Qt и ваша видеокарта

Сегодня речь пойдет о том, как, используя дополнительные возможности видеокарты, решить различные графические и неграфические задачи. Примеры из этой главы будут работать у вас только в том случае, если ваша видеокарта и драйвер поддерживают аппаратное ускорение OpenGL. На это предупреждение должны обратить особое внимание пользователи Linux и компьютеров со встроенными графическими чипсетами.

Графическая система *Arthur*

Хотя изменения графической системы Qt 4 по сравнению с Qt 3 не очень заметны на первый взгляд, новая система предоставляет весьма широкие возможности. Старые классы **`QPainter`** и **`QPaintDevice`** остались, но к ним добавился абстрактный класс **`QPaintEngine`**. Теперь все функции, реализующие специфику графического вывода на разных устройствах, собраны в классах-потомках **`QPaintEngine`**, соответствующих этим устройствам. Классы **`QPainter`** и **`QPaintDevice`** используют методы **`QPaintEngine`** для доступа к графическим устройствам, а не обращаются к этим устройствам напрямую, как раньше. Программисту же, наоборот, не придется иметь дело с потомками **`QPaintEngine`**, если только он не захочет расширить функциональность Qt 4, реализовав графический вывод на каком-нибудь неподдерживаемом устройстве. На практике это означает, что «сладкая парочка» **`QPainter`** и **`QPaintDevice`** теперь может рисовать практически на всех графических устройствах, доступных на данной платформе, причем работа с разными устройствами, будь то принтер или окно OpenGL, в значительной степени унифицирована. Еще одно преимущество новой системы заключается в том, что многие графические операции, которые раньше были реализованы чисто программными средствами, теперь могут использовать аппаратное ускорение и другие функции, поддерживаемые «железом» (раньше это было невозможно потому, что между **`QPainter`** и устройством лежала «прослойка» эмулятора растрового массива). Например, класс **`QGLWidget`**, работающий с OpenGL, теперь поддерживает операции класса **`QPainter`**, используя, где возможно, аппаратное ускорение.

Новая архитектура позволила добавить новые возможности, среди которых стоит отметить градиентные кисти (заполняющие фигуры заданным градиентом) и поддержку альфа-канала, которая позволяет создавать полупрозрачные изображения.

Посмотрим, как все это работает на практике. В качестве примера рассмотрим визуальный компонент **`QGLWidget`**. Благодаря новой архитектуре мы можем создавать изображения средствами **`QPainter`** в рабочем окне **`QGLWidget`** точно так же, как и на поверхности любого другого виджета. Не могу не отметить некоторую диалектичность процесса: когда-то я демонстрировал, как выводить графику OpenGL на поверхности объекта-потомка **`QWidget`**. Теперь мы воспользуемся **`QGLWidget`** для вывода «обычных» изображений, которые не являются частью трехмерных сцен. Такое использование OpenGL «не по назначению» отражает популярную в последнее время тенденцию – задействовать мощь 3D-ускорителей в традиционно не-трехмерных задачах, например при отрисовке окон или работе с растровыми картинками.

Самый простой способ добавить в программу сцену, которая отрисовывается средствами OpenGL – создать класс-потомок **`QGLWidget`**. В простейшем случае у дочернего класса достаточно перекрыть метод **`paintEvent()`**, который, как мы знаем, перерисовывает виджет всякий раз, когда виджет в этом нуждается.

В Qt 4, вплоть до версии 4.7 центральным виджетом программ, использующих OpenGL, обычно является виджет **`QGLWidget`**. В будущем ситуация может измениться, о чем подробно сказано в конце этой главы. Обычно мы создаем класса-потомка этого класса, в котором перекрываем некоторые методы класса **`QGLWidget`**.

Рассмотрим метод **paintEvent**() класса **GLWidget** (который является потомком **QGLWidget**) в программе arthur-demo (вы найдете исходные тексты в папке arthur-demo).

Листинг 1. Метод `paintEvent()` ***класса*** `GLWidget`

```
void GLWidget::paintEvent(QPaintEvent *event)
{
        QPainter painter;
        QPen pen;
        painter.begin(this);
        painter.setRenderHint(QPainter::HighQualityAntialiasing);
        painter.eraseRect(QRect(0, 0, width(), height()));
        painter.drawPixmap(0, 0, width(), height(), *pixmap);
        painter.translate(width()/2, height()/2);
        painter.rotate(angle);
        painter.translate(-width()/2, -height()/2);
        pen.setColor(QColor(0, 127, 0));
        pen.setWidth(4);
        painter.setPen(pen);
        painter.drawLine(0, 0, width(), height());
        pen.setColor(QColor(255, 0, 0));
        painter.setPen(pen);
        painter.drawLine(0, height(), width(), 0);
        painter.setBrush(QColor(255, 0, 0, 127));
        painter.drawRect(0, 0, width()/2, height());
        painter.setBrush(QColor(0, 0, 255, 127));
        painter.drawRect(0, 0, width(), height()/2);
        painter.setBrush(QColor(0, 255, 0, 127));
        painter.drawRect(width()/2, 0, width()/2, height());
        painter.end();
        event->accept();
}
```

Прежде всего, обратите внимание на то, что хотя виджет **GLWidget** наследует **QGLWidget**, мы пользуемся исключительно методами двухмерного рисовальщика **QPainter**. Рисование начинается с вызова метода **begin()** объекта **painter**. Аргументом этого метода должен быть указатель на объект **QPaintDevice**, к каковому типу теперь приводится и объект класса **QGLWidget** (и, естественно, его потомки). Останавливаться на каждой инструкции вывода графики мы не будем. Обращаю внимание читателя только на некоторые «продвинутые» способности **painter**.

При отрисовке сцены мы включаем поддержку сглаживания контуров с помощью вызова метода **setRenderHint()**. Этому методу передается константа **QPainter::HighQualityAntialiasing**, которая определяет наиболее качественное сглаживание. С помощью метода **setRenderHint()** можно настроить и другие тонкие параметры вывода графики, например, сглаживание текста или сглаживание при преобразовании растровых изображений.

Помимо настройки сглаживания мы используем смешивание цветов – alpha blending (обратите внимание на четвертый аргумент конструктора класса **QColor()**). Наконец, чтобы сцена была анимированной, мы используем методы трансформации **translate()** и **rotate()**. Сглаживание, смешивание и геометрические преобразования могут выполняться с использованием аппаратной поддержки (например, поддержки 3D-ускорителя), если аппаратная поддержка включена в вашей системе.

В приведенном выше примере специально не использовались функции OpenGL, чтобы показать, что в графической системе Arthur можно задействовать

возможности OpenGL, не используя сами команды OpenGL (например, воспользоваться методом **eraseRect()** вместо **glClearColor()**). В результате один и тот же код может быть задействован для вывода графики в окне с помощью 3D-ускорителя, для записи графики в растровые форматы хранения изображений или для вывода изображений на принтер PostScript.

Примечание

Впрочем, не все так просто. Если вы захотите повторно использовать тот же самый код **GLWidget::paintEvent()**, например, для вывода на печать, а не просто скопировать его другой класс, то ни метод **QPainter::setRedirected()**, ни более продвинутый метод **QWidget::render()** вам не помогут, так как ни тот ни другой не умеют работать с графикой OpenGL. В лучшем случае вы получите пресловутый черный квадрат.

Если вы хотите вывести на принтер именно картинку, отображаемую средствами OpenGL, придется пойти в обход. Сначала, с помощью метода **grabFrameBuffer()**, объявленного в классе **QGLWidget**, получаем объект **QImage**, который содержит снимок графической сцены OpenGL. Теперь этот объект можно выводить на печать, например в псевдо принтер, «печатающий» в файлы PDF (листинг 2).

Листинг 2. Сохранение вывода виджета-потомка `QGLWidget` в формате PDF

```
QImage img = glWidget->grabFrameBuffer(true);
QPrinter printer;
printer.setOutputFormat(QPrinter::PdfFormat);
printer.setOutputFileName("D:\\pdfs\\page1.pdf");
QPainter painter;
painter.begin(&printer);
painter.drawImage(0, 0, img);
painter.end();
```

Вернемся теперь немного назад и посмотрим на конструктор класса **GLWidget** (листинг 3).

Листинг 3. Конструктор класса `GLWidget`

```
GLWidget::GLWidget(QWidget *parent) : QGLWidget(parent)
{
    angle = 0;
    pixmap = new QPixmap();
    pixmap->load("mashrooms.png");
    if (pixmap->isNull())
        qWarning() << QString::fromUtf8("Файл 'mashrooms.png' не
загружен.");
}
```

Здесь мы загружаем графический файл в объект **QPixmap**. Если загрузить файл не удалось (метод **isNull()** возвращает **true**), мы выводим предупреждающее сообщение на отладочную консоль. Функция **qWarning**() – одна из семейства функций, предназначенных для того, чтобы сообщить вам, что в программе пошло что-то не так. Для того чтобы использовать синтаксис вызова **qWarning()** с оператором **<<**, необходимо включить заголовочный файл **<QDebug>** в исходный текст. Qt Creator на всех платформах перехватывает все сообщения, которые графическая программа выводит на консоль. Запуская программу из-под Eclipse (при использовании инструментария GNU) вы так же может увидеть эти сообщения. Наконец, все сообщения, выводимые на консоль программой и ее компонентами, можно прочитать, если программа запущена из консоли Linux. А вот при отладке программы в среде Microsoft Visual Studio эти сообщения проще всего увидеть, если сначала запустить специальную консоль **Visual Studio Command Prompt**.

Для анимации изображения мы используем встроенный таймер класса QMainWindow (листинг 4).

Листинг 4. Анимация сцены OpenGL с помощью таймера

```
void OGLWindow::timerEvent(QTimerEvent *event)
{
    glWidget->setAngle(glWidget->getAngle()+0.1);
    glWidget->update();
}
```

Методы **`setAngle()`** и **`getAngle()`** позволяют получить доступ к переменной **`angle`**, которую мы используем в методе **`GLWidget::paintEvent()`**. В результате получаем анимированное изображение с аппаратно ускоренными анимацией, сглаживанием и альфа-каналом (рис. 1).

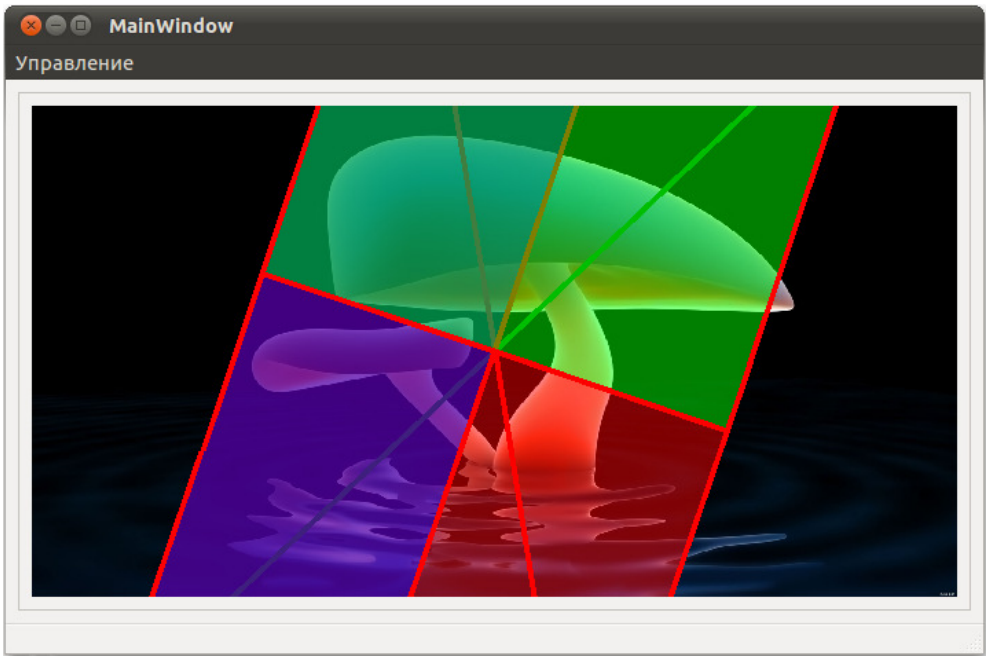

Рис. 1. Программа, демонстрирующая возможности Arthur, в действии

Еще один практически полезный пример, где аппаратное ускорение OpenGL может помочь – работа с очень большими изображениями. Имеются в виду изображения, разрешение которых многократно превышает разрешение монитора. Задача работы с такими изображениями возникает в геоинформационных приложениях, приложениях обработки сканированных изображений и цифровой фотографии. Программа scalefast, которую вы найдете в папке Ch8/scalefast, демонстрирует возможности аппаратно ускоренного масштабирования и прокручивания больших изображений. Для изменения масштаба изображения пользуйтесь клавишами «плюс» или «минус», а для прокрутки – клавишами стрелок (рис. 2).

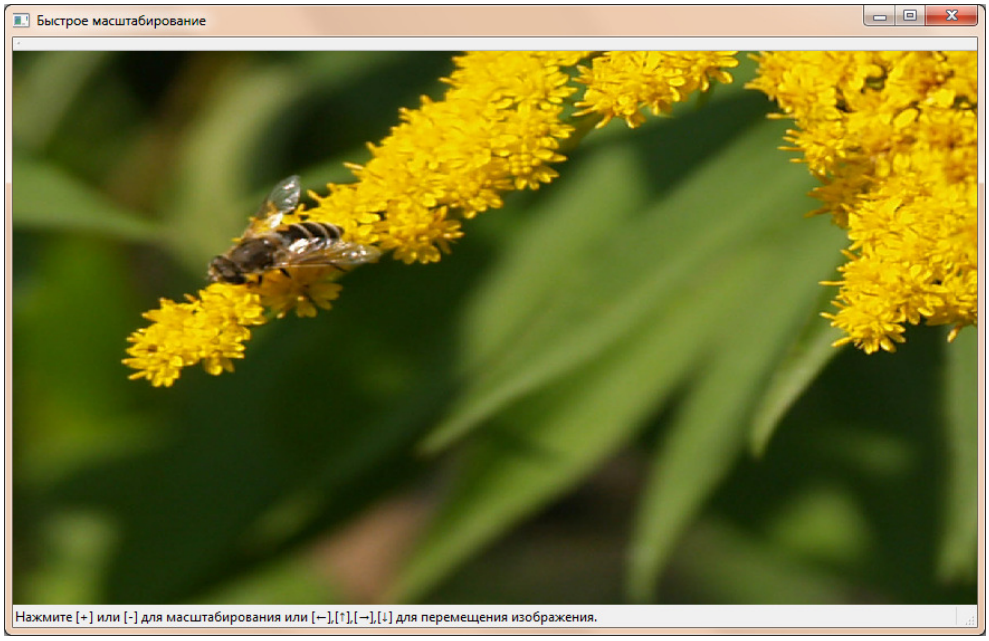

Рис. 2. Программа, scalefast

Для вывода мы изображения используем метод **`drawPixmap()`** класса **`QPainter`**, в той его версии, которая позволяет скопировать прямоугольную область объекта **`QPixmap`** в буфер устройства ввода-вывода.

Аппаратное поддержка масштабирования и прокрутки изображения действительно может ускорить эти операции, особенно в операционной системе Linux, однако у нее есть ограничение, связанное с максимально допустимым размером изображения, которое можно загрузить в буфер OpenGL (имеется в виду размер исходного изображения, возможности масштабирования практически не ограничены). В моей системе максимальный размер изображения составляет 4098*4098 пикселей, на других графических картах он, вероятно, может быть другим. И хотя указанный размер превышает разрешающую способность большинства мониторов и видеокарт, нам, конечно, хочется работать с изображениями еще большего размера. В следующей главе мы рассмотрим один из способов обхода указанного ограничения.

Класс *QGLContext*

Контекстом OpenGL называется набор переменных, поддерживающих внутреннее состояние OpenGL. Все команды OpenGL выполняются с учетом определенного контекста, но сам контекст при этом не указывается. Если в приложении используется несколько контекстов OpenGL, один из них должен быть выбран в качестве ткущего контекста. Объекты класса **`QGLContext`** помогают настроить множество параметров OpenGL, о которых вы прочтете в документации. Параметры контекста обычно задаются с помощью объекта класса **`QGLFormat`**. Важно знать, что у каждого объекта **`QGlWidget`** есть свой контекст OpenGL. Чтобы сделать этот контекст текущим, необходимо вызвать метод **`makeCurrent()`**.

Прежде чем перейти к рассмотрению конкретного примера, коснемся одного специфического вопроса, связанного с программированием OpenGL в Qt. Работая с потоками, мы никогда не обновляли содержимое графических объектов из вторичного потока. Причины этого очевидны, и главная из них – необходимость четко разделять доступ к критическому ресурсу, которым является графический объект. Так что до сих пор мы либо располагали весь код, обновляющий графический объект, в главном потоке, либо передавали главному потоку сообщения, в ответ на которые он должен был обновлять графические данные. Вторая модель, очевидно, не подходит при работе с графикой OpenGL, по крайней

мере, в большинстве случаев. Сцена OpenGL должна обновляться слишком часто, чтобы это обновление можно было поручить системе сообщений.

Начинающие программисты Qt, пишущие для Windows, почему-то часто используют для команд OpenGL отдельные потоки, которые выводят данные непосредственно в виджет OpenGL, принадлежащий главному потоку. Я бы не советовал делать этого без крайней необходимости. Если объект графической информации для вывода не очень велик, лучше расположить весь код обновления сцены в главном потоке. Помимо обычных причин, по которым мы стремимся не обновлять виджеты напрямую из вторичных потоков, в случае OpenGL есть еще одна причина.

Как уже отмечалось, сами по себе, команды OpenGL не привязываются ни к какому средству вывода (окну, виджету, и так далее). Команды OpenGL абстрагированы от конкретных компонентов вывода, управлением которыми занимается операционная система. Разные операционные системы делают это по-разному. В некоторых системах, например в Linux, результаты работы OpenGL привязываются к потоку, в котором выполняются команды OpenGL (строго говоря, в этом случае к потоку привязывается контекст OpenGL). В Windows наоборот, можно выполнять команды OpenGL в одном потоке, а графические результаты отображать в другом. Если вы хотите, чтобы ваша программа работала предсказуемо во всех поддерживаемых Qt операционных системах, лучше придерживаться строгих правил относительно обновления виджетов. Ну и наконец, если в процессе отрисовки сцены OpenGL не выполняется интенсивных расчетов (а если такие расчеты необходимы, они часто, могут быть возложены на графический процессор), то использование потоков мало что даст для ускорения отрисовки сцены даже на многоядерных процессорах. Поскольку отрисовка сцены OpenGL сводится, в основном, к посылке драйверу видеокарты относительно коротких сообщений, эта операция не очень сильно нагружает поток.

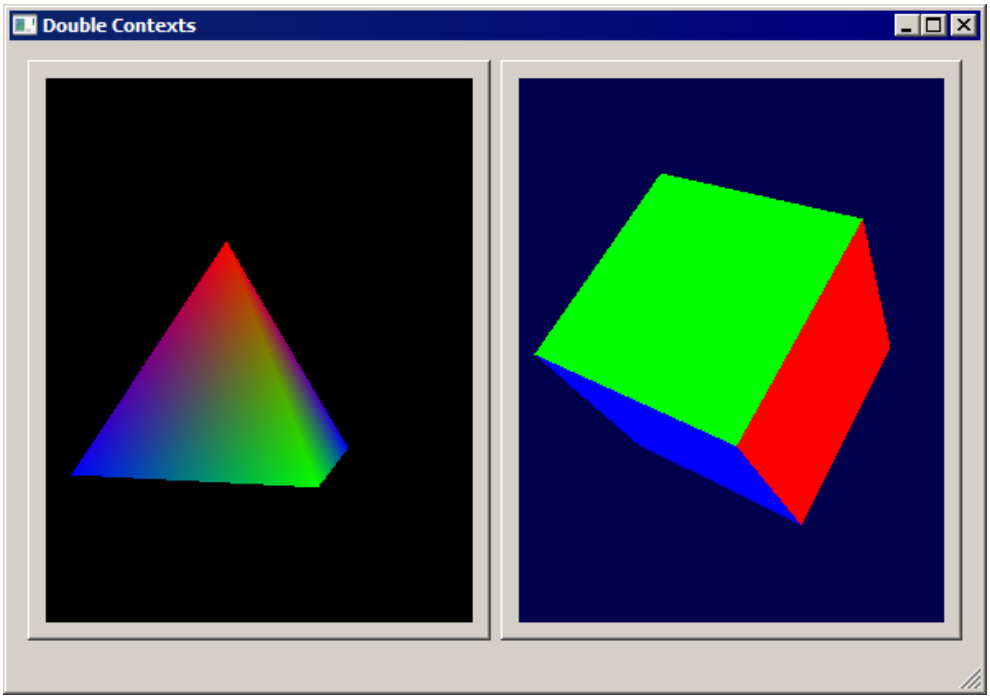

Рис. 3. Две сцены OpenGL в одной программе

Приложение doublecontexts (рис. 3) демонстрирует вывод двух независимых сцен OpenGL в два виджета Qt. Вы найдете эту программу в папке Ch8/doublecontexts. Поскольку в этом приложении мы имеем дело не с двухмерной графикой, ускоренной с помощью 3D-ускорителя, а с «настоящей» трехмерной графикой OpenGL, в виджете, в классе **`GLWidget`** необходимо выполнить весь стандартный цикл обработки графики OpenGL: инициализацию сцены OpenGL, отрисовку,

изменение размеров сцены. Для этих трех составляющих в классе **QGLWidget** объявлены специальные виртуальные функции, которые должны быть перекрыты в его потомках. Рассмотрим подробно один из двух виджетов, выполняющих вывод графики OpenGL (листинг 5).

Листинг 5. Класс *GLWidget1*

```
class GLWidget1 : public QGLWidget
{
    Q_OBJECT
public:
    explicit GLWidget1(QWidget *parent = 0);
    ~GLWidget1();
signals:
public slots:
protected:
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
private:
    double x;
};
```

Три метода, которые мы должны перекрыть в классе-потомке **QGLWidget**, это **initializeGL()**, **resizeGL()** и **paintGL()**. Первый из этих методов вызывается тогда, когда виджету необходимо инициализировать сцену OpenGL. За время жизни виджета этот метод может быть вызван больше одного раза, поэтому в нем не следует размещать код, который должен быть выполнен только один раз (такой код лучше разместить в конструкторе класса). Метод **resizeGL()** вызывается тогда, когда виджет меняет свои размеры. В этом методе выполняется традиционный код OpenGL, приноравливающий размеры области видимости сцены к размерам порта вывода. Наконец, метод **paintGL()** выполняет прорисовку сцены. Рассмотрим реализацию этих методов (листинг 6).

Листинг 6. Перекрытые методы в классе *GLWidget1*

```
void GLWidget1::initializeGL()
{
    glShadeModel(GL_SMOOTH);
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
}

void GLWidget1::resizeGL(int width, int height)
{
    glViewport(0,0,width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0f,(GLfloat)width/(GLfloat)height,0.1f,100.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void GLWidget1::paintGL()
{
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(-0.3f,0.0f,-6.0f);
    glRotatef(x, 0.0f, 1.0f, 0.0f);
    glBegin(GL_TRIANGLES);
    ...
    glEnd();
    x += 1.0;
}
```

Те, кто знаком с языком OpenGL, поймут, что именно делается в этих методах. Обращаю внимание на то, что хотя по умолчанию виджеты OpenGL использую двойную буферизацию, нам нет необходимости переключать буферы явным образом, если конечно, вы не отключите автоматическое переключение буферов с помощью метода **`setAutoBufferSwap()`**. В этом случае переключать буферы придется с помощью вызова метода **`swapBuffers()`**. Обратите внимание, что мы переключаем буферы методами виджета **`QGLWidget`**, а не командами OpenGL.

Анимация сцены, как и в предыдущем примере выполняется встроенным таймером главного окна (листинг 7).

Листинг 7. Перекрытые методы в классе `GLWidget1`

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    glw1 = new GLWidget1(ui->frame);
    glw2 = new GLWidget2(ui->frame_2);
    ...
    startTimer(50);
}

void MainWindow::timerEvent(QTimerEvent *)
{
    glw1->makeCurrent();
    glw1->updateGL();
    glw2->makeCurrent();
    glw2->updateGL();
}
```

Метод **`updateGL()`** заставляет виджет перерисовывать сцену.

Шейдеры OpenGL в Qt

Жизнь не стоит на месте, и, начиная с версии 4.6, библиотека Qt обзавелась поддержкой шейдеров OpenGL и OpenGL-ES. Qt предоставляет нам два класса, предназначенных для работы с шейдерами: **`QGLShaderProgram`** и **`QGLShader`**. Класс **`QGLShaderProgram`** управляет шейдерами, компилирует и связывает шейдерные программы, написанные на языке GLSL (OpenGL Shading Language). Этот класс так же позволяет обмениваться данными между нашими программами, написанными на Qt, и шейдерными программами. Класс **`QGLShaderProgram`** поддерживает работу с вершинными, геометрическими и фрагментными шейдерами. В качестве языка программирования используется GLSL или GLSL-ES. Фактическая компиляция шейдера того или иного поддерживаемого типа выполняется объектами класса **`QGLShader`**.

В качестве демонстрации поддержки шейдеров в Qt мы рассмотрим программу shader-demo, которая отображает квадрат и использует фрагментный шейдер для его раскраски (эту программу вы сможете найти в каталоге Ch8/shader-demo).

Как и в предыдущем примере, самым главным и самым интересным виджетом программы-демонстратора шейдеров будет виджет **GLShaderWidget**, являющийся потомком класса **QGLWidget**.

Имеет смысл рассмотреть подробнее класс GLShaderWidget демонстратора шейдеров (листинг 8).

Листинг 8. Перекрытые методы в классе *GLWidget1*

```
class GLShaderWidget : public QGLWidget
{
    Q_OBJECT
public:
    explicit GLShaderWidget(QWidget *parent = 0);
    ~GLShaderWidget();
signals:
public slots:
protected:
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
    void loadShader(const QString &vshader, const QString &fshader);
private:
    int frameCounter;
    QGLShaderProgram * shaderProgram;
    QGLShader * vertexShader, * fragmentShader;
    double x;
};
```

Помимо стандартных методов, которые необходимо перекрыть к классе-потомке **QGLWidget**, мы добавили еще один метод – **loadShader()**. Этот метод нужен нам просто для того, чтобы сделать код более читабельным. Задача метода **loadShader()** заключается в том, чтобы загрузить шейдерные программы, соответственно, для вершинного и фрагментного шейдеров из файлов шейдерных программ, написанных на языке GLSL. Имена этих файлов передаются методу в качестве аргументов. Мы так же добавили в определение класса переменные-указатели на объекты **QGLShaderProgram** и **QGLShader**. Начнем исследование класса с метода **loadShader()** (листинг 9).

Листинг 9. Перекрытые методы в классе *GLWidget1*

```
void GLShaderWidget::loadShader(const QString &vshader, const QString
&fshader)
{
    if(shaderProgram)
        {
        shaderProgram->release();
        shaderProgram->removeAllShaders();
        }
    else shaderProgram = new QGLShaderProgram;
    if(vertexShader)
        {
        delete vertexShader;
        vertexShader = NULL;
        }
    if(fragmentShader)
        {
        delete fragmentShader;
        fragmentShader = NULL;
```

```
        }
    QFileInfo vsh(vshader);
    if(vsh.exists())
        {
        vertexShader = new QGLShader(QGLShader::Vertex);
        if(vertexShader->compileSourceFile(vshader))
            shaderProgram->addShader(vertexShader);
        else qWarning() << QString::fromUtf8("Ошибка вершинного шейдера")
<< vertexShader->log();
        }
    QFileInfo fsh(fshader);
    if(fsh.exists())
        {
        fragmentShader = new QGLShader(QGLShader::Fragment);
        if(fragmentShader->compileSourceFile(fshader))
            shaderProgram->addShader(fragmentShader);
        else qWarning() << QString::fromUtf8("Ошибка фрагментного
шейдера") << fragmentShader->log();
        }
    if(!shaderProgram->link())
        {
        qWarning() << QString::fromUtf8("Ошибка компоновки шейдерной
программы") << shaderProgram->log();
        }
}
```

В начале своей работы метод **loadShader()** проверяет, не были ли уже созданы объекты для работы с шейдерами (в конструкторе класса **GLShaderWidget** необходимо присвоить значения **NULL** соответствующим указателям). Это связано с тем, что метод **loadShader()** вызывается из метода **initializeGL()**, а этот метод, как мы уже знаем, может быть вызван не один раз в ходе жизненного цикла виджета **GLShaderWidget**. Если объекты, необходимые для работы с шейдерами, уже созданы, незачем выполнять эту операцию повторно. Мы создаем вершинный и фрагментный шейдеры, указывая в конструкторах соответствующих объектов значения **QGLShader::Vertex** и **QGLShader::Fragment**. Магию преобразования программы на языке GLSL в скомпилированную шейдерную программу выполняет метод **compileSourceFile()** класса **QGLShader**. У этого метода есть «двойник» - метод **compileSourceCode()**, который компилирует шейдерную программу из исходного текста, переданного методу в качестве строки. При этом строка исходного текста может быть объектом **QString**, **QByteArray** или строкой в стиле C (массивом char с нулевым окончанием). Стоит отметить, что компилятор GLSL, используемый Qt, не принимает исходные тексты программ ни в каких кодировках, кроме восьмибитной. При использовании объекта **QString** текст будет автоматически преобразован в нужную кодировку, а вот при использовании объектов **QByteArray** вам самим придется позаботиться об этом. То же самое касается загрузки текста программ из файлов. Поскольку у загрузчика нет магического способа выяснения кодировки текста (для его последующего преобразования), файлы программ должны быть написаны в восьмибитной кодировке.

Связывание шейдерных программ и превращение их в исполнимый объект OpenGL выполняет объект класса **QGLShaderProgram**. Сначала мы передаем этому объекту объекты, представляющие шейдеры (с помощью метода **addShader()**). Затем мы выполняем компоновку шейдерных программ с помощью метода **link()** объекта этого класса. Перекрыв метод **link()**, мы можем инициализировать униформные переменные, но мы этого делать не будем, хотя в используемой нами программе фрагментного шейдера такая переменная есть (листинг 10).

Листинг 10. Программа для фрагментного шейдера

```
uniform vec3 myColor;
void main()
{
       gl_FragColor = vec4(myColor, 1.);
}
```

Как именно мы передаем значение этой переменной, будет показано ниже.

> ***Примечание***
> Мы не будем останавливаться подробно на языке GLSL, так же как мы не останавливались на описании OpenGL. Эти темы выходят за пределы данной книги. Тем более, что GLSL – не единственное средство, которой Qt предполагает использовать для программирования графического процессора.

В случае с шейдерными программами мы впервые (но не в последний раз) сталкиваемся с ситуацией, когда некоторая часть нашей программы компилируется во время выполнения самой программы. Это означает, что во время выполнения программы, использующей шейдеры, нам придется иметь дело как с ошибками времени выполнения, так и с ошибками времени компиляции, когда шейдерные программы просто не удается скомпилировать, в основном, из-за неверного синтаксиса. У классов **`QGLShader`** и **`QGLShaderProgram`** есть метод **`log()`**, который возвращает информацию о таких ошибках в текстовой форме.

Если шейдерные программы собраны успешно, наше приложение выводит на консоль сообщение

```
QGLShader::link: "Vertex shader(s) linked, fragment shader(s) linked."
```

Это сообщение создается самими классами Qt, вот почему оно не русифицировано.

Рассмотрим теперь остальные методы класса **`GLShaderWidget`** (листинг 11).

Листинг 11. Реализация методов класса *`GLShaderWidget`*

```
GLShaderWidget::GLShaderWidget(QWidget *parent) : QGLWidget(parent)
{
    makeCurrent();
    shaderProgram = NULL;
    vertexShader = NULL;
    fragmentShader = NULL;
}

GLShaderWidget::~GLShaderWidget()
{
    delete shaderProgram;
    delete vertexShader;
    delete fragmentShader;
}
void GLShaderWidget::initializeGL()
{
    loadShader(":/Basic.vsh", ":/Basic.fsh");
    shaderProgram->bind();
    x = 0;
    frameCounter = 0;
}

void GLShaderWidget::resizeGL(int width, int height)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.,((GLfloat)width)/((GLfloat)height),0.1f,1000.0f);
```

```
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void GLShaderWidget::paintGL()
{
    glLoadIdentity();
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(0.25f, 0.25f, 0.25f, 0.0f);
    frameCounter ++;
    glTranslatef(0.0f, 0.0f, -5.0f);
    glRotatef(frameCounter,0.0f,0.0f,0.5f);
    glBegin(GL_QUADS);
        glVertex3f(-1.0, -1.0,0.0);
        glVertex3f(1.0, -1.0,0.0);
        glVertex3f(1.0, 1.0,0.0);
        glVertex3f(-1.0, 1.0,0.0);
    glEnd();
    x += 0.1;
    shaderProgram->setUniformValue("myColor", sin(x), cos(x), 1.);
}
```

Начнем с конструктора. Поскольку виджет **GLShaderWidget** – единственный компонент, использующий графику OpenGL в нашей демонстрационной программе, мы делаем контекст OpenGL этого виджета текущим контекстом непосредственно в его конструкторе. Далее мы присваиваем значение **NULL** всем указателям на объекты, работающие с шейдерами. Делается это по причинам, описанным выше.

В методе **initializeGL()** мы создаем шейдеры с помощью метода **loadShader()**, а затем вызываем метод **bind()** созданного в **loaShader()** объекта **shaderProgram**. Этот метод связывает шейдерную программу и текущий контекст OpenGL. Между прочим, у класса **QGLShaderProgram** есть метод **programId()**, который возвращает идентификатор шейдерной программы, присвоенный ей OpenGL. Этот идентификатор можно использовать для связывания шейдерной программы с контекстом с помощью команды OpenGL **glUseProgram()**.

Класс **QGLShaderProgram** позволяет не только собрать шейдерную программу, но и обмениваться с ней значениями. В тексте программы для фрагментного шейдера была определена переменная **myColor**, имеющая тип **vec3**, то есть вектор из трех чисел с плавающей точкой. Для передачи значения униформной переменной используется метод **setUniformValue()**. Первый аргумент метода – имя переменной в шейдерной программе. Далее следуют значения, которыми мы инициализируем шейдерную переменную. У метода **setUniformValue()** 50 перегруженных вариантов (вероятно, этот метод является чемпионом Qt по числу перегрузок), каждый из которых соответствует определенному типу униформной переменной. Помимо метода **setUniformValue()** у класса **QGLShaderProgram** есть еще метод **setAttributeValue()**, позволяющий задавать значения атрибутов.

Если ваша система поддерживает шейдеры, то собрав программу shader-demo, вы увидите вращающийся квадрат постоянно меняющегося цвета (рис. 4).

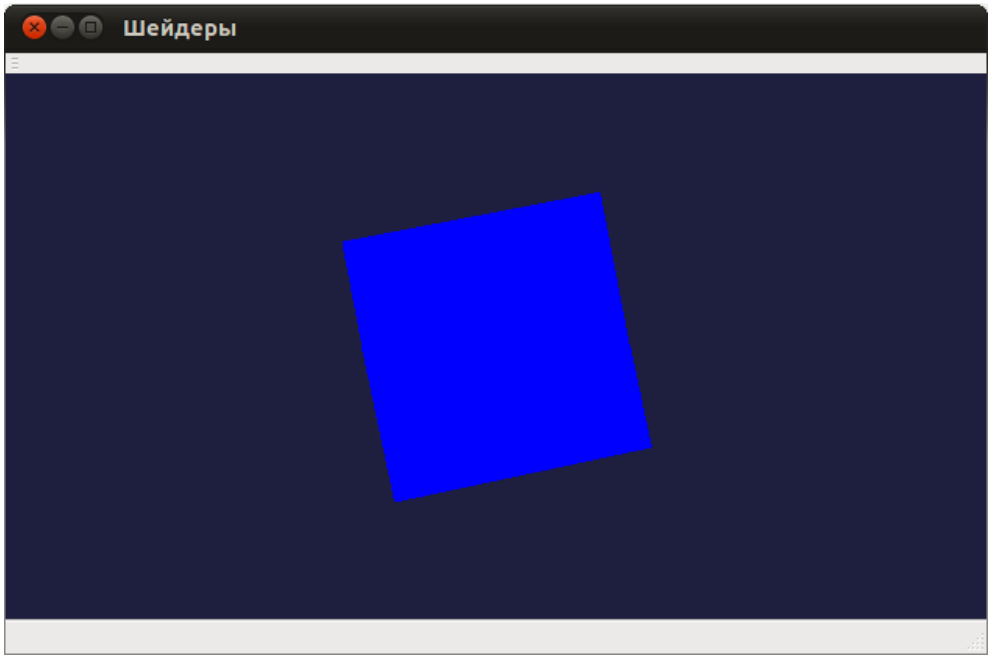

Рис. 4. Программа shader-demo

Взгляд в будущее

Обилие технологий, которые использует Qt, настолько велико, что ни одна книга, вероятно не способна описать их все (эта – точно не способна). Из этого факта следует несколько выводов. Во-первых, вряд ли в настоящее время даже среди сертифицированных разработчиков Qt найдется человек, который знает все технологии Qt. Во-вторых, многие серьезные программы могут быть написаны целиком на Qt, без привлечения сторонних библиотек. Если конечно вы сумеете отыскать в океане технологий Qt то, что вам нужно.

Уже сейчас мы можем познакомиться с подсистемой Qt/3D, которая войдет в будущие версии библиотеки Qt.

Подсистема Qt/3D

Если определить Qt/3D совсем просто, то это – надстройка над OpenGL/ES. Однако цель этой надстройки, немного, немало, принести на платформы OpenGL/ES те методы, к которым мы привыкли при работе с двухмерной графикой. В некотором смысле задачи Qt/3D противоположны задачам Arthur. В Arthur мы использовали трехмерную «подложку» для ускорения и расширения возможностей двухмерных графических операций. В Qt/3D технологии, отработанные в области двухмерной графики переносятся в трехмерный мир. Помимо этого, Qt/3D позволяет поддерживать ряж устройств, с которыми раньше нельзя было работать средствами Qt, например, стерео-очки и трехмерные мыши.

Рассмотрим базовые классы Qt/3D и сравним их с классами других графических систем Qt. Класс **`QGLPainter`** является, своего рода, аналогом класса **`QPainter`** для OpenGL/ES. Так же, как и у класса **`QPainter`**, у **`QGLPainter`** есть методы **`begin()`** и **`end()`**. Вместо семейства методов **`draw*`** (drawLine(), **`drawRect()`**) у **`QGLPainter`** есть один метод **`draw()`**, доступный в нескольких перегруженных вариантах. Первым параметром у всех этих перегруженных методов является значение типа **`QGL::DrawingMode`**, которое определяет, что же именно будет рисовать метод **`draw()`**. Конкретные значения перечислимого типа **`QGL::DrawingMode`** соответствуют стандартным графическим примитивам OpenGL: **`QGL::Lines`**, **`QGL::Triangles`**, **`QGL::TriangleStrip`** и так далее. Координаты для рисования примитивов берутся из вершинных массивов (vertex arrays). Метод **`isCullable()`**,

существующий в двух перегруженных вариантах, позволяет проверить находится ли заданная точка (или, во втором варианте – трехмерный «ящик», то есть прямоугольный параллелепипед) внутри видимого пространства. Второй вариант метода возвращает значение **`false`** только в том случае, если прямоугольный параллелепипед целиком находится за пределами видимого пространства. Кроме того **`QGLPainter`** позволяет манипулировать матрицами преобразования, материалами, освещением, камерой и другими объектами OpenGL.

Предком класса **`QGLPainter`** является класс **`QOpenGLFunctions`**. Этот класс инкапсулирует функции OpenGL/ES 2.0 таким образом, что при использовании этого класса, вам не придется беспокоиться о переносе приложения с одной платформы на другую. Просто вместо функций OpenGL нужно вызывать одноименные методы класса **`QOpenGLFunctions`**. Таким образом, классы Qt/3D полностью скрывают разницу между OpenGL и OpenGL для встроенных систем. Один и тот же код будет одинаково работать и на десктопе, и на мобильном телефоне.

Класс **`QGLView`** является потомком знакомого нам класса **`QGLWidget`**. В отличие от **`QGLWidget`**, который, сам по себе, может только показывать трехмерную сцену, **`QGLView`** обрабатывает события клавиатуры и мыши таким образом, чтобы зритель мог перемещаться по трехмерной сцене. Кроме того класс **`QGLView`** позволяет формировать изображения для тех стерео-устройств, которые формируют отдельное изображение для каждого глаза.

Очень полезны классы **`QPlane3D`**, **`QRay3D`**, **`QTriangle3D`**. Эти классы упрощают выполнение таких важных в программировании трехмерной графики операций, как нахождение точки пересечения прямой и плоскости, принадлежности точки прямой и т.д.

Перед программистами OpenGL часто возникает проблема, связанная с трансляцией координат окна виджета в координаты трехмерного мира модели OpenGL. Например, когда пользователь щелкает мышью в окне, в котором выполняется вывод графики OpenGL, по одному из трехмерных объектов, мы хотим, чтобы этот объект как-то реагировал на щелчок пользователя. Для этого нам нужно геометрическое преобразование, которое бы преобразовывало координаты точки, заданные в системе координат окна виджета, в луч, соответствующий этой точке.

Напомню, что с точки зрения OpenGL, изображение, которое формируется в окне программы, представляет собой проекцию трехмерной сцены на двухмерную плоскость окна. Луч, совпадающий с линией проекции некоторой точки трехмерного мира, проецируется на плоскость виджета как точка. При преобразовании координат виджета в координаты трехмерного мира мы решаем обратную задачу: находим луч, совпадающий с линией проекции, соответствующей данной точке. Решение этой задачи требует знаний аналитической геометрии и сравнительно громоздких вычислений. К счастью такие классы как **`QPlane3D`**, **`QRay3D`**, **`QVector3D`**, и специальные методы классов **`QGLView`** и **`QGLCamera`** упрощают решение этой задачи. Программа tank (главные значения этого слова – «бак», «цистерна»), входящая в дистрибутив Qt/3D (рис. 5), демонстрирует преобразование точки окна программы в луч в трехмерной сцене OpenGL.

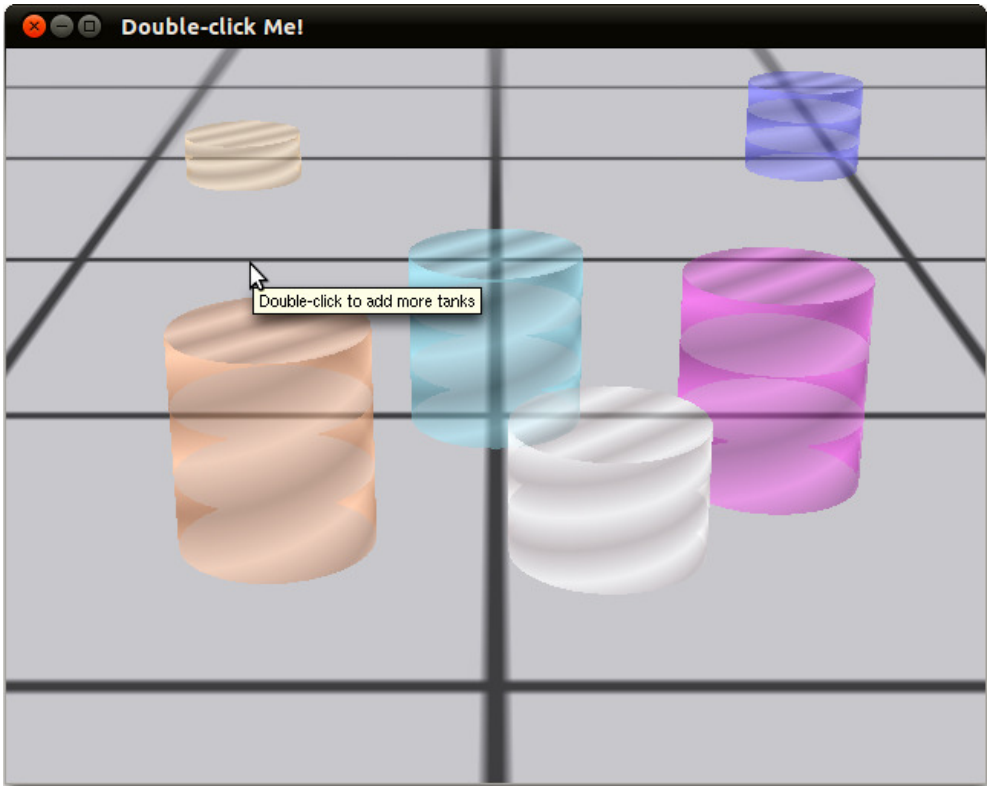

Рис. 5. Программа tank взаимодействует с мышью

Если после всего сказанного вы захотите знать получить библиотеки Qt/3D в свое распоряжение, готовьтесь преодолевать препятствия. Исходные тексты Qt/3D открыты на тех же условиях, что и тексты Qt, но нас ждут технические трудности. Поскольку на момент написания этих строк Qt/3D еще не является официальной частью Qt, библиотеки придется собирать из исходных текстов. Получить исходные тексты можно из репозитория git. Подробная инструкция содержится на сайте **http://doc.qt.nokia.com/qt3d-snapshot/qt3d-building.html**. Отмечу только, что для сборки Qt/3D вам понадобятся еще и исходные тексты самой Qt, не ниже версии 4.7. То есть, сначала надо собрать библиотеку Qt из исходных текстов (готовый SDK тут не подойдет), а затем уже собирать Qt/3D. Если вы все еще не передумали, то рассмотрим простой пример использования Qt/3D.

Сколько строк кода OpenGL необходимо, чтобы вывести на экран текстурированную полусферу? Я полагаю – несколько десятков. В Qt/3D эта же задача решается значительно меньшим числом строк кода. Программа dome наглядно это демонстрирует. Основой программы является объект класса **`DomeView`**, которые наследует классу **`QGLView`** (листинг 12).

Листинг 12. Объявление класса `DomeView`

```
class DomeView : public QGLView
{
    Q_OBJECT
public:
    DomeView(QWidget *parent = 0);
    ~DomeView();
protected:
    void paintGL(QGLPainter *painter);
    void timerEvent (QTimerEvent *);
private:
    QGLSceneNode * dome;
```

```
    QGLTexture2D tex;
    int angle;
};
```

Первое, на что нужно обратить внимание, при анализе класса **DomeView**, это то, что мы перекрываем только метод **paintGL()**, но не **initializeGL()** и не **resizeGL()**. В классе **QGLView** эти методу уже делают то, что требуется нам по умолчанию. Что, разумеется, не мешает нам перекрыть их, если настройки по умолчанию нас не устраивают.

Функции класса **QGLTexture2D** должны быть понятны из его названия. Этот класс отвечает за работу с двухмерными текстурами. Гораздо интереснее второй класс – **QGLSceneNode**. Класс **QGLSceneNode** и его производные позволяют нам строить трехмерные сцены как иерархии объектов. При этом родительский элемент иерархии «отвечает» за поведение дочерних элементов. Например, при применении геометрических преобразований, описаний свойств материалов или графических эффектов к родителю, к дочерним по умолчанию те же преобразования и эффекты применяются к дочерним элементам. Создав элемент класса **QGLSceneNode** один раз, мы можем манипулировать им, и всеми связанными с ним элементами, как единым графическим примитивом. Например, для графического отображения иерархической сцены достаточно вызвать один метод **draw()** объекта класса **QGLSceneNode**. От общей структуры класса **DomeView** перейдем к его реализации (листинг 13).

Листинг 13. Реализация класса *DomeView*

```
DomeView::DomeView(QWidget *parent)
    : QGLView(parent)
{
    QGLBuilder builder;
    builder << QGL::Faceted;
    builder << QGLDome();
    dome = builder.finalizedSceneNode();
    startTimer(50);
    tex.setImage(QImage(QLatin1String(":/stripes.png")));
}

DomeView::~DomeView()
{
    delete dome;
}

void DomeView::paintGL(QGLPainter *painter)
{
    painter->setStandardEffect(QGL::LitDecalTexture2D);
    painter->setFaceColor(QGL::AllFaces, QColor(170, 202, 0));
    tex.bind();
    painter->modelViewMatrix().rotate(angle, 1, 1, 1.0f);
    painter->modelViewMatrix().rotate(-45, 1,  1, 1.0f);
    dome->draw(painter);
}

void DomeView::timerEvent (QTimerEvent *)
{
  angle +=1;
  updateGL();
}
```

В процессе изучения реализации класса **DomeView** нам придется познакомиться с еще одним классом Qt/3D – **QGLBuilder**. Для того чтобы создать объект класса

QGLSceneNode, содержащий требуемые нам данные в оптимальной для отображения форме, необходимо воспользоваться услугами «помощника», каковым и является объект класса **QGLBuilder**. Идея, которая лежит в основе **QGLBuilder**, заключается в следующем: объект класса **QGLSceneNode** может содержать множество треугольников, и других примитивов OpenGL. Кроме того, объект **QGLSceneNode** может содержать дочерние объекты с не меньшим, а то и большим количеством графических примитивов. Очевидно, что отрисовка этих сложных геометрических структур может быть оптимизирована. Именно это и делает класс **QGLBuilder**. В рассматриваемом примере мы используем перегруженный оператор **<<**. Для класса **QGLBuilder** этот оператор существует в двух вариантах. В первом варианте операндом оператора должно быть значение типа **Smoothing**, которое указывает, должны ли, и если должны, то, как именно, сглаживаться грани фигуры. Константа **QGL::Faceted** означает, что мы не хотим сглаживания граней (даже полусфера, заданная по умолчанию, содержит так много граней, что сглаживание не требуется).

Во втором варианте этот оператор принимает операнд типа **QGeometryData**, который представляет собой набор вершин и их атрибутов. Как ни странно, класс **QGLDome**, который собственно и отвечает за отрисовку полусферы, не является потомком **QGeometryData**. В текущей версии Qt/3D у этого класса вообще нет предка (поскольку документация по Qt/3D, доступная на сайте проекта, не упоминает иерархические отношения классов, мы можем сделать вывод, что в будущем они могут меняться). Класс **QGLDome** обладает собственным перегруженным оператором **<<**, который позволяет добавлять объект этого класса в коллекцию объекта **QGLBuilder**. Вероятно, такая сложная структура так же реализована в интересах оптимизации. Объект **dome** класса **QGLSceneNode** создается с помощью метода **finalizedSceneNode**() класса **QGLBuilder**. Все, что нам теперь нужно сделать для отрисовки полусферы – добавить вызов **dome.draw()** в метод **paintGL()** класса **DomeView**.

Назначение объекту текстуру в методе **paintGL()** соответствует распространенной практике программирования в OpenGL, но не очень соответствует идеологии класса **QGLSceneNode**, который должен, по идее, содержать всю информацию, касающуюся своего трехмерного объекта. Можно ли задать информацию о текстуре объекта **QGLSceneNode** в самом объекте? Можно (листинг 14).

Листинг 14. Привязка текстуры к объекту *QGLSceneNode*

```
QGLBuilder builder;
builder << QGL::Faceted;
builder << QGLDome();
dome = builder.finalizedSceneNode();
tex.setImage(QImage(QLatin1String(":/stripes.png")));
QGLMaterial * material = new QGLMaterial();
material->setTexture(&tex);
dome->setMaterial(material);
```

Метод **setMaterial()** позволяет задать материал объекта (описываемый объектом класса **QGLMaterial**), а среди прочих параметров материала можно указать и текстуру (метод **material->setTexture()**). Обратите внимание на то, что объекты **material** и tex должны существовать, пока существует объект **dome**, который получил на них ссылки. Теперь из метода **paintGL()** можно убрать вызов **tex.bind()**, поскольку сам объект **dome** «знает», какая текстура ему назначена. В результате мы получим текстурированную полусферу, которую можно перемещать при помощи клавиш-стрелок или приближать и отдалять с помощью колесика мыши (рис. 6).

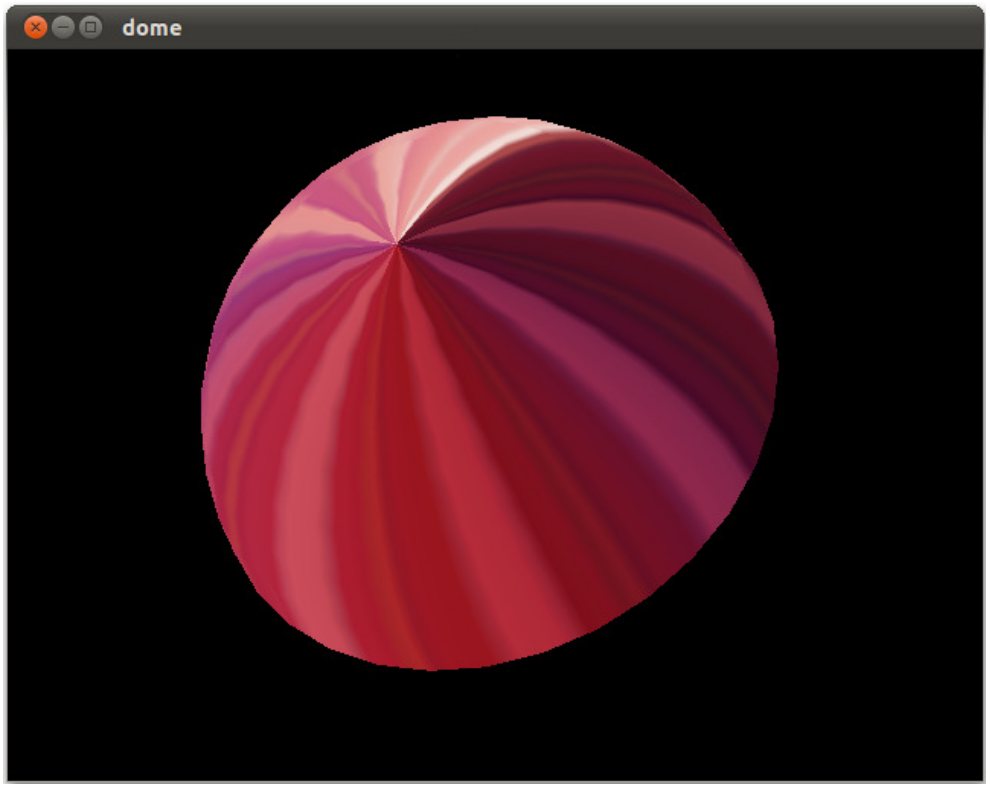

Рис. 8.6. Полусфера, покрытая текстурой

Примечание

Необходимо сделать некоторые замечания о сборке проектов Qt/3D. Даже после выполнения make install эта подсистема не становится частью стандартной установки Qt. Вы должны позаботиться о доступности всех заголовочных файлов вашему проекту, а так же о том, чтобы библиотека libQt3D.so (или, соответственно, Qt3D.dll) была доступна для связывания и динамической загрузки в вашей программе.

Если вы пишете программу, которая использует различные «продвинутые» возможности OpenGL, вам не помешает средство надежного определения возможности видеокарты той машины, на которой ваша программа запущена. В Qt/3D эту задачу решает класс **`QGLInfo`**. У класса **`QGLInfo`** есть один единственный метод report(), который возвращает все данные, которые удалось собрать системе, одной строкой. В данный момент этот класс не расположен в основной ветке исходных текстов Qt/3D, и документация о нем отсутствует. Исходные тексты класса расположены в директории qt-labs/qt3d/blobs/master/util/qglinfo/. Там же расположена программа, которая демонстрирует его использование (рис. 7). Формат, в котором класс **`QGLInfo`** выдает данные о возможностях OpenGL, может вам не подойти, но исходные тексты класса научат вас получать необходимые сведения в любом формате.

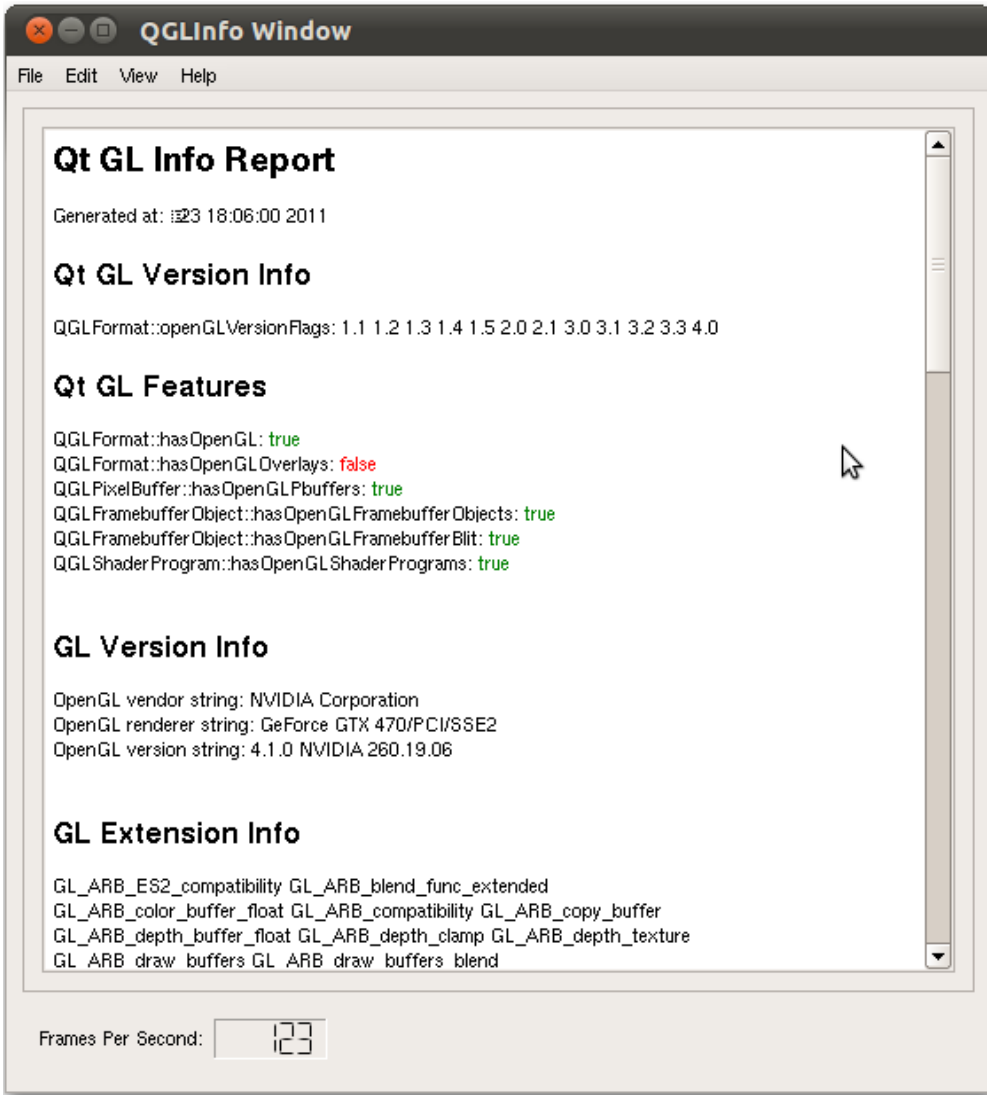

Рис. 7. Окно программы qglinfo

Поддержка OpenCL в Qt 4.8

Видеокарты современных компьютеров вполне сопоставимы по мощности с их основными блоками (а в решении некоторых задач даже превосходят центральный процессор). Тем не менее, эта мощь редко используется на все 100%, разве что, когда вы играете в новейшие компьютерные игры. Идея использовать расширенные возможности видеокарт не только для игр и трехмерного моделирования возникла давно (и не первый год применяется на практике). Например, язык GLSL, с которым мы познакомились в этой главе, может использоваться не только для программирования шейдеров, но и, например, для вычислений в области векторной алгебры. Главная проблема с использованием графических процессоров в неграфических целях до сих пор заключается в отсутствии единого стандарта такого использования. Очень часто решение приходится создавать под конкретную систему. Ситуация осложняется еще и тем, что традиционные языки программирования плохо поддерживают те инструкции современных процессоров, которые ориентированы на одновременную обработку больших массивов данных. Для реализации таких инструкций приходится либо опускаться на уровень ассемблера, либо использовать специальные наборы функций, написанных на том же уровне. Однако ситуация начинает постепенно меняться. Язык OpenCL (Open Computing Language — открытый язык вычислений) должен стать для сложных вычислений тем же, чем стал язык OpenGL для трехмерной компьютерной графики – кросс-платформенным

средством, позволяющим максимально задействовать имеющиеся в системе ресурсы для параллельной обработки данных.

Адрес официальной страницы OpenCL – **www.khronos.org/opencl**. На этой странице вы можете узнать о целях и возможностях языка, а так же о текущем состоянии его разработки и распространения. Поскольку OpenCL ориентирован на работу со специализированными процессорами (или процессорными модулями), предназначенными для параллельной обработки данных, представленных в виде векторов, лучше всего он подходит именно для решения задач, в которых используются массивы данных. Примерами таких задач являются быстрое преобразование Фурье, вычисление сверток, применение к данным цифровых фильтров и тому подобное. Задачи такого рода возникают не только при обработке графики или в научных расчетах, но и, например, при обработке аудиоданных, так что язык OpenCL может пригодиться очень многим.

Примечание

Хотя программы, использующие OpenCL, могут загрузить специализированные процессоры вашей системы так, что у нее не хватит мощности для выполнения обычных задач, такая ситуация вовсе не является типичной. Как правило, программы, использующие OpenCL для решения математических задач могут выполняться совместно с другими программами, например, с графическими.

Важная особенность языка OpenCL – высокий уровень абстракции. OpenCL не делает принципиальной разницы между графическим процессором видеокарты и расширенными наборами инструкций центрального процессора. Это означает, что программы OpenCL могут выполняться, хотя и с разным уровнем производительности, как на системах с мощными видеокартами, так и на системах, где сами виедопроцессоры не поддерживают сложные наборы инструкций.

Не следует, однако, быть слишком большими оптимистами в том, что касается современного состояния OpenCL. Далеко не все видеокарты, потенциально способные поддерживать OpenCL, могут делать это на практике, так как для них отсутствуют специальные драйверы (ситуация аналогичная той, когда аппаратные средства видеокарты поддерживают ускорение операций OpenGL, а реализация драйвера – нет).

Но это еще не все. Если OpenGL создавалась как клиент-серверная система, так что для работы с аппаратно-ускоренными средствами OpenGL вам необходимы только видеокарта и драйвер, поддерживающий эти операции, то для работы с OpenCL необходим еще и специальный компилятор, который преобразует программу OpenCL в последовательность команд для определенного процессора. У каждого производителя графических чипсетов эти компиляторы, естественно, свои.

Примечание

На практике применяются две стратегии работы с OpenCL. В одном случае программа OpenCL распространяется в виде исходных текстов, и компилируется во время выполнения основной программы, компилятором, установленным в конкретной системе, с учетом особенностей этой системы. Если вы не хотите распространять свои модули OpenCL в исходных текстах, вы можете предварительно скомпилировать их для нескольких целевых платформ. Основная программа во время выполнения определит, на какой платформе она выполняется и загрузит подходящий модуль.

В чем же заключается поддержка OpenCL в Qt? Как и в случае с Qt/3D, основная цель здесь – создание совместимого с Qt интерфейса, который максимально скрывал бы от программиста особенности работы OpenCL на разных платформах.

Хотя поддержка OpenCL в Qt появится только начиная с версии 4.8, протестировать работу QtOpenCL можно уже с версией 4.7, и даже с более ранними версиями, причем исходных текстов самих библиотек Qt для этого не нужно. Так же как и Qt/3D, QtOpenCL необходим собрать из исходных текстов, которые доступны в репозитории git. Подробное описание процесса сборки и установки, так же как и другую информацию, можно найти на сайте **http://doc.qt.nokia.com/opencl-snapshot/**. Отмечу сразу, что, поскольку реализация QtOpenCL носит пока еще

экспериментальный характер, сборка этой системы может оказаться делом нетривиальным. Мне, например, пришлось вносить изменения в исходные тексты только для того, чтобы система QtOpenCL скомпилировалась. Впрочем, разработка не стоит на месте и к моменту выхода книги из печати положение дел со сборкой QtOpenCL может улучшиться.

Как уже отмечалось, для того чтобы задействовать видеокарту в решении математических задач, вам нужен драйвер видеокарты с поддержкой OpenCL 1.1 и соответствующий SDK с компилятором и библиотеками среды времени выполнения. Убедиться в том, что все это у вас есть и правильно настроено поможет программа clinfo. Исходные тексты этой программы вы найдете в директории util пакета QtOpenCL.

Эта программа выводит довольно много информации обо всех обнаруженных платформах OpenCL. Посмотрим на сокращенный вариант вывода clinfo в моей системе (листинг 15). Если вы пользуетесь Qt Creator, то для того чтобы увидеть этот вывод, вам необходимо запустить программу в режиме отладки.

Листинг 15. Вывод программы clinfo

```
OpenCL Platforms:
    Platform ID       : 5851472
    Profile           : FULL_PROFILE
    Version           : OpenCL 1.1 WINDOWS
    Name              : Intel(R) OpenCL
    Vendor            : Intel(R) Corporation
    Extension Suffix  : Intel
    Extensions        :
...
    Platform ID       : 239706308
    Profile           : FULL_PROFILE
    Version           : OpenCL 1.1 AMD-APP-SDK-v2.4 (595.10)
    Name              : AMD Accelerated Parallel Processing
    Vendor            : Advanced Micro Devices, Inc.
    Extension Suffix  : AMD
...
OpenCL Devices:
...
    Device ID         : 240713712
    Platform ID       : 239706308
    Vendor ID         : 4098
    Type              : GPU
    Profile           : FULL_PROFILE
    Version           : OpenCL 1.1 AMD-APP-SDK-v2.4 (595.10)
    Driver Version    : CAL 1.4.1385 (VM)
    Language Version  : OpenCL C 1.1
    Name              : Cypress
    Vendor            : Advanced Micro Devices, Inc.
    Available         : true
    Compute Units     : 14
    Clock Frequency   : 800 MHz
    Address Bits      : 32
    Byte Order        : Little Endian
    ...
    Device ID         : 240718192
    Platform ID       : 239706308
    Vendor ID         : 4098
    Type              : CPU
```

```
    Profile           : FULL_PROFILE
    Version           : OpenCL 1.1 AMD-APP-SDK-v2.4 (595.10)
    Driver Version    : 2.0
    Language Version  : OpenCL C 1.1
    Name              : Intel(R) Core(TM)2 Quad CPU    Q9400  @ 2.66GHz
    Vendor            : GenuineIntel
    Available         : true
    Compute Units     : 4
    Clock Frequency   : 2666 MHz
    Address Bits      : 32
    Byte Order        : Little Endian
   ...
```

Как видим, программа нашла две платформы OpenCL – Intel и AMD и два устройства – центральный процессор Intel и графический процессор AMD, расположенный на видеокарте. Все это означает, что я могу использовать программы OpenCL в программах Qt, используя в качестве процессора как ЦП, так и видеопроцессор.

В качестве примера использования графического процессора для решения не-графических задач, рассмотрим программу vectoradd из пакета **QtOpenCL**. Эта программа использует OpenCL для сложения двух векторов, каждый из которых состоит из 2048 чисел типа **int**. Я внес небольшие модификации в исходный текст этой программы (листинг 16).

Листинг 16. Модифицированный вариант программы vectoradd

```
int main(int, char **)
{
    QCLContext context;
    if (!context.create(QCLDevice::GPU)) {
        fprintf(stderr, "Could not create OpenCL context for the GPU\n");
        return 1;
    }
    QCLVector<int> input1 = context.createVector<int>(2048);
    QCLVector<int> input2 = context.createVector<int>(2048);
    for (int index = 0; index < 2048; ++index) {
        input1[index] = index/2;
        input2[index] = (index + 1)/2;
    }
    QCLVector<int> output = context.createVector<int>(2048);
    QCLProgram program =
context.buildProgramFromSourceFile(":/vectoradd.cl");
    QCLKernel kernel = program.createKernel("vectorAdd");
    kernel.setGlobalWorkSize(2048);
    kernel(input1, input2, output);
    for (int index = 0; index < 2048; ++index) {
        if (output[index] != index) {
            fprintf(stderr, "Answer at index %d is %d, should be %d\n",
                    index, output[index], index);
            return 1;
        } else
        printf("Answer is correct: %d\n", index);
    }
    return 0;
}
```

Описание состояния системы OpenCL хранится в объекте context класса **QCLContext**. Как и в случае **QGLContext**, класс **QCLContext** является не изобретением разработчиков Qt, а отображением одноименной концепции OpenCL. Для

инициализации контекста OpenCL используется метод **`create()`** класса **`QCLContext`**. В качестве параметра метода **`create()`** можно указать устройство OpenCL, на котором следует выполнять программу. Если устройство не указывать, система выберет то устройство, которое она считает наиболее подходящим. Я хочу, чтобы программа выполнялась именно на графическом процессоре, поэтому в качестве аргумента команды указываю **`QCLDevice::GPU`**.

Далее мы готовим векторы для ввода и вывода данных, собираем программу OpenCL, создаем объект **`kernel`** (вызов **`program.createKernel()`**) и локальную рабочую область. После выполнения программы OpenCL мы сравниваем результат сложения двух векторов с ожидаемым результатом.

Как и программа clinfo, программа vectoradd является консольной (не могу не обратить внимания на иронию ситуации: программа, в которой задействованы самые передовые возможности видеопроцессора, выводит данные в консольном режиме). Так что, если вы работаете в Qt Creator, то, как и в случае с программой clinfo, для того чтобы увидеть вывод данной программы, необходимо запустить Qt Creator в режиме отладки.