

Qt/3D Введение

Вопреки моим (и, наверное, не только моим) ожиданиям, Qt/3D не войдет в будущую версию Qt – 4.8. Во всяком случае, в предварительном релизе Qt 4.8 нет даже намека на эту систему, хотя изначально планировалось включить Qt/3D именно в эту версию Qt library, и даже документация по Qt/3D сообщает, что соответствующие классы становятся доступны в Qt 4.8. На данный момент это не так.

Возможно, включение Qt/3D в основную ветку Qt вообще отложено до версии 5.0, которая, как говорят, выйдет в следующем году. Но познакомиться с Qt/3D можно уже сейчас, и не только познакомиться, но и использовать в собственных проектах, особенно, если вы не боитесь риска.

Qt/3D преследует три цели: реализовать при работе с трехмерной графикой некоторые из тех возможностей, которые Graphics View Framework реализует для двухмерной графики и добавить поддержку трехмерной графики в систему QtQuick и язык QML. Еще одна цель – предоставить программистам прикладной интерфейс, который позволял бы писать приложения для OpenGL и OpenGL/ES, используя общий набор классов Qt.

Ну и помимо всего этого, Qt/3D заметно упрощает программирование трехмерной графики (по сравнению с «сырым» кодом OpenGL).

На данный момент система Qt/3D не доступна на сайте Qt даже в виде архива исходных текстов. Скачать ее можно только из репозитория Git. Это легко сделать, если вы работаете под Linux, поскольку соответствующая утилита входит в любой дистрибутив. Под Windows это тоже сделать нетрудно, так как есть утилита TortoiseGit (аналог программ TortoiseCVS и TortoiseSVN, которые вам наверняка известны). Ссылку на скачивание можно найти на сайте проекта <http://doc.qt.nokia.com/qt3d-snapshot/> (раздел Building, там же находятся инструкции по сборке).

На платформе Windows я собрал Qt/3D под Qt 4.8 Technology Preview и Microsoft Visual Studio 2010. Кстати, если вы не знали. По умолчанию Qt Creator работает с компилятором MinGW, но он может работать и с компилятором Microsoft. В результате над проектом можно будет работать и в Microsoft Visual Studio, и в Qt Creator. Qt Creator последней версии (2.2.1 и более поздних) сам найдет имеющиеся у вас сборки Qt library. Если сборка Qt собиралась под Microsoft Visual Studio, то и Qt Creator будет использовать соответствующий компилятор. Для полноценной работы с инструментарием Microsoft понадобится еще один компонент – консольный отладчик CDB. Этот отладчик входит в состав Windows SDK и, если SDK установлен в вашей системе, Qt Creator тоже его найдет.

Центральная идея Qt/3D – построение трехмерной сцены в виде иерархии графических узлов. Идея эта не нова, она используется во многих библиотеках-надстройках над OpenGL, так что, возможно, вы уже знакомы с этой идеей в том или ином варианте. Узлом сцены может быть треугольник, фигура, состоящая из нескольких треугольников, сцена, состоящая из нескольких фигур. Кроме того, у узла могут быть дочерние узлы.

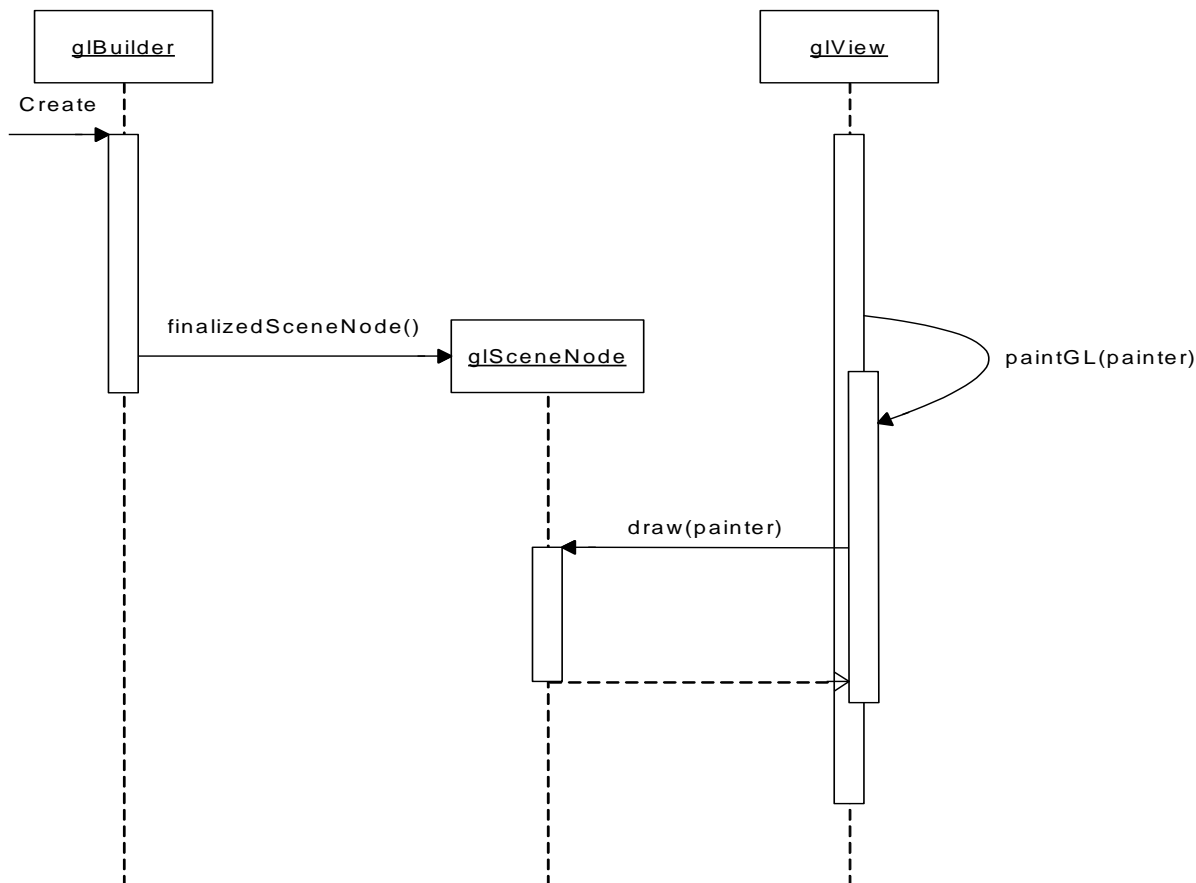
Система Qt/3D включает в себя множество классов. Самые важные из них – **QGLView**, **QGLBuilder**, **QGLSceneNode** и **QGLPainter**. Класс **QGLView** является расширением (и потомком) класса **QGLWidget**, который, напомним, играет центральную роль в работе с OpenGL в нынешних версиях Qt. По сравнению с **QGLWidget** класс **QGLView** обладает некоторыми дополнительными возможностями, например, взаимодействует с мышью, так что по умолчанию трехмерную сцену можно вращать и масштабировать с помощью мыши, как это принято в программах просмотра трехмерной графики. Во многих примерах, которые входят в дистрибутив Qt/3D. С помощью объекта этого класса реализуется главное окно программы. То же самое сделаем и мы.

Класс **QGLPainter** можно рассматривать как аналог **QPainter** для Qt/3D, хотя «генетического родства» между этими классами нет. **QGLPainter** специально предназначен для работы с трехмерной графикой. Впрочем, если вам нужно использовать возможности **QPainter** совместно с Qt/3D, вы можете это сделать. Поскольку **QGLView** наследует классу **QGLWidget**, он является потомком и класса **QPaintDevice**.

На класс **QGLBuilder** возложена задача построения трехмерной сцены. Этот класс получает сведения о геометрии сцены, текстурах, свойствах материалов и т.д. При построении сцен выполняются различные оптимизации, призванные ускорить последующий вывод сцены. Как мы увидим далее, эти оптимизации не всегда делают именно то, что нам бы хотелось. Результатом работы объекта класса **QGLBuilder** является объект класса **QGLSceneNode**, который представляет собой узел графической сцены. Обычно работа объектом **QGLBuilder** заканчивается вызовом метода **finalizedSceneNode()**, который возвращает указатель на объект **QGLSceneNode**, который является корневым узлом сцены. После этого объект **QGLBuilder** в принципе не нужен.

Когда объекту **QGLView** требуется перерисовать содержимое окна, он вызывает метод **paintGL()**, которому в качестве параметра painter передается указатель на объект **QGLPainter**. Для того чтобы

выполнить отрисовку сцены, мы вызываем метод `draw()` объекта `QGLSceneNode`, которому в качестве аргумента передаем указатель `painter`.



Вообще говоря, ничто не мешает вам создать несколько независимых объектов `QGLSceneNode`, а в методе `paintGL()` последовательно вызывать методы `draw()` каждого из них. Можно вызывать несколько раз метод `draw()` одного и того же объекта `QGLSceneNode`. Один из случаев, когда это может понадобиться, будет показан ниже. Еще один возможный случай – когда вы хотите отобразить несколько копий сцены в разных областях окна `QGLView`.

Посмотрим теперь, как это выглядит в коде. Программа-пример создает простую сцену: полупрозрачный куб и в нем две сферы – одна равномерно окрашена, вторая текстурирована.

Графическая сцена создается в конструкторе класса `SceneView`, который наследует классу `QGLView` и реализует главное окно программы.

```

QGLBuilder builder;
builder << QGL::Smooth;
builder << QGLCube(3);
builder.newSection(QGL::Smooth);
builder << QGLSphere();
QGLSceneNode * sphere = builder.currentNode();
QGLMaterial * sphereMat = new QGLMaterial();
sphereMat->setEmittedLight(QColor(150,250,80));
sphere->setMaterial(sphereMat);
sphere->setX(-0.5);
builder.newSection(QGL::Smooth);
builder << QGLSphere();
QGLSceneNode * sphere2 = builder.currentNode();
tex.setImage(QImage(QLatin1String(":/stripes.png")));
QGLMaterial * sphereMat2 = new QGLMaterial();
sphereMat2->setTexture(&tex);
sphere2->setMaterial(sphereMat2);
sphere2->setEffect(QGL::LitDecalTexture2D);
sphere2->setY(-0.8);
  
```

```
sphere2->setX(0.2);
rootNode = builder.finalizedSceneNode();
```

С помощью оператора `<<` объекту класса **QGLBuilder** можно передавать самые разные значения. Константа **QGL::Smooth** указывает на то, что при освещении сцены эффект должен быть мягким (альтернатива – константа **QGL::Faceted**). Класс **QGLCube** содержит геометрические данные, необходимые для создания куба. Это один из нескольких классов Qt/3D, которые специально предназначены для того чтобы упростить добавление в сцену некоторых распространенных фигур. Помимо куба, таким простым способом можно добавить в сцену сферу (класс **QGLSphere**), цилиндр (класс **QGLCylinder**), закрытую полусферу (класс **QGLDome**) и «чайник Ньюела» - знаменитый чайник, который часто используется для демонстрации трехмерных эффектов (класс **QGLTeapot**). Помимо списка координат, описывающих геометрию самой фигуры, эти классы содержат также списки двухмерных координат, необходимых для нанесения текстур.

Все перечисленные фигуры имеют по умолчанию некий «единичный» размер. Этот размер можно изменить, указав коэффициент масштабирования (с помощью метода **setSize()** или, как в нашем примере – в качестве аргумента конструктора). То есть линейные размеры куба, который мы добавляем в сцену, будут в 3 раза больше размеров куба по умолчанию.

Помимо узлов сцена разделяется на секторы (sections). При добавлении нового графического элемента (за исключением первого элемента) всегда необходимо создавать новый *сектор* (section) сцены. Многие методы класса **QGLBuilder** (такие как **addTriangle()**, **newNode()** и им подобные) создают новые секторы автоматически. Однако при работе с оператором `<<` новый сектор необходимо добавлять явно. Это и делает метод **newSection()**, который позволяет задать для новой секции новый режим освещения (с помощью уже известных нам констант **QGL::Smooth** и **QGL::Faceted**). Наша сцена содержит три самостоятельных графических элемента: куб и две сферы. Соответственно мы дважды вызываем метод **newSection()**. Если мы забудем вызывать **newSection()** в промежутках между добавлением графических элементов, во время выполнения программы на отладочную консоль будет выдано предупреждение.

После добавления нового сектора текущий узел сцены содержит графический элемент из этого сектора. Получить ссылку на текущий узел (объект **QGLSceneNode**) можно с помощью метода **currentNode()**. Что представляет собой этот узел? Он содержит геометрические данные фигуры. При построении узла сцены объект **QGLBuilder** выполняет триангуляцию поверхностей и может применять некоторые оптимизации, например, использовать вершинные массивы. Мы можем так же указать нормали, от направления которых зависят, например, параметры освещения, но делать это не обязательно. При отрисовке освещенной сцены нормали, если они не были заданы, будут рассчитаны автоматически. Помимо геометрических данных, узел содержит данные о материале фигуры и текстуре, заданные по умолчанию (то есть, унаследованные от родительского узла, или примеренные ко всей иерархии в целом). Мы, разумеется, можем изменить параметры материалов и текстуры, заданные по умолчанию. Эти параметры содержит объект класса **QGLMaterial**. В приведенном выше листинге вы можете видеть, как создаются цвет и другие параметры материала для одной сферы и параметры текстуры – для другой.

Еще один интересный элемент это методы **setX()**, **setY()** и **setZ()**. Эти методы позволяют задать координаты узла сцены в локальной системе координат сцены. Есть еще метод **setPosition()**, который позволяет задать все три координаты сразу и метод **addTransform()**, который позволяет задавать более сложные преобразования. Например, фигуры, создаваемые **QGLCube** и **QGLSphere**, по умолчанию располагаются так, чтобы геометрический центр фигуры совпадал с началом локальной системы координат. То есть, по умолчанию все фигуры будут расположены по центру. Если мы этого не хотим (а мы не хотим), можно изменить локальные координаты каждой фигуры. Локальная система координат называется так потому, что преобразования, выполняемые в ней, не влияют на другие сцены. При применении к сцене глобальных преобразований (с помощью непосредственной манипуляции матрицами моделирования и проектирования OpenGL), глобальные преобразования выполняются с учетом локальных.

Завершаем обзор конструктора.

```
lp.setAmbientColor(QColor("white"));
lp.setDiffuseColor(QColor("white"));
lp.setSpecularColor(QColor("white"));
lp.setPosition(QVector3D(2,2,-4));
startTimer(50);
makeCurrent();
glEnable(GL_BLEND);
glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_DST_ALPHA);
```

Переменная `lp` является объектом класса `QGLLightParameters`. Этот класс, как следует из его имени, определяет параметры источника освещения. В самом конструкторе объект этого класса нам не нужен. Мы создаем его «на будущее», для применения в методе `paintGL()`. Поскольку в демонстрационной программе используются полупрозрачные материалы, следует вызвать функцию `glEnable(GL_BLEND)` для включения соответствующего эффекта. Обратите внимание на то, что в нынешнем варианте Qt/3D наблюдается некоторая непоследовательность относительно вызовов функции OpenGL `glEnable()`. По идее, поскольку в программе используются текстуры и освещение, мы должны были бы вызвать так же `glEnable(GL_TEXTURE_2D)`, `glEnable(GL_LIGHTING)`, `glEnable(GL_LIGHT0)`, но этого не требуется.

Для анимации сцены мы используем таймер объекта Qt. В следующих примерах мы рассмотрим другие, более продвинутые способы анимации.

Метод `paintGL()` выглядит совсем несложно:

```
void SceneView::paintGL(QGLPainter *painter)
{
    painter->modelViewMatrix().rotate(angle, 1, 1, 1.0f);
    painter->modelViewMatrix().rotate(-45, 1, 1, 1.0f);
    glCullFace(GL_FRONT);
    glEnable(GL_CULL_FACE);
    rootNode->draw(painter);
    glCullFace(GL_BACK);
    rootNode->draw(painter);
    glDisable(GL_CULL_FACE);
    painter->addLight(&lp);
}
```

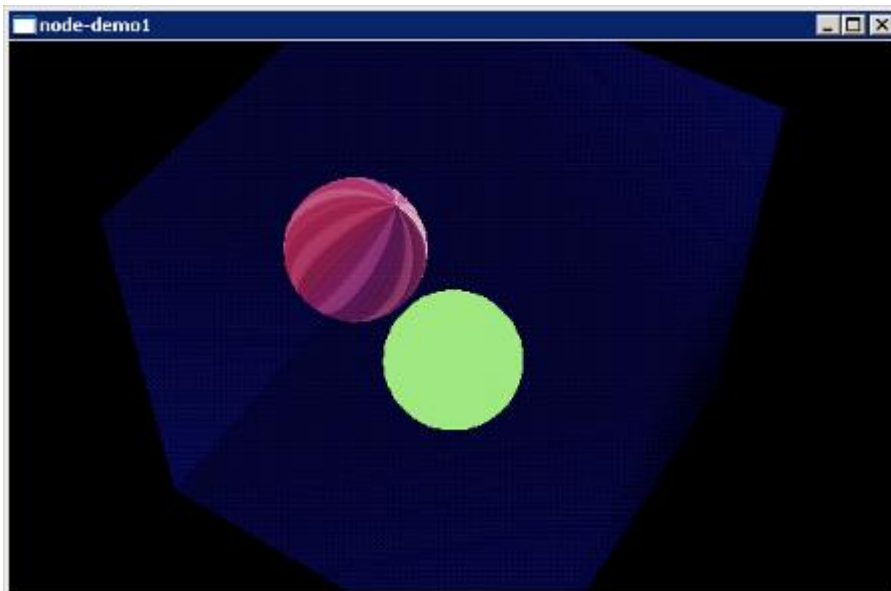
Как уже было отмечено, объект класса `QGLPainter` представляет собой движок, с помощью которого мы отображаем сцену. Если бы нам требовалось просто отобразить сцену, созданную в конструкторе, достаточно было бы одной строчки

```
rootNode->draw(painter);
```

Однако мы хотим добавить кое-что. Класс `QGLPainter` позволяет нам получить доступ к матрицам моделирования и проектирования. Эти методы преобразуют глобальные координаты сцены. Метод `modelViewMatrix()` возвращает не объект, описывающий матрицу (объект класса `QMatrix4x4`), а объект класса `QMatrix4x4Stack`, который представляет стек матриц. Методы `rotate()`, `translate()` и `scale()` манипулируют матрицей, находящейся на вершине стека. У объекта, возвращаемого методом `modelViewMatrix()`, есть так же методы `push()` и `pop()`.

Вызов функций `glCullFace()` необходим нам для того чтобы избежать некоторых артефактов, которые иначе возникают при отрисовке полупрозрачного куба (мы отрисовываем сцену в два приема: сначала только лицевые грани, потом только задние). Как вы уже заметили, мы можем свободно смешивать методы классов Qt/3D и прямые вызовы функций OpenGL (благодаря вызову `makeCurrent()` система OpenGL «знает», к какому контексту относятся вызовы функций OpenGL). В общем случае, лучше, конечно, использовать либо то, либо другое. Мы применяем методы классов Qt/3D везде, где это возможно, а функциями OpenGL пользуемся тогда, когда нам нужно сделать нечто, чего Qt/3D делать не умеет.

Метод `addLight()` класса `QGLPainter` добавляет источник освещения. Некоторые методы класса `QGLPainter` дублируют методы `QGLSceneNode`. Например, метод `setFaceColor()` позволяет задать цвет граней, а метод `setStandardEffect()`, помимо прочего, может включить поддержку текстур. Эти методы оказывают влияние на сцену, если соответствующие параметры не были заданы в объекте `QGLSceneNode` явным образом.



Еще одно небольшое дополнение. Файл проекта Qt для демонстрационной программы содержит, помимо ссылок на саму библиотеку Qt ссылки на заголовочные файлы и библиотеку Qt/3D (Qt3Dd или Qt3D, в зависимости от конфигурации сборки). Например

```
LIBS += -L../../../../lib -L../../../../bin  
include../../../../src/threed/threed_dep.pri
```

Эти относительные ссылки предполагают, что, во-первых, двоичные файлы Qt/3D расположены в поддиректориях bin и lib директории qt3d, а во-вторых, что сам проект демонстрационной программы расположен в одной из поддиректорий директории qt3d/examples/tutorials. Если у вас расположение директорий иное, в файл проекта необходимо внести соответствующие изменения.