  1,
                QVector3D(0, <<     1,  QVector3D(0,-1,             1,  QVector3D(0,-1,             1,
                QVector3D(0,-1, <<
                QVector3D(0,-1,
    кубТекстураКоординаты<<QVector2D(0,0)<<QVector2D(1,0)<<QVector2D(1,1)//Ф
                        <<QVector2D(1,1)<<QVector2D(0,1)<<QVector2D(0,0) <<QVector2D(0,0)<<
                        QVector2D(1,0)<<QVector2D(1,1)//Б <<QVector2D(1,1)<<QVector2D(0,1)<<
                        QVector2D(0,0) <<QVector2D(0,0)<<QVector2D(1,0)<<QVector2D(1,1)//л <<
                        QVector2D(1,1)<<QVector2D(0,1)<<QVector2D(0,0) <<QVector2D(0,0)<<
                        QVector2D(1,0)<<QVector2D(1,1)//р <<QVector2D(1,1)<<QVector2D(0,1)<<
                        QVector2D(0,0) <<QVector2D(0,0)<<QVector2D(1,0)<<QVector2D(1,1)//Т <<
                        QVector2D(1,1)<<QVector2D(0,1)<<QVector2D(0,0) <<QVector2D(0,0)<<
                        QVector2D(1,0)<<QVector2D(1,1)//Б <<QVector2D(1,1)<<QVector2D(0,1)<<
                        QVector2D(0,0);



    кубТекстура"="bindTexture(QPixmap(":/cubeTexture.png"));

    coloringShaderProgram.addShaderFromSourceFile(QGLShader::Вертекс,":/coloringVertexSh
    coloringShaderProgram.addShaderFromSourceFile(QGLShader::Фрагмент,":/coloringFragme
    раскраскаShaderProgram.link();

    прожекторВершины<<QVector3D(        0,      1,      0)<<QVector3D(-0,5, 0)<<        0,0,5)<<вопрос
                    <<QVector3D(        0,      1,      QVector3D(0,5, 0)<<             0,-0,5)<<вопрос
                    <<QVector3D(        0,      1,      QVector3D(-0,5, 0)<<            ,-0,5)<<вопрос 0,
                    <<QVector3D(        0,      1,      QVector3D(0,5,                  0,5)<<вопрос
                    <<QVector3D(-0,5, <<    0,-0,5)<<QVector3D(0,5, 0,0,5)<<            0,-0,5)<<вопрос
                    QVector3D(0,5,          QVector3D(-0,5,                             ,0,5)<<вопрос
```

```
    прожекторЦвета<<QVector3D(0,2,0,2,0,2)<<QVector3D(0,2,0,2,0,2)<<QVector3D
                  <<QVector3D(0,2,0,2,0,2)<<QVector3D(0,2,0,2,0,2)<<QVector3D <<QVector3D(0,2,0,2,0,2)<<
                  QVector3D(0,2,0,2,0,2)<<QVector3D <<QVector3D(0,2,0,2,0,2)<<QVector3D(0,2,0,2,0,2)<<
                  QVector3D <<QVector3D(
                                  1,    1,    1)<<QVector3D( 1,    1,    1,    1)<<QVector3D 1,
                  <<QVector3D(    1,          1)<<QVector3D(    1,          1)<<QVector3D
}
```

После очистки экрана и расчета матрицы вида (одинаковой для обоих объектов) в *ГлВиджет::painGL()*Сначала мы визуализируем куб с помощью шейдеров освещения, а затем визуализируем прожектор с помощью шейдера окраски.

Поскольку мы хотим, чтобы начало координат куба совпадало с началом координат мира, мы оставляем матрицу модели (*mMatrix*) установлен в единичную матрицу. Затем мы вычисляем матрицу представления модели, которую нам также необходимо отправить в вершинный шейдер освещения, и извлекаем матрицу нормалей с помощью Qt.*QMatrix4x4::normal()*метод. Как мы уже говорили, эта матрица преобразует нормали поверхности нашего куба из координат модели в координаты средства просмотра. После этого вычисляем положение источника света в мировых координатах по углу.

Теперь мы можем визуализировать куб. Мы связываем программу шейдера освещения, устанавливаем униформы и текстурные блоки, устанавливаем и включаем массивы атрибутов, запускаем рендеринг, а затем отключаем массивы атрибутов и выпускаем программу. Для свойств источника света и материала мы устанавливаем значения, которые придают поверхности глянцевый вид.

Далее мы визуализируем центр внимания.

Поскольку мы хотим переместить прожектор в то же место, что и источник света, нам необходимо изменить матрицу его модели. Сначала мы восстанавливаем единичную матрицу (на самом деле мы раньше не изменяли матрицу модели, поэтому она все равно установлена на единичную матрицу). Затем перемещаем прожектор в положение источников света. Теперь мы все еще хотим повернуть его, поскольку он выглядит лучше, если смотреть на наш куб. Поэтому мы применяем две матрицы вращения сверху. Поскольку пирамида, представляющая наше световое пятно, все еще слишком велика, чтобы хорошо вписаться в нашу сцену, мы уменьшаем ее до десятой части ее исходного размера.

Теперь мы снова следуем обычной процедуре рендеринга, на этот раз используя*раскраскаShaderProgram* и данные прожектора. Благодаря тестированию глубины новый объект будет легко интегрирован в существующую сцену.

```
пустотаГлвиджет::краскаGL() {

    . . .

    mMatrix.setToIdentity();

    QMatrix4x4 мвМатрикс;
    МВМатрикс"="vMatrix*мМатрикс;

    QMatrix3x3 нормальная матрица; нормальныйМатрица
    "="mvMatrix.normalMatrix();

    QMatrix4x4 LightTransformation;
    LightTransformation.rotate(lightAngle,0,1,0);

    QVector3D LightPosition"="светТрансформация*QVector3D(0,1,1);

    освещениеShaderProgram.bind();
```

```
        освещениеShaderProgram.setUniformValue("мвпматрица", pМатрица*МВМатрикс);
        освещениеShaderProgram.setUniformValue("мвМатрикс", мвМатрикс);
        освещениеShaderProgram.setUniformValue("нормальная матрица", нормальная матрица);
        освещениеShaderProgram.setUniformValue("световая позиция", vMatrix*СветПозиция);

        освещениеShaderProgram.setUniformValue("ambientColor", QЦвет(32,32,32));
        освещениеShaderProgram.setUniformValue("диффузный цвет", QЦвет(128,128,128));
        освещениеShaderProgram.setUniformValue("зеркальный цвет", QЦвет(255,255,255));
        освещениеShaderProgram.setUniformValue("ambientReflection", (GLfloat)1.0);
        освещениеShaderProgram.setUniformValue("диффузное отражение", (GLfloat)1.0);
        освещениеShaderProgram.setUniformValue("зеркальное отражение", (GLfloat)1.0);
        освещениеShaderProgram.setUniformValue("блеск", (GLfloat)100,0);
        освещениеShaderProgram.setUniformValue("текстура",0);

        glActiveTexture (GL_TEXTURE0); glBindTexture
        (GL_TEXTURE_2D, CubeTexture); glActiveTexture(0);


        освещениеShaderProgram.setAttributeArray("вершина", CubeVertices.constData());
        освещениеShaderProgram.enableAttributeArray("вершина"); освещениеShaderProgram.setAttributeArray(
        "нормальный", CubeNormals.constData()); освещениеShaderProgram.enableAttributeArray("нормальный");
        освещениеShaderProgram.setAttributeArray("Координата текстуры", координаты кубатекстуры.
        освещениеShaderProgram.enableAttributeArray("Координата текстуры");


        glDrawArrays(GL_TRIANGLES,0, CubeVertices.size());

        освещениеShaderProgram.disableAttributeArray("вершина");
        освещениеShaderProgram.disableAttributeArray("нормальный");
        освещениеShaderProgram.disableAttributeArray("Координата текстуры");

        освещениеShaderProgram.release();

        mMatrix.setToIdentity();
        mMatrix.translate(lightPosition);
        mMatrix.rotate(lightAngle,0,1,0);
        mMatrix.rotate(45,1,0,0); mMatrix.scale(0,1);


        раскраскаShaderProgram.bind();

        coloringShaderProgram.setUniformValue("мвпматрица", pМатрица*vMatrix*мМатрикс);

        coloringShaderProgram.setAttributeArray("вершина", SpotlightVertices.constData());
        coloringShaderProgram.enableAttributeArray("вершина");

        coloringShaderProgram.setAttributeArray("цвет", SpotlightColors.constData());
        coloringShaderProgram.enableAttributeArray("цвет");

        glDrawArrays(GL_TRIANGLES,0, SpotlightVertices.size());

        coloringShaderProgram.disableAttributeArray("вершина");

        coloringShaderProgram.disableAttributeArray("цвет");

        раскраскаShaderProgram.release();
}
```

Последнее, что осталось сделать, это инициализировать положение источника света и настроить таймер. Мы говорим таймеру периодически вызывать *таймаут()* слот.

```
Глвиджет::GlWidget(QWidget*родитель)
    :QGLWidget(QGLFormat(/* Дополнительные параметры формата */), родитель)
{
    . . .

    светУгол"="0;

    QTimer*таймер"="новыйQTimer(этот);
    подключиться(таймер, СИГНАЛ(таймаут()),этот, СЛОТ(тайм-аут()));
    таймер->начинать(20);
}
```

В этом слоте мы обновляем угол обращения источника света и обновляем экран. Также мы убираем звонки на *QGLWidget::updateGL()* в обработчиках событий.

```
пустотаГлвиджет::таймаут() {

    светУгол+=1; пока(светУгол>=360)
    {
        светУгол"="360;
    }

    обновитьГЛ();
}
```

Теперь мы закончили реализацию. Если вы соберете и запустите программу, вы увидите освещенный текстурированный куб.

3.6 Буферный объект

До сих пор мы передавали все данные по вершинам объектов из оперативной памяти компьютера через шину памяти и шину AGP на видеокарту всякий раз, когда нам нужно было повторно отрисовать сцену. Очевидно, что это не очень эффективно и приводит к значительному снижению производительности, особенно при работе с большими наборами данных. В этом примере мы решим эту проблему, добавив *объекты буфера вершин* к примеру освещения.

Примечание:Исходный код, относящийся к этому разделу, находится в *примеры/буферные объекты/* каталог

Объекты буфера — это массивы данных общего назначения, находящиеся в памяти видеокарты. После того, как мы выделили его пространство и заполнили его данными, мы можем повторно использовать его на разных этапах конвейера рендеринга. Это означает чтение и запись в него. Мы также можем перемещать эти данные. Все эти операции ничего не требуют от процессора.

Существуют разные типы буферных объектов для разных целей. Наиболее часто используемый объект буфера — это объект буфера вершин, который служит источником массивов вершин.

В этом примере мы намерены использовать один буфер вершин для каждого объекта (т. е. один буфер вершин для куба и один буфер вершин для прожектора), в котором атрибуты плотно упакованы рядом с

друг друга в памяти. Мы не ограничены использованием одного буфера вершин для всех атрибутов. В качестве альтернативы мы могли бы также использовать один буфер вершин для каждой вершины или их комбинацию. Обратите внимание, что мы также можем смешивать использование массивов вершин и буферов вершин в одном рендеринге.

Вместо использования вызовов API OpenGL мы используем *QGLBuffer* класс для управления буферами вершин куба и прожектора. Тип объекта буфера можно установить в конструкторе. По умолчанию это буфер вершин.

Мы добавляем *QGLBuffer* для каждого объекта, удалите массивы вершин, которые мы использовали в предыдущей версии примера освещения, и добавьте переменные для хранения количества вершин, которые будут необходимы, чтобы сообщить OpenGL количество вершин для рендеринга в *ГлВиджет::updateGL()* метод.

```
сортГлвиджет:общественныйQGLWidget {

    частный:
    . . .

    QGLBuffer кубБуфер;
    . . .

    интервалnumSpotlightVertices;
    QGLBuffer прожекторBuffer; . . .

};
```

Объекты буфера — это объекты OpenGL, такие же, как программы шейдеров и текстуры, которые мы уже использовали в предыдущих примерах. Таким образом, синтаксис их обработки очень похож.

в *GlWidget::initializeGL()* метод, нам сначала нужно создать объект буфера. Это делается путем вызова *QGLBuffer::create()*. Он запросит идентификатор объекта свободного буфера (аналогично имени переменной) у видеокарты.

Затем, как и в случае с текстурами, нам нужно привязать его к контексту рендеринга, чтобы сделать его активным, используя *QGLBuffer::bind()*.

После этого мы вызываем *QGLBuffer::выделить()* чтобы выделить объем памяти, необходимый для хранения наших вершин, нормалей и координат текстур. Эта функция ожидает, что количество байтов будет зарезервировано в качестве параметра. Используя этот метод, мы также могли бы напрямую указать указатель на данные, которые мы хотим скопировать, но мы хотим расположить несколько наборов данных один за другим, поэтому мы выполняем копирование в следующих нескольких строках. Выделение памяти также возлагает на нас ответственность за освобождение этого пространства, когда оно больше не нужно, с помощью *QGLBuffer::destroy()*. Qt сделает это за нас, когда *QGLBuffer* объект уничтожен.

Загрузка данных на видеокарту осуществляется с помощью *QGLBuffer::write()*. Он считывает смещение (в байтах) от начала буферного объекта, указатель на данные в системной памяти, из которых необходимо прочитать, и количество байтов для копирования. Сначала копируем вершины кубов. Затем мы добавляем нормали к поверхности и координаты текстуры. Обратите внимание: поскольку OpenGL использует *GLfloats* для его вычислений нам необходимо учитывать размер типа с {GLfloat} при указании смещений и размеров памяти. Затем мы отвязываем объект буфера, используя *QGLBuffer::release()*.

Мы делаем то же самое для объекта прожектора.

```
пустотаГлвиджет::инициализироватьGL() {

    . . .

    numCubeVertices"="36;

    кубБуфер.создать();
    кубБуфер.бинд();
    CubeBuffer.allocate(numCubeVertices*(3+3+2)*размер(GLfloat));

    интервалкомпенсировать"="0;
    CubaBuffer.write(смещение, CubaVertices.constData(), numCubeVertices*3*размер(смещение GLfl+=
    numCubeVertices*3*размер(GLfloat);
    CubeBuffer.write(смещение, CubaNormals.constData(), numCubeVertices*3*размер(смещение GLflo+=
    numCubeVertices*3*размер(GLfloat);
    CubeBuffer.write(смещение, CubaTextureCoordinates.constData(), numCubeVertices*2*с

    кубБуфер.релиз();
    . . .

    numSpotlightVertices"="18;

    SpotlightBuffer.create();
    SpotlightBuffer.bind();
    SpotlightBuffer.allocate(numSpotlightVertices*(3+3)*размер(GLfloat));

    компенсировать"="0;
    CubeBuffer.write(смещение, SpotlightVertices.constData(), numSpotlightVertices*3*с компенсировать+=
    numSpotlightVertices*3*размер(GLfloat);
    CubeBuffer.write(смещение, SpotlightColors.constData(), numSpotlightVertices*3*размер

    SpotlightBuffer.release();
}
```

На всякий случай, если вам интересно, вот как бы работало создание буферных объектов, если бы мы не использовали Qt*QGLBuffer*класс для этой цели: мы бы позвонили*void glGenBuffers (GLsizei n, буферы GLuint)* запрашивать*н*количество буферных объектов с их идентификаторами, хранящимися в*буферы*. Далее мы свяжем буфер, используя*void glBindBuffer (цель перечисления, имя буфера uint)*, где мы также указываем тип буфера. Тогда мы бы использовали*void glBufferData (цель перечисления, размер sizeiptr, const void *data, использование перечисления)*чтобы загрузить данные. Перечисление под названием*Применение*определяет способ использования буфера основной программой, работающей на ЦП (например, только запись, только чтение и только копирование), а также частоту использования буфера для поддержки оптимизации. *void glDeleteBuffers (GLsizei n, const GLuint *buffers)*— это функция API OpenGL для удаления буферов и освобождения их памяти.

Чтобы OpenGL использовал наши объекты буфера вершин в качестве источника своих атрибутов вершин, нам нужно установить их по-другому в*ГлВиджет::updateGL()*метод.

Вместо того, чтобы звонить*QGLShaderProgram::setAttributeArray()*, нам нужно позвонить *QGLShaderProgram::setAttributeBuffer()*с*QGLBuffer*экземпляр, привязанный к контексту рендеринга. Параметры*QGLShaderProgram::setAttributeBuffer()*такие же, как у*QGLShader-Program::setAttributeArray()*. Нам нужно только адаптировать параметр смещения, чтобы однозначно идентифицировать расположение данных, поскольку теперь мы используем один большой участок памяти для каждого атрибута вместо одного массива для каждого из них.

```
пустотаГлвиджет::краскаGL() {

    . . .

    кубБуфер.бинд();
    интервалкомпенсировать"="0;
    освещениеShaderProgram.setAttributeBuffer("вершина", GL_FLOAT, смещение,3,0);
    освещениеShaderProgram.enableAttributeArray("вершина"); компенсировать+=numCubeVertices*3*размер
    (GLfloat); освещениеShaderProgram.setAttributeBuffer("нормальный", GL_FLOAT, смещение,3,0);
    освещениеShaderProgram.enableAttributeArray("нормальный"); компенсировать+=numCubeVertices*3*
    размер(GLfloat); освещениеShaderProgram.setAttributeBuffer("Координата текстуры", GL_FLOAT, смещение
    2,0 освещениеShaderProgram.enableAttributeArray("Координата текстуры"); кубБуфер.релиз();



    . . .

    glDrawArrays(GL_TRIANGLES,0, numCubeVertices);

    SpotlightBuffer.bind();
    компенсировать"="0;
    coloringShaderProgram.setAttributeBuffer("вершина", GL_FLOAT, смещение,3,0);
    coloringShaderProgram.enableAttributeArray("вершина"); компенсировать+=
    numSpotlightVertices*3*размер(GLfloat); coloringShaderProgram.setAttributeBuffer("цвет",
    GL_FLOAT, смещение,3,0); coloringShaderProgram.enableAttributeArray("цвет");
    SpotlightBuffer.release();


    glDrawArrays(GL_TRIANGLES,0, numSpotlightVertices); . . .

}
```

Рендеринг сцены теперь требует меньшего использования ЦП, и данные атрибутов больше не передаются повторно из системной памяти на видеокарту. Хотя в этом небольшом примере это может быть не заметно, это определенно повышает скорость программ, в которых задействовано больше геометрических данных.

ГЛАВА4

Заключение и дальнейшее чтение

Мы надеемся, что вам понравился этот урок и что мы пробудили у вас еще больший интерес к OpenGL и 3D-программированию. Если вы хотите глубже углубиться в OpenGL, вам следует обязательно подумать о том, чтобы приобрести хорошую книгу, посвященную этой теме.Домашняя страница OpenGL[1]содержит довольно много рекомендаций. Если вы ищете подход еще более высокого уровня, вы можете рассмотреть возможность взглянуть наQt/3D[2]и/илиQtQuick3D[3].

Поскольку OpenGL способен очень быстро вычислять большой объем информации, у вас могут возникнуть мысли об использовании его не только для компьютерной графики. Структура, основанная на этом подходе, называется *OpenCL*(которым также управляет Khronos Group Inc.). Для этой платформы даже существует модуль Qt. Это называетсяQtOpenCL[4].

Использованная литература:

- *http://www.opengl.org*-Домашняя страница OpenGL
- *http://www.khronos.org/opengl*-Домашняя страница Khronos Group Inc., посвященная OpenGL
- *http://www.khronos.org/opengles*-Домашняя страница Khronos Group Inc., посвященная OpenGL ES.
- *http://doc-snapshot.qt-project.org/qt3d-1.0*-Справочная документация Qt/3D
- *http://doc.qt.digia.com/qt-quick3d-snapshot*-Справочная документация QtQuick3D

[1]http://www.opengl.org
[2]http://doc-snapshot.qt-project.org/qt3d-1.0
[3]http://doc.qt.digia.com/qt-quick3d-snapshot
[4]http://doc.qt.digia.com/opencl-snapshot/index.html