

# Методические указания к лабораторным работам по компьютерной графике

## 1. Введение

В данном курсе лабораторных работ используются язык программирования Си++, открытая версия среды разработки Nokia Qt и графическая библиотека OpenGL. Этот набор инструментов позволяет быстро создавать приложения с графическим интерфейсом, которые можно собирать и запускать в большинстве существующих современных систем: в Windows-системах, в Unix-системах с графикой на основе оконной системы X (Linux, BSD, Solaris и т.п.), в Mac OS X и в большинстве мобильных вариантов перечисленных систем. Процесс создания приложений, использующих Qt, почти единообразен для всех систем, поэтому в данных указаниях рассматривается только работа в Windows.

Вопросы, касающиеся собственно компьютерной графики, рассматриваются в лекциях по этому курсу. В данных методических указаниях описано только применение инструментов (Qt и OpenGL; предполагается, что с языком Си++ студенты знакомы из более ранних курсов).

## 2. Дополнительная литература

Приведём источники, в которых можно получить дополнительные сведения по рассматриваемому инструментарию.

### 2.1. Язык Си++

1. Страуструп Б. Язык программирования C++. Специальное издание. – М.: ООО «Бином-Пресс», 2004.
2. Мейерс С. Эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов. – М.: ДМК Пресс; СПб.: Питер, 2006.
3. Саттер Г. Решение сложных задач на C++. Серия C++ In-Depth. – М.: Издательский дом «Вильямс», 2008.
4. Саттер Г. Новые сложные задачи на C++. – М.: Издательский дом «Вильямс», 2005.

### 2.2. Библиотека и среда Qt

1. Встроенная документация Qt (Qt Assistant).

### 2.3. Библиотека OpenGL

1. Ву М., Нейдер Дж., Девис Т., Шрайнер Д. OpenGL. Официальное руководство программиста. – СПб.: ООО «ДиаСофтЮП», 2002.
2. Гайдуков С.А. OpenGL. Профессиональное программирование трёхмерной графики на C++. – СПб.: БХВ-Петербург, 2004.
3. Боресков А.В. Графика трёхмерной компьютерной игры на основе OpenGL. – М.: ДИАЛОГ-МИФИ, 2004.
4. Боресков А.В. Расширения OpenGL. – СПб.: БХВ-Петербург, 2005.
5. Боресков А.В. Разработка и отладка шейдеров. – СПб.: БХВ-Петербург, 2006.

6. OpenGL – The Industry Standard for High Performance Graphics (<http://www.opengl.org/>).
7. Developer Zone: OpenGL Resources from NVIDIA (<http://developer.nvidia.com/page/opengl.html>).
8. AMD Developer Central: GPU Tools (<http://developer.amd.com/GPU/Pages/default.aspx>).

## 3. Средства разработки Qt

### 3.1. Готовим машину к работе

Установщик Qt можно загрузить из интернета (178 Мб) или получить у преподавателя. Чтобы загрузить установщик из сети, сделайте следующее:

- зайдите на сайт <http://qt.nokia.com/>;
- выберите ссылку **Download**, а затем вкладку **LGPL / Free**. На рисунке 3.1.1 показан фрагмент страницы, которая должна появиться. Выберите ссылку **Download Qt SDK for Windows**, чтобы загрузить установщик;



Рис. 3.1.1. Страница загрузки Qt SDK

- запустите загруженный файл;
- когда установщик загрузится (рис. 3.1.2), нажмите кнопку **Next**;
- выберите пункт **I accept the terms of the License Agreement** и нажмите кнопку **Next** (рис. 3.1.3);
- в следующих двух окнах (рис. 3.1.4) нажмите **Next**;
- в окне выбора каталога (рис. 3.1.5) укажите каталог (папку), в который вы хотите установить Qt SDK, и нажмите **Next**;
- в окне настройки «Главного меню» (рис. 3.1.6) нажмите **Install**;



Рис. 3.1.2. Начало установки Nokia Qt SDK

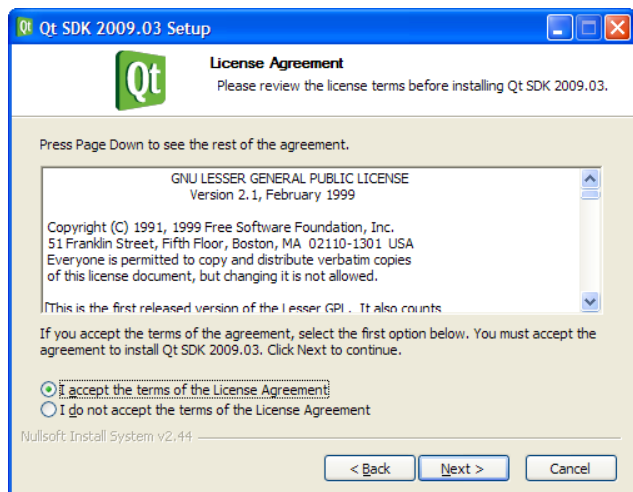


Рис. 3.1.3. Лицензионное соглашение

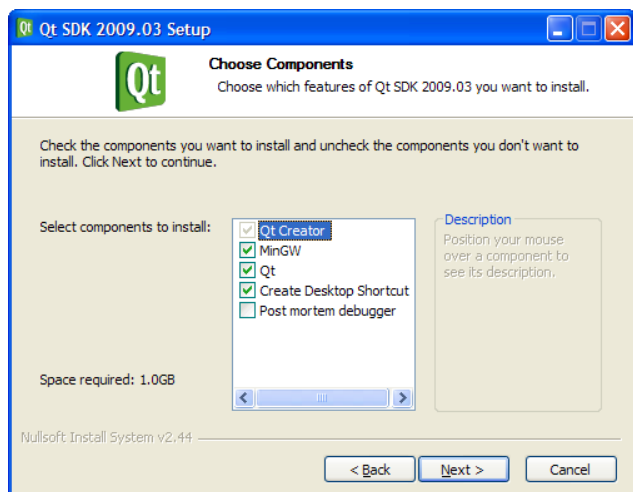
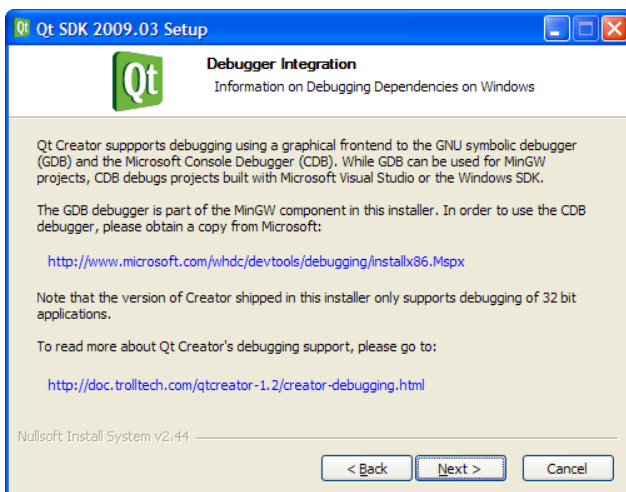


Рис. 3.1.4. Информация об отладчике и выбор компонентов

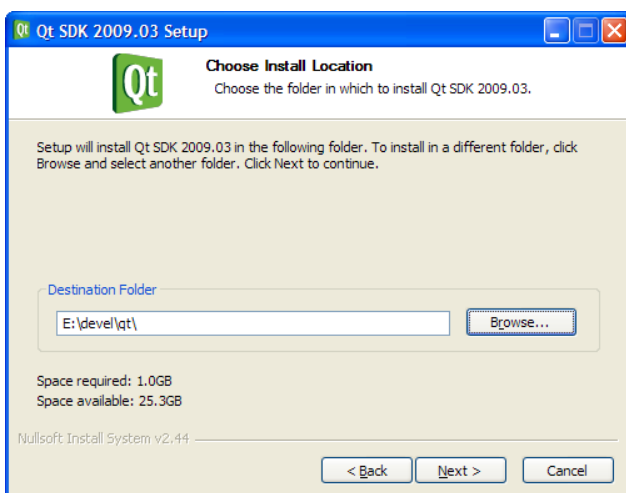


Рис. 3.1.5. Выбор каталога установки

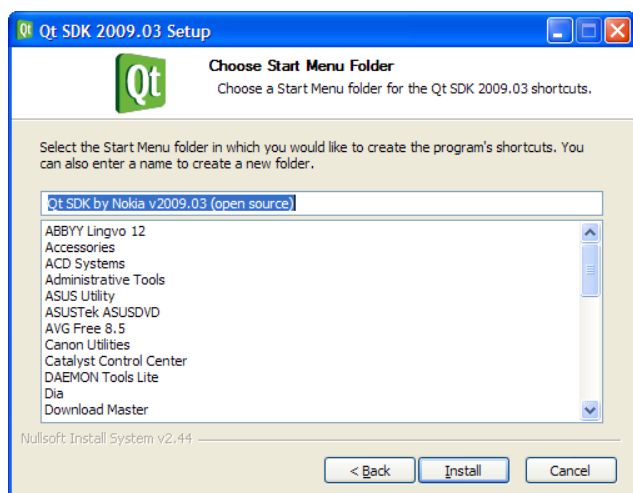


Рис. 3.1.6. Выбор раздела «Главного меню»

- после завершения установки (рис. 3.1.7) нажмите **Next**;
- в последнем окне установщика (рис. 3.1.8) нажмите **Finish**, чтобы выйти из него. Запустится среда Qt Creator, в которой вы сразу же сможете начать работать (в дальнейшем Qt Creator можно запускать из «Главного меню» Windows).

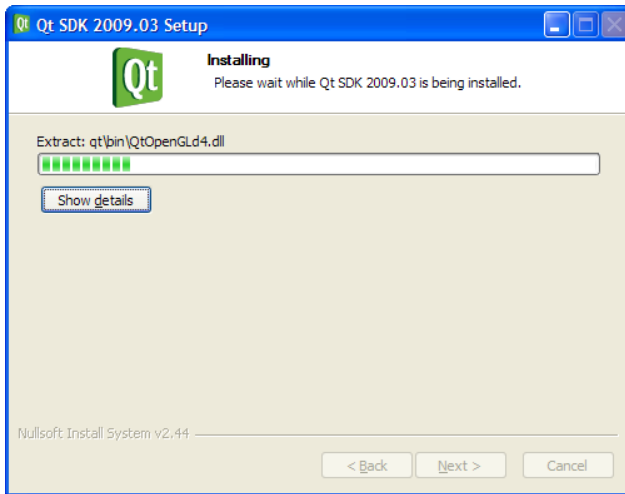


Рис. 3.1.7. Установка Qt SDK

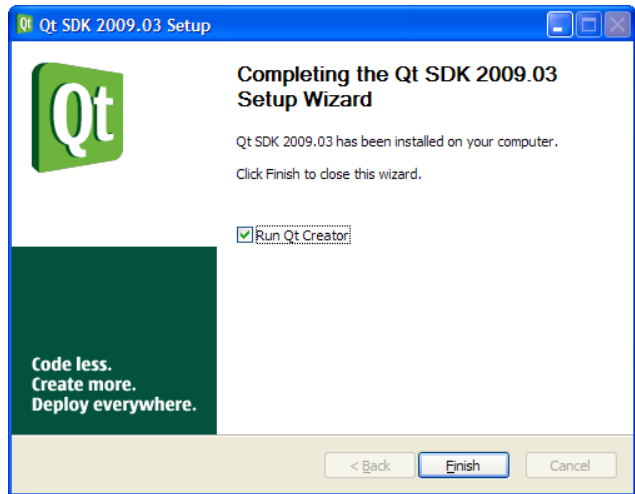


Рис. 3.1.8. Установка Qt SDK завершена

## 3.2. Пишем простую графическую программу

В этом разделе мы покажем, как создать простое приложение, которое выводит в окно окружность, приближенную отрезками.

Основные технологии:

- основы работы со средой Qt Creator;
- основы создания приложений, которые используют Qt;
- графический вывод.

### 3.2.1. Создаём проект

Чтобы создать в Qt Creator новый проект (новое приложение):

- нажмите **Control+N** (**Файл** → **Новый**), появится окно **Новый**. Выберите в нём пункт **GUI приложение Qt4** в разделе **Проекты** и нажмите кнопку **ОК** (рис. 3.2.1);
- появится окно **Введение и размещение проекта**. В поле **Название** введите название проекта, а в поле **Создать в** – каталог, в котором будет храниться проект. Qt Creator создаст в указанном месте каталог с именем проекта, в котором будут храниться файлы приложения. Нажмите кнопку **Вперед** (рис. 3.2.2);
- в данном проекте дополнительные модули не нужны, поэтому в следующем окне нажмите **Вперед** (рис. 3.2.3);
- в окне **Информация о классе** в списке **Базовый класс** выберите **QWidget** и снимите галочку **Создать форму**. Нажмите **Вперед** (рис. 3.2.4);
- в последнем окне мастера нажмите кнопку **Финиш**, и Qt Creator создаст проект нового приложения с одним пустым окном (рис. 3.2.5).

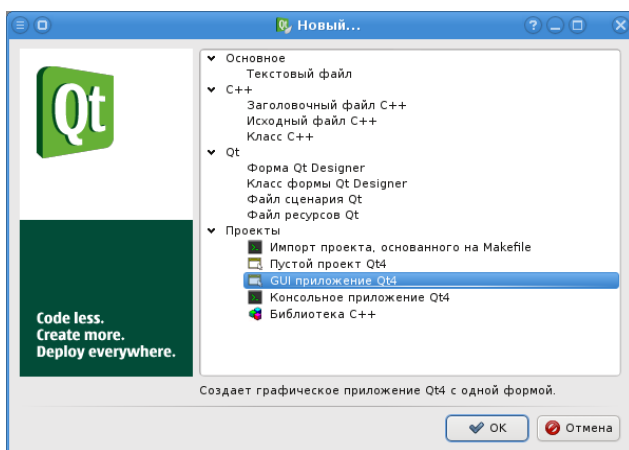


Рис. 3.2.1. Выбор вида проекта

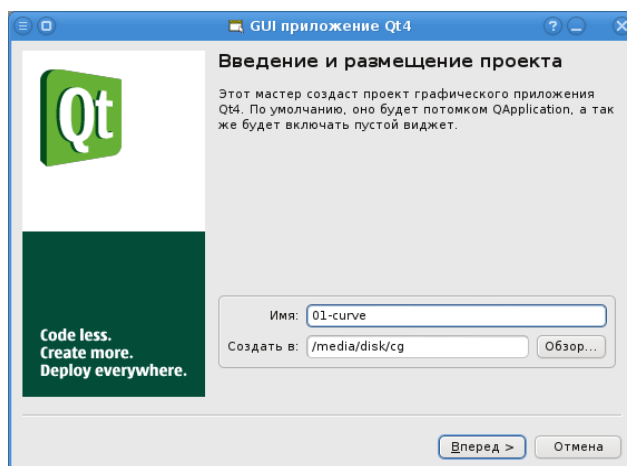


Рис. 3.2.2. Выбор расположения проекта

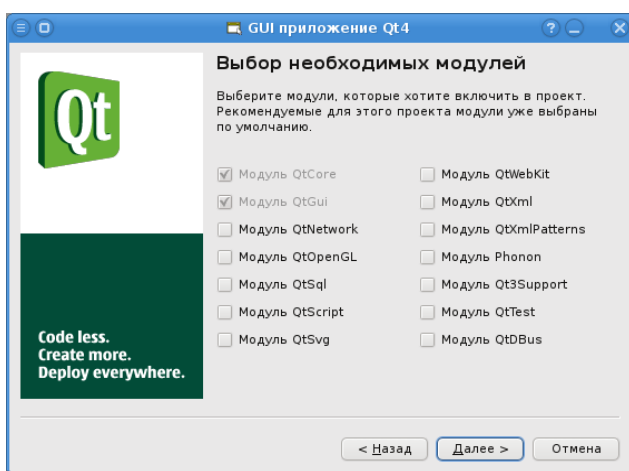


Рис. 3.2.3. Выбор дополнительных модулей Qt

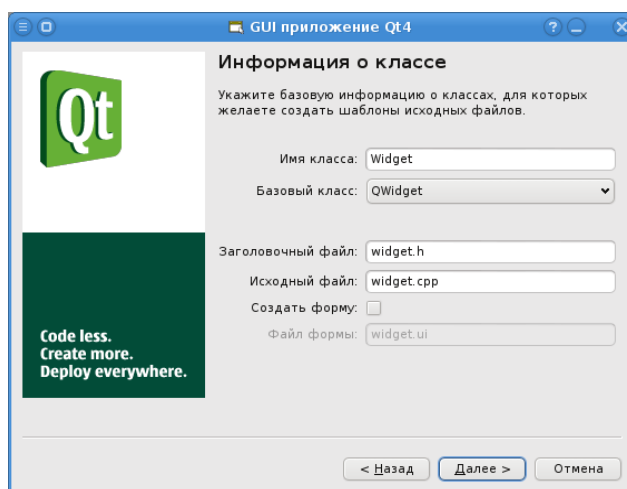


Рис. 3.2.4. Создание класса окна программы

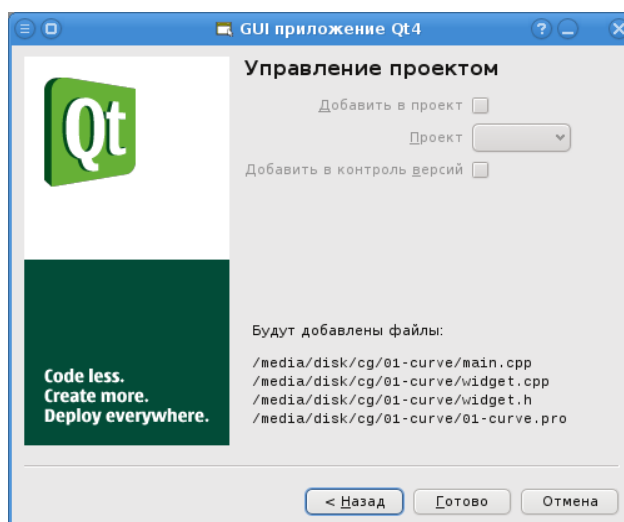




Рис. 3.2.5. Завершение создания проекта

### 3.2.2. Собираем и запускаем приложение

По созданному нами проекту можно сразу собрать работающее приложение; правда, единственное, что мы получим – пустое окно.

Чтобы собрать и запустить приложение нажмите **Control+R** или кнопку  в левом нижнем углу окна программы Qt Creator. Система соберёт приложение и запустит его. Появится пустое серое окно, с которым можно обращаться, как и с любым другим окном Windows: перемещать, изменять размер и т.п. Чтобы завершить работу этой простой программы, закройте её окно.

Если вам нужно только собрать программу (например, чтобы проверить, есть ли в ней ошибки), нажмите **Control+B** или кнопку  в левом нижнем углу окна Qt Creator. Если в программе есть ошибки, внизу окна откроется их список.

### 3.2.3. Структура проекта

В каталоге проекта находится несколько файлов и каталогов (ниже **01-curve** является именем проекта, у вас оно может быть другим – см. раздел 3.2.1 и рисунок 3.2.2):

- **01-curve.pro** – файл проекта, содержит основные свойства проекта Qt;
- **01-curve.pro.user** – файл настроек проекта программы Qt Creator;
- **main.cpp** – исходный файл, в котором находится функция `main`;
- **widget.h** – заголовочный файл с определением класса `Widget` – класса окна программы;
- **widget.cpp** – исходный файл с определением функций класса `Widget`;
- **Makefile** и **Makefile.\*** – вспомогательные файлы для сборки приложения;
- **debug** и **release** – каталоги с выходными файлами сборки;
- **debug\01-curve.exe** – файл приложения.

Чтобы перенести проект на другую машину (или в другую систему), где есть Qt, достаточно взять следующие файлы (остальные файлы и каталоги можно спокойно удалять – Qt Creator создаст их при необходимости сам):

- файл проекта (\*.pro);
- исходные файлы (\*.cpp и \*.h).

Если вы хотите запустить собранное приложение на другой Windows-машине, на которой не установлена Qt, вам понадобятся следующие файлы (**QTDIR** означает каталог, в который вы установили Qt SDK, – см. раздел 3.1 и рисунок 3.1.5):

- исполняемый файл вашего приложения (из каталога **debug** проекта);
- библиотека **QTDIR\mingw\bin\mingwm10.dll**;
- библиотеки **QTDIR\qt\bin\QtCored4.dll** и **QTDIR\qt\bin\QtGuid4.dll**.

### 3.2.4. Редактирование текста

Чтобы открыть исходный файл, достаточно дважды щёлкнуть по его названию в списке файлов слева (рис. 3.2.6). Файлы, открытые в данное время, отображаются в левом нижнем окне среды.

Чтобы сохранить файл, нажмите **Control+S** (**Файл → Сохранить**). Команда **Файл → Сохранить всё** сохранит все файлы проекта.

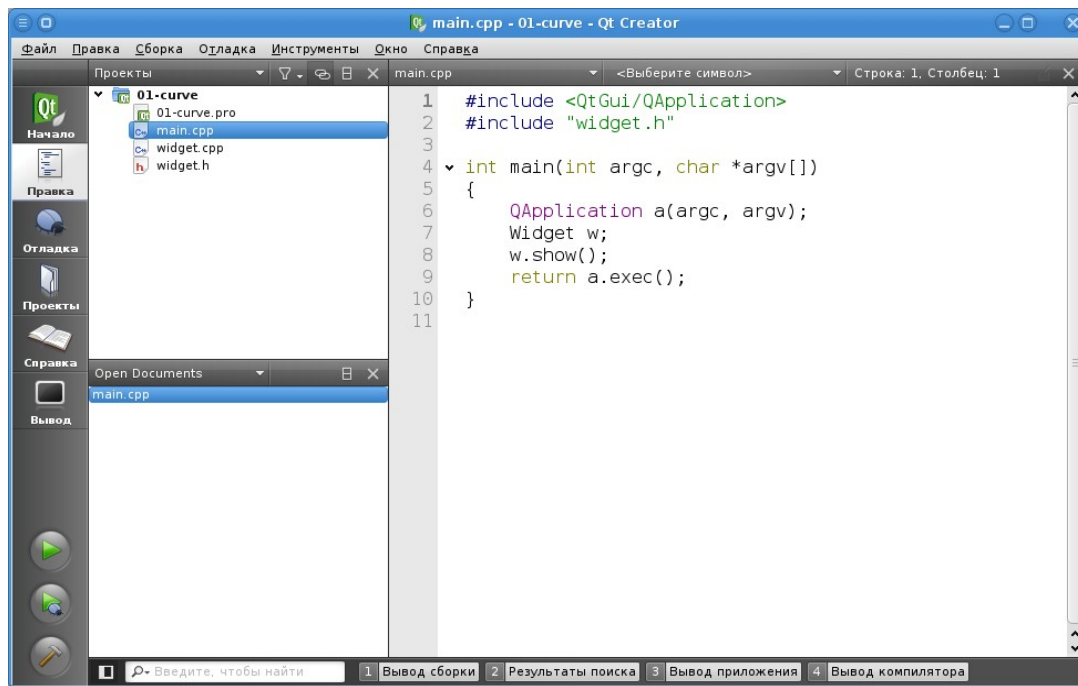
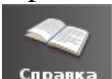
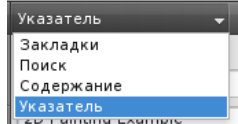
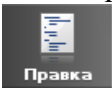


Рис. 3.2.6. Редактирование исходного текста в Qt Creator

### 3.2.5. Встроенный справочник

Среда Qt практически полностью документирована (на английском языке), за исключением некоторых примеров приложений. В среде есть удобный встроенный справочник. Чтобы воспользоваться им, можно нажать **Control+5** или кнопку  в крайней левой панели. В режиме справки в правой части окна отображается справочная информация, а в левой – служебная панель, в которой можно открыть содержание (Содержание), предметный указатель (Указатель), поиск (Поиск) или список закладок

(Закладки). Режим работы панели можно выбрать в списке  в её верхней части. Например, чтобы посмотреть документацию на класс QWidget, выберите режим **Указатель** и в строке поиска (Искать) введите QWidget и нажмите **Enter**.

Чтобы вернуться к редактированию текста программы, можно нажать **Control+2** или кнопку  в крайней левой панели.

Также справочную систему можно вызвать прямо из текстового редактора (см. рис. 3.2.7). Для этого наведите курсор (текстовый или мыши) на имя нужного класса Qt и нажмите **F1** (при этом необходимо, чтобы класс был описан в программе, то есть должен быть включён соответствующий заголовочный файл – см. раздел 3.2.6.1). Чтобы закрыть справочную панель, нажмите на серый крестик в её правом верхнем углу или два раза клавишу **Escape**.

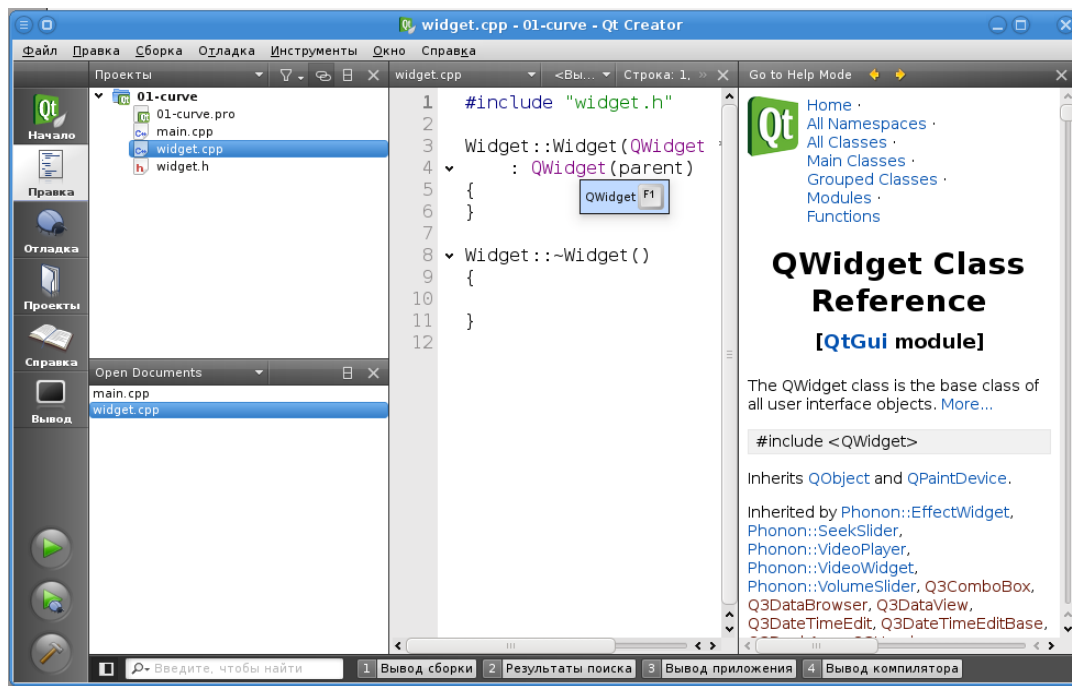


Рис. 3.2.7. Вызов контекстной справки в Qt Creator

## 3.2.6. Структура приложения

Рассмотрим подробнее программу, созданную средой, – её структура обычна для большинства приложений, которые используют Qt. Мы настоятельно рекомендуем параллельно с данными указаниями читать документацию на используемые классы и функции Qt, поскольку здесь они описаны вкратце.

### 3.2.6.1. Функция main

В файле **main.cpp** определена функция `main`. Обычно в Qt-приложениях она очень проста, как и в нашем примере:

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();
    return a.exec();
}
```

В начале функции мы создаём объект класса `QApplication`. Он представляет в нашей программе приложение Qt и должен существовать в единственном экземпляре.

После этого мы создаём объект-окно (нашего класса `Widget`) и показываем окно на экране.

И наконец, в последней инструкции мы запускаем основной цикл приложения – цикл обработки событий.



В Qt имена заголовочных файлов совпадают с именами классов, которые в них определены. Например, чтобы использовать класс `QApplication`, в начале файла достаточно написать

```
#include <QApplication>
```

### 3.2.6.2. Класс `Widget`

Класс `Widget` определён в файлах **`widget.h`** и **`widget.cpp`**. Этот класс представляет окно нашей программы и является производным от класса `QWidget`. `QWidget` – основной базовый класс для всех окон библиотеки Qt. Изначально у класса `Widget` есть только конструктор и деструктор:

```
class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();
};
```

Макрос `Q_OBJECT` добавляет служебные средства Qt в наш класс, он понадобится позже.

### 3.2.7. Рисуем окружность, аппроксимированную отрезками прямых

Современные программы с оконным интерфейсом обычно управляются событиями: от системы приходят некие сигналы (например, если пользователь нажал клавишу или изменил размер окна), а программа их обрабатывает.

Так же строится и процесс рисования. Система вызывает для окна событие «Перерисовка», в ответ на него программа может в этом окне рисовать.

Один из способов обработать системное событие в Qt – перегрузить виртуальную функцию-обработчик базового класса. В нашем случае необходимо перегрузить защищённую функцию `QWidget::paintEvent`:

```
class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

protected:
    void paintEvent(QPaintEvent*);
};
```

Сначала приведём текст нашей функции `Widget::paintEvent` (мы определили её в файле **widget.cpp**), а потом опишем средства Qt, которые в ней используются:

```
void Widget::paintEvent(QPaintEvent*)
{
    const double radius(150);
    const double step(0.1);

    const double pi(4 * atan(1));
    const double end(2 * pi + step);

    QPointF p1(radius, 0);

    QPainter ptr(this);
    ptr.setPen(Qt::blue);

    const QPointF center(width() / 2.0, height() / 2.0);

    for (double t(step); t < end; t += step)
    {
        QPointF p2(cos(t), sin(t));
        p2 *= radius;

        ptr.drawLine(p1 + center, p2 + center);

        p1 = p2;
    }
}
```

Функции `atan`, `sin` и `cos` – часть стандартной библиотеки языка Си++. Нужно лишь не забыть включить заголовочный файл `cmath`, чтобы дать компилятору их объявления.

Класс `QPointF` представляет двумерную точку с вещественными координатами (если необходимы целые координаты, используйте класс `QPoint`). В объекте `p1` мы храним первую точку отрезка, а в `p2` – вторую.

Для рисования в Qt используют класс `QPainter`. При создании его объекта (`ptr`) мы указываем, куда он будет рисовать (передаём указатель на своё окно). Функция `QPainter::setPen` устанавливает текущее перо. Перо описывает, как нужно рисовать линии, например каким цветом, какой толщины, с какой штриховкой. В программе выше мы задаём синее сплошное перо (без штриховки).

Для рисования отрезков прямых в классе `QPainter` есть несколько вариантов функции `drawLine`. Мы рисуем прямую по двум вещественным точкам.

Функции `width` и `height` принадлежат нашему родительскому классу `QWidget`. Первая возвращает текущую ширину внутренней (клиентской) области окна, а вторая – его текущую высоту.

Также в начале файла необходимо включить заголовок класса `QPainter`:

```
#include <QPainter>
```

Окно программы с окружностью показано на рисунке 3.2.8.

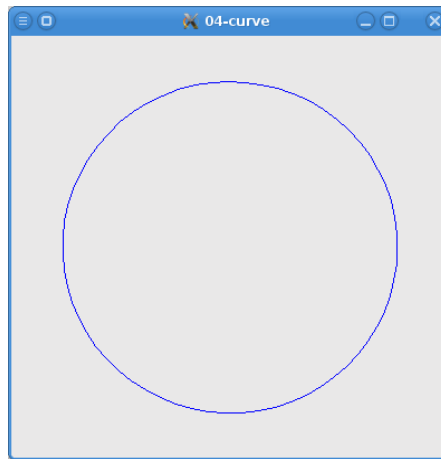


Рис. 3.2.8. Окно программы с нарисованной окружностью

## 3.3. Простое взаимодействие с пользователем

Здесь мы объясним, как можно получать от пользователя данные и реагировать на его команды. Пользователь сможет менять радиус окружности и шаг аппроксимации.

Основные технологии:

- создание и использование элементов управления;
- использование сигналов и слотов.

### 3.3.1. Создаём проект

Исходный проект для данного приложения аналогичен предыдущему (см. раздел 3.2.1).

### 3.3.2. Создаём элементы управления

В Qt (как и во многих других оконных библиотеках) элементы управления являются дочерними окнами основного окна. Обычно их создают с помощью оператора `new` и отдают во владение основному окну, оно же их и удалит при выходе из программы (самостоятельно удалять их с помощью `delete` в этом случае нельзя).

Существует множество средств для ввода данных. Нам необходимо вводить вещественные числа в определённом диапазоне. Для этого удобен класс `QDoubleSpinBox` (аналогичный класс для ввода целых чисел называется `QSpinBox`).

Для подписей к элементам обычно используют класс `QLabel` (он может отображать обычный текст, форматированный текст в формате HTML, картинку или анимацию).

#### 3.3.2.1. Создаём объекты

В конструкторе класса `Widget` создаём два поля ввода для вещественных чисел и две подписи к ним:

```
Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    QLabel* lblRadius(new QLabel("Radius:", this));
    lblRadius->move(10, 10);
```

```

radius = new QDoubleSpinBox(this);
radius->setRange(1, 300);
radius->setSingleStep(0.1);
radius->setValue(150);
radius->move(10, 30);

QLabel* lblStep(new QLabel("Step:", this));
lblStep->move(10, 50);

step = new QDoubleSpinBox(this);
step->setRange(0.01, 1);
step->setSingleStep(0.01);
step->setValue(0.1);
step->move(10, 70);
}

```

Всем элементам управления мы передаём в качестве родителя `this`, в таком случае основное окно становится ответственным за их удаление.

Поля ввода нужны при перерисовке окна, поэтому соответствующие указатели мы поместили внутрь класса (раздел 3.3.2.2). Функция `setRange` класса `QDoubleSpinBox` устанавливает возможный диапазон изменения чисел для соответствующего поля. С помощью функции `QDoubleSpinBox::setSingleStep` мы устанавливаем шаг изменения (пользователь может изменять значения в элементе `QDoubleSpinBox` дискретно, с помощью кнопок). Функция `QDoubleSpinBox::setValue` устанавливает текущее значение поля.

Функция `QWidget::move` меняет положение элемента относительно родительского окна.

В начале `cpp`-файла добавляем две команды `#include`:

```

#include <QLabel>
#include <QDoubleSpinBox>

```

### 3.3.2.2. Добавляем поля в класс

В класс окна имеет смысл вводить те элементы управления, которые используются в разных функциях в течение всего времени работы программы. В нашем случае окна ввода данных создаются в конструкторе и используются при рисовании, поэтому мы сохраняем их в самом классе:

```

class QDoubleSpinBox;

class Widget : public QWidget
{
    // ...

private:
    QDoubleSpinBox* radius;
    QDoubleSpinBox* step;
};

```

### 3.3.3. Связываем ввод данных с перерисовкой

Если собрать и запустить программу, поля ввода будут работать, значения изменяться, но окружность реагировать на них не будет. Необходимо как-то связать изменение значений в полях с рисованием окружности.

В разделе 3.2.7 описано, как получить и обработать событие оконной системы напрямую. В Qt есть более общий механизм, который позволяет передавать сообщения между объектами.

Если объект хочет сообщить о наступивших событиях, он выдаёт *сигналы*. Если объект может реагировать на какие-то события, он создаёт *слоты*. Мы можем *подключить* сигнал одного объекта к слоту другого (если у них одинаковые параметры). Тогда при выдаче сигнала будет вызван указанный слот, в параметрах (если они есть) он получит основные сведения о событии. К сигналу можно подключить несколько слотов. Кроме слотов, к сигналу можно подключить другие сигналы – событие будет передано по цепочке.

Сигналы и слоты являются обычными функциями-членами Си++, которые подчиняются некоторым дополнительным правилам. Например, слот можно вызвать напрямую как обычную функцию (если позволяет область видимости).

Класс, который определяет сигналы и/или слоты, должен быть прямым или косвенным наследником `QObject`. Также в начале описания такого класса в `private`-области должен присутствовать макрос `Q_OBJECT` (см. текст класса в разделе 3.2.7 или файле **widget.h**). Кроме того, название файла-заголовка с описанием класса должно присутствовать в переменной `HEADERS` в файле проекта (это обычный текстовый файл). В нашем случае все эти условия уже выполнены средой Qt Creator (класс `QWidget` – наследник класса `QObject`).

Подробнее о сигналах и слотах можно прочесть в документации Qt (раздел **Signals and Slots**).

У класса `QDoubleSpinBox` есть сигнал `valueChanged(double)`, который вызывается, когда пользователь меняет число в окне. В параметре передаётся новое числовое значение. У класса `QWidget` есть слот `update`, который перерисовывает окно (кроме прочего, он вызывает нашу функцию `paintEvent`). Он не принимает никаких параметров, но нам это и не нужно – наш класс хранит указатели на окна ввода. Это порождает проблему: у сигнала и слота разные наборы параметров.

#### 3.3.3.1. Подключаемся к событию

Чтобы решить задачу, мы создадим ещё один уровень косвенности – слот, который можно подключить к нашему полю ввода. Во-первых, его надо объявить в заголовочном файле:

```
class Widget : public QWidget
{
    // ...

private slots:
    void redrawOnValueChange(double);
};
```

Слот (как и другие функции) может быть открытым, защищённым или закрытым, но его нужно объявить в специальном разделе: `public slots`, `protected slots` или `private slots` соответственно. Мы не будем использовать передаваемое число, поэтому

имя параметра опущено.

Во-вторых, функцию необходимо реализовать в сpp-файле. Она очень проста:

```
void Widget::redrawOnValueChange(double)
{
    update();
}
```

Осталось только связать сигналы с нашим слотом. Это делает функция `QObject::connect` (SIGNAL и SLOT – специальные макросы):

```
Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    // ...0
    connect(radius, SIGNAL(valueChanged(double)),
            this, SLOT(redrawOnValueChange(double)));

    connect(step, SIGNAL(valueChanged(double)),
            this, SLOT(redrawOnValueChange(double)));
}
```

### 3.3.3.2. Применяем пользовательские значения при рисовании

При рисовании мы используем значения из полей ввода (его возвращает функция `QDoubleSpinBox::value`). Новая функция `Widget::paintEvent`:

```
void Widget::paintEvent(QPaintEvent*)
{
    const double rad(radius->value());
    const double st(step->value());

    const double pi(4 * atan(1));
    const double end(2 * pi + st);

    QPointF p1(rad, 0);

    QPainter ptr(this);
    ptr.setPen(Qt::blue);

    const QPointF center(width() / 2.0, height() / 2.0);

    for (double t(st); t < end; t += st)
    {
        QPointF p2(cos(t), sin(t));
        p2 *= rad;

        ptr.drawLine(p1 + center, p2 + center);

        p1 = p2;
    }
}
```

Не забудьте объявить её в классе (если вы этого ещё не сделали) – см. также раздел 3.2.7.

### 3.3.4. Запускаем программу

Работающая программа показана на рисунке 3.3.1.

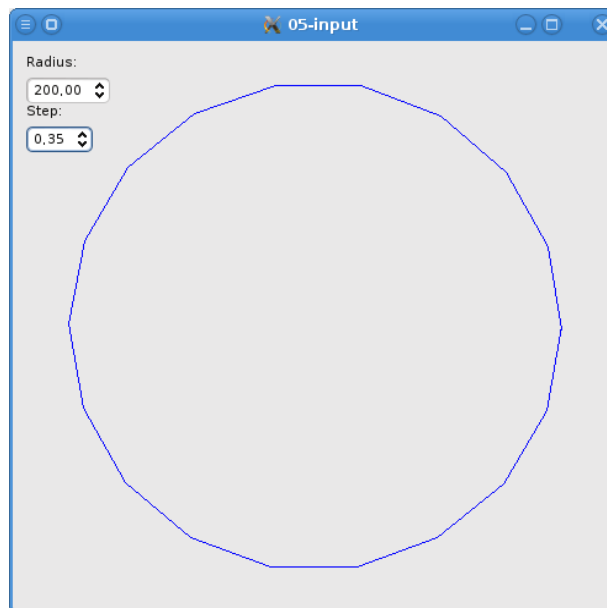


Рис. 3.3.1. Окно программы с изменёнными параметрами

Если окно программы слишком маленькое, можно изначально установить подходящий размер – например, в функции `main`:

```
// ...  
  
Widget w;  
w.resize(400, 400);  
w.show();  
  
// ...
```

## 3.4. Гибкое управление интерфейсом

Qt – мощная библиотека, которая позволяет легко создавать сложные и гибкие программные системы. Этот и следующий разделы посвящены двум средствам, которые облегчают создание сложных графических человеко-машинных интерфейсов.

Основные технологии:

- модульное построение приложения;
- механизм сигналов и слотов;
- механизм размещения элементов интерфейса.

### 3.4.1. Создаём проект

Исходный проект создаётся аналогично двум предыдущим (см. раздел 3.2.1), но использовать заготовку мы будем немного по-другому.

### 3.4.2. Краткий обзор приложения

Из предыдущего примера видно, что интерфейс нашей программы состоит из двух основных частей: области рисования и области управления или ввода данных. Выразим это напрямую: сделаем их разными объектами. Также нам понадобится организовать их взаимодействие. Такой подход сделает программу гибкой, расширяемой. Независимые модули легче использовать, документировать, поддерживать, переносить в другие приложения. Например, в разделе 3.5 мы сможем легко расширить возможности нашей программы, сделав её больше похожей на профессиональное приложение.

Qt Creator создала пустой класс `Widget`, который в данном случае является основным окном приложения. В этом примере вся настоящая работа выполняется в двух других окнах, а класс `Widget` объединяет их в единое целое.

### 3.4.3. Создаём управляющую панель

Управляющая панель позволит менять параметры объекта. Она будет содержать поля ввода и подписи к ним. Также на ней мы покажем, как работать с механизмом размещения элементов интерфейса Qt.

В прошлом примере мы давали элементам управления жёсткое положение и жёсткий размер (который определялся автоматически). Этот подход имеет ряд недостатков. Вот некоторые из них:

- добавляя, убирая элементы или меняя их местами, нужно вручную менять положение других элементов;
- чтобы переместить группу элементов внутри родительской области, нужно изменить координаты всех элементов группы;
- при изменении размера отдельных элементов необходимо сдвигать остальные;
- если меняется размер родительского окна (области), нужно отслеживать это и пересчитывать размеры (и, возможно, положения) элементов.

Механизм размещения (layout engine) освобождает программиста от всех этих неудобств. Положение и размеры элементов управляются специальным объектом. Программист лишь указывает, как разместить их, и может управлять их относительными размерами, расстояниями между ними и вокруг них и прочими общими параметрами; точные координаты и размеры элементов рассчитывает Qt.

Для управляющей области мы создадим отдельный класс `Panel`. Чтобы создать для него новую пару файлов (заголовок и реализация):

- нажмите клавиши **Control+N** (**Файл → Новый**) – появится окно **Новый**. В разделе **C++** выберите пункт **Класс C++** (рис. 3.4.1) и нажмите кнопку **ОК**;
- в окне **Введите имя класса** в поле **Имя класса** введите `Panel`, в списке **Базовый класс** выберите **QWidget** (рис. 3.4.2). Нажмите **Вперед**;
- нажмите **Финиш** (рис. 3.4.3), чтобы создать заготовку для нового класса. В списке файлов проекта появятся **panel.h** и **panel.cpp** (рис. 3.4.4).



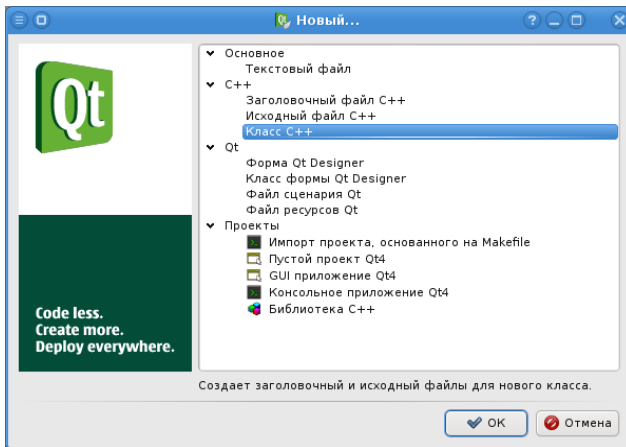


Рис. 3.4.1. Создание нового класса

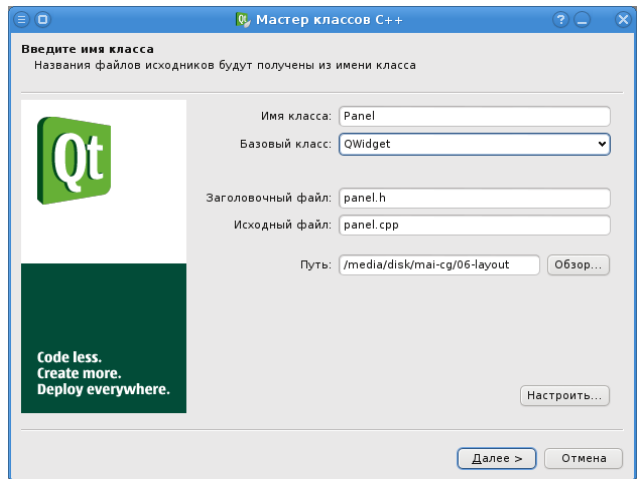


Рис. 3.4.2. Ввод параметров

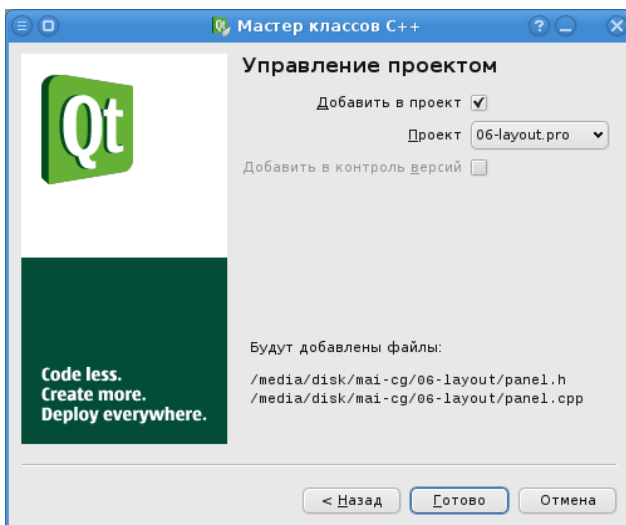


Рис. 3.4.3. Завершение создания класса

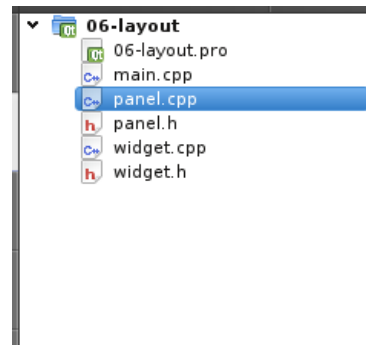


Рис. 3.4.4. Новые файлы в составе проекта

Текст класса Panel (из файла **panel.h**):

```
class QdoubleSpinBox;

class Panel : public QWidget
{
    Q_OBJECT
public:
    Panel(QWidget* parent = 0);
    double radius() const;
    double step() const;

signals:
    void radiusChanged(double);
    void stepChanged(double);

private:
    QDoubleSpinBox* rad;
    QDoubleSpinBox* st;
};
```

Функции `radius` и `step` выдают текущие значения из соответствующих полей ввода (радиус окружности и шаг разбиения).

`radiusChanged` и `stepChanged` – это сигналы, сообщающие внешним объектам об изменении соответствующих параметров (см. также раздел 3.3.3). Они объявляются в специальном разделе `signals` и реализуются самой Qt. Мы можем лишь вызывать их и подключать к ним слоты или другие сигналы.

Реализация функций класса (из **panel.cpp**):

```
#include <QLabel>
#include <QDoubleSpinBox>
#include <QVBoxLayout>

Panel::Panel(QWidget* parent) : QWidget(parent)
{
    QLabel* lblRadius(new QLabel("Radius:"));

    rad = new QDoubleSpinBox;
    rad->setRange(1, 300);
    rad->setSingleStep(0.1);
    rad->setValue(150);

    QLabel* lblStep(new QLabel("Step:"));

    st = new QDoubleSpinBox;
    st->setRange(0.01, 1);
    st->setSingleStep(0.01);
    st->setValue(0.1);

    QVBoxLayout* lout(new QVBoxLayout);
    lout->addWidget(lblRadius);
    lout->addWidget(rad);
    lout->addWidget(lblStep);
    lout->addWidget(st);
    lout->addStretch();

    setLayout(lout);

    connect(rad, SIGNAL(valueChanged(double)),
            this, SIGNAL(radiusChanged(double)));
    connect(st, SIGNAL(valueChanged(double)),
            this, SIGNAL(stepChanged(double)));
}

double Panel::radius() const
{
    return rad->value();
}

double Panel::step() const
{
    return st->value();
}
```

Как и в предыдущем примере, в конструкторе окна мы создаём дочерние элементы управления: два поля ввода для вещественных чисел и две подписи к ним. В этот раз мы не передаём им «родителя» (указатель на объект своего окна) явно: это сделает размещающий объект.

Размещением элементов в данном примере занимается класс `QVBoxLayout`. Он располагает элементы по вертикали, в «столбец», в том порядке, в котором они ему передаются. Чтобы расположить элементы в ряд (по горизонтали) или в виде «таблицы», можно использовать `QHBoxLayout` или `QGridLayout` соответственно.

По умолчанию `QVBoxLayout` равномерно распределит их по высоте родительского окна. Чтобы «собрать» их вверху, мы с помощью функции `QVBoxLayout::addStretch` снизу добавляем пустой «элемент», занимающий максимально возможное свободное место.

Функция `QWidget::setLayout` делает объект-размещение основным содержимым окна. `QWidget` забирает владение этим объектом и всеми элементами внутри него и при необходимости самостоятельно удалит их.

В конце конструктора мы с помощью функции `QObject::connect` соединяем сигналы полей ввода, которые сообщают об изменении значений, со своими сигналами: как только вызовется сигнал в поле ввода, вызовется наш сигнал, который передаст сообщение дальше, внешним объектам.

### 3.4.4. Создаём область рисования

В данной программе рисованием занимается класс `View`. В разделе 3.4.3 описано, как создать заготовку для нового класса. В качестве базового класса мы используем `QWidget`.

Определение класса (**view.h**):

```
class Panel;

class View : public QWidget
{
public:
    View(QWidget* parent = 0);

    const Panel* controlPanel() const;
    void setControlPanel(const Panel* p);

protected:
    void paintEvent(QPaintEvent*);

private:
    const Panel* pan;
};
```

Класс `View` лишь рисует окружность. Его функция `paintEvent` очень похожа на аналогичную из предыдущего примера, только теперь, вместо полей ввода, мы работаем с объектом класса `Panel`. Реализация класса `View` (из **view.cpp**):

```

#include "panel.h"
#include <QPainter>
#include <cmath>

View::View(QWidget* parent) : QWidget(parent), pan(0)
{
    QPalette pal(palette());
    pal.setColor(QPalette::Window, Qt::white);
    setPalette(pal);

    setAutoFillBackground(true);
}

void View::paintEvent(QPaintEvent*)
{
    if (!pan)
        return;

    const double rad(pan->radius());
    const double st(pan->step());

    const double pi(4 * atan(1));
    const double end(2 * pi + st);
    QPointF p1(rad, 0);

    QPainter ptr(this);
    ptr.setPen(Qt::blue);

    const QPointF center(width() / 2.0, height() / 2.0);

    for (double t(st); t < end; t += st)
    {
        QPointF p2(cos(t), sin(t));
        p2 *= rad;

        ptr.drawLine(p1 + center, p2 + center);

        p1 = p2;
    }
}

const Panel* View::controlPanel() const
{
    return pan;
}

void View::setControlPanel(const Panel* p)
{
    pan = p;
    update();
}

```

В конструкторе мы настраиваем цвет фона. Все основные цвета окна собраны в класс `QPalette` и распределены по своему назначению. Цвет, соответствующий `QPalette::Window`, отвечает за общий фон. Функция `QWidget::palette` выдаёт текущую палитру окна, а `QWidget::setPalette` меняет её. Функция `QWidget::setAutoFillBackground` включает (или, с параметром `false`, отключает) автоматическое заполнение фона окна.

### 3.4.5. Собираем всё вместе

Нам осталось написать класс `Widget` – он представляет основное окно и соединяет самостоятельные модули приложения в единое целое:

```
class View;

class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
    ~Widget();

private slots:
    void redrawOnValueChange(double);

private:
    View* view;
};
```

Внешне класс напоминает своего «тёзку» из предыдущего примера (раздел 3.3), но теперь из него ушло всё лишнее. Особенно это заметно на реализации класса:

```
#include "view.h"
#include "panel.h"
#include <QHBoxLayout>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    Panel* pan(new Panel);

    view = new View;
    view->setControlPanel(pan);

    QHBoxLayout* lout(new QHBoxLayout);
    lout->addWidget(view, 3);
    lout->addWidget(pan, 1);
    setLayout(lout);

    connect(pan, SIGNAL(radiusChanged(double)),
            this, SLOT(redrawOnValueChange(double)));
    connect(pan, SIGNAL(stepChanged(double)),
            this, SLOT(redrawOnValueChange(double)));
}
```

```
void Widget::redrawOnValueChange(double)
{
    view->update();
}
```

Здесь мы используем горизонтальное размещение (QHBoxLayout). Второй параметр функции `QBoxLayout::addWidget` задаёт относительный размер окна (в данном случае – по горизонтали) в частях от суммарного размера. В данном случае графическая область занимает три части, а панель – одну (75 и 25% соответственно) от ширины окна.

На рисунке 3.4.5 показаны снимки работающего приложения. На нём видно, что внутренние связи между областями работают, и видно также, как меняются размеры и положения областей и элементов в зависимости от размеров окна.

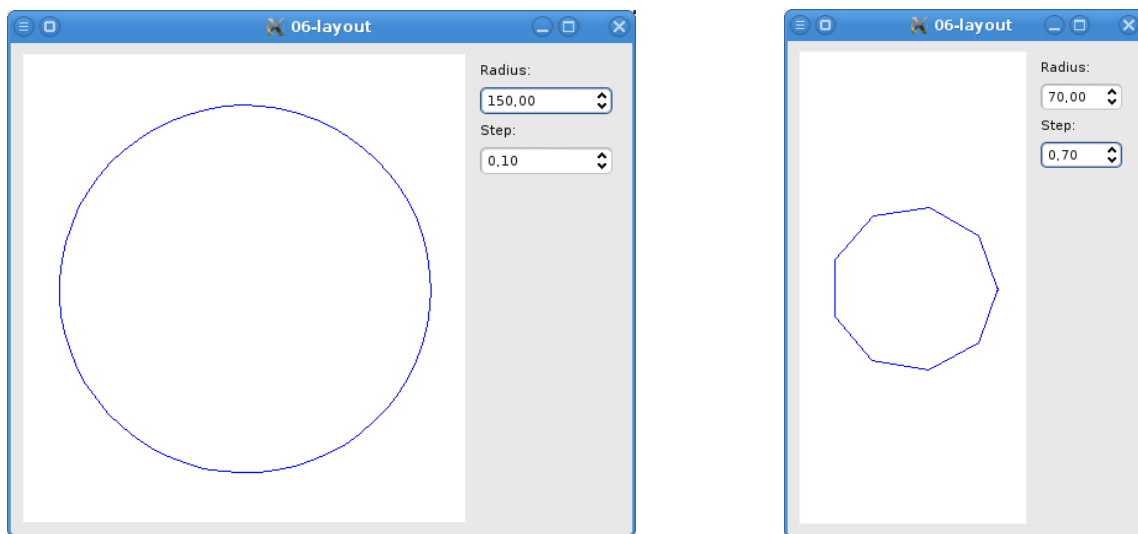


Рис. 3.4.5. Окно приложения с разными параметрами и размерами

## 3.5. Расширение возможностей приложения

В данном разделе мы покажем, как придать приложению более законченный вид. Мы добавим к нему строку меню, панель инструментов, строку состояния, а также сделаем управляющую панель «плавающей».

Основные технологии:

- механизм «главного окна» (main window);
- механизм «плавающих» окон (dock/floating window).

### 3.5.1. Создаём проект

В данном случае в создании проекта есть небольшое изменение – основное окно будет наследником класса `QMainWindow`, а не `QWidget`.

Чтобы создать проект в Qt Creator:

- нажмите **Control+N** (**Файл** → **Новый**) – появится окно **Новый**. Выберите в нём пункт **GUI приложение Qt4** в разделе **Проекты** и нажмите кнопку **ОК** (рис. 3.5.1);
- появится окно **Введение и размещение проекта**. В поле **Название** введите название проекта, а в поле **Создать в** – каталог, в котором будет храниться проект. Qt Creator создаст в указанном месте каталог с именем проекта, в котором будут храниться файлы приложения. Нажмите кнопку **Вперед** (рис. 3.5.2);

- в следующем окне (**Выбор необходимых модулей**) нажмите **Вперед** (рис. 3.5.3);
- в окне **Информация о классе** снимите галочку **Создать форму** (рис. 3.5.4). В этот раз базовым классом для нашего окна будет QMainWindow, поэтому остальные настройки мы оставляем как есть. Нажмите **Вперед**;
- в последнем окне мастера нажмите кнопку **Финиш**, и Qt Creator создаст проект нового приложения с одним пустым окном (рис. 3.5.5).

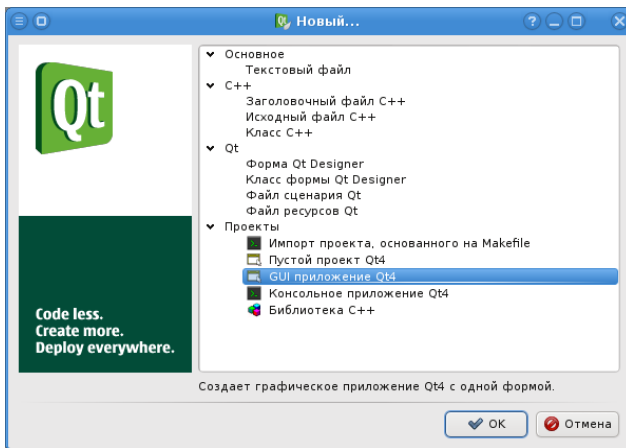


Рис. 3.5.1. Выбор вида проекта

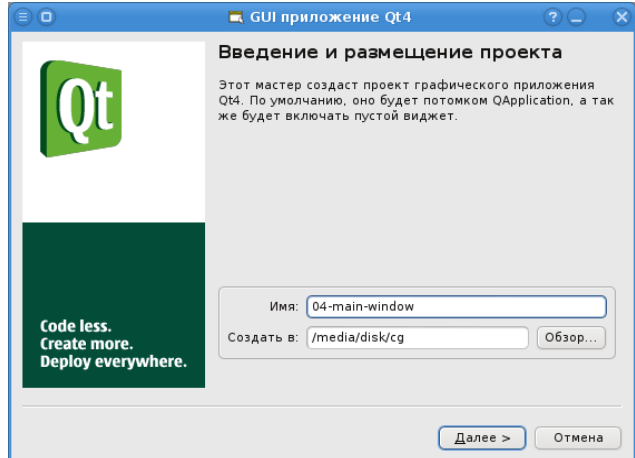


Рис. 3.5.2. Выбор расположения проекта

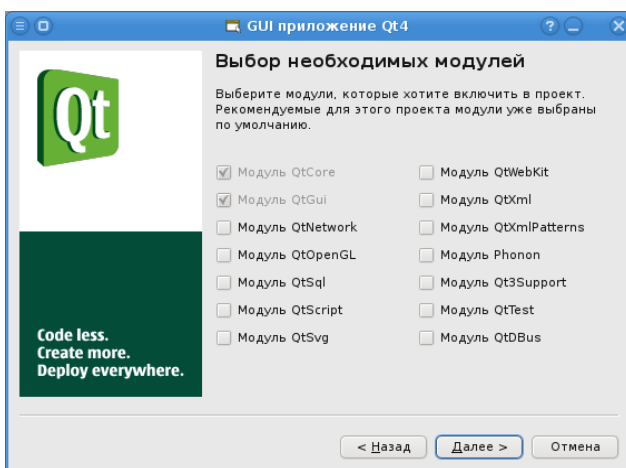


Рис. 3.5.3. Выбор дополнительных модулей Qt

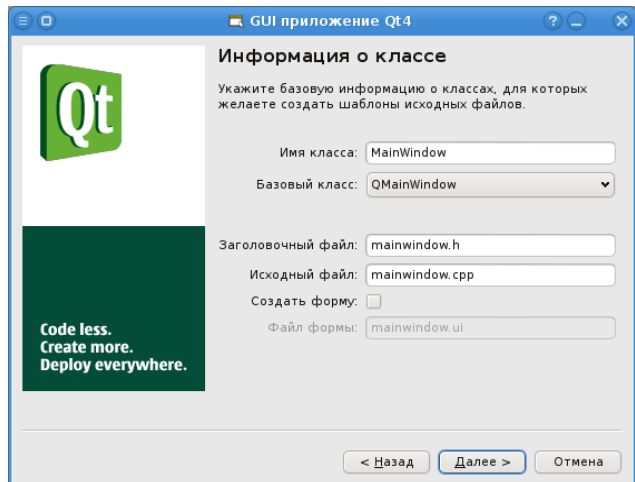


Рис. 3.5.4. Создание класса окна программы

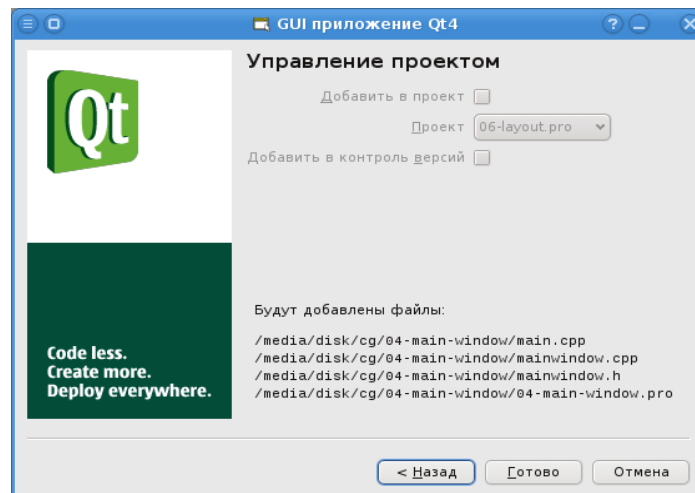


Рис. 3.5.5. Завершение создания проекта

### 3.5.2. Добавляем существующие файлы в проект

В прошлом разделе мы выделили окно рисования и управляющую панель в разные классы и теперь можем использовать их в данном проекте:

- скопируйте существующие файлы **view.h**, **view.cpp**, **panel.h** и **panel.cpp** из каталога с предыдущим проектом в каталог данного проекта;
- в панели **Проекты** среды Qt Creator щёлкните правой кнопкой по названию проекта и выберите команду **Добавить существующие файлы** (рис. 3.5.6). Появится окно открытия файлов; выберите в нём четыре скопированных файла и нажмите кнопку **Открыть** (рис. 3.5.7). Файлы добавятся в проект.

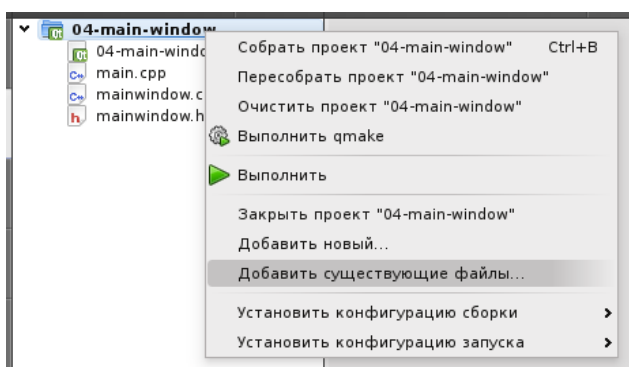


Рис. 3.5.6. Контекстное меню проекта

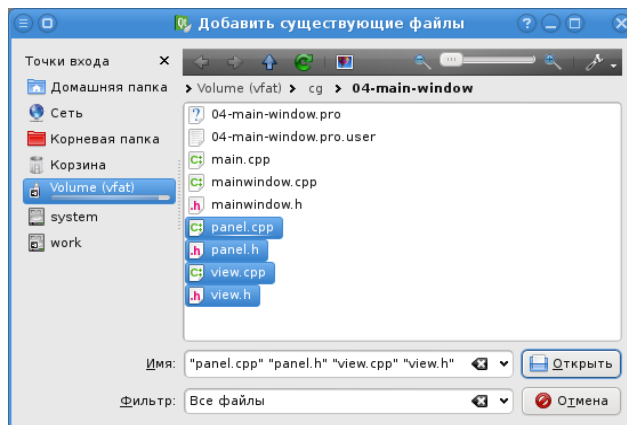


Рис. 3.5.7. Выбор файлов

### 3.5.3. Добавляем окна в основное окно приложения

Класс `MainWindow` является основным окном нашего приложения. Его определение очень похоже на определение класса `Widget` из предыдущего примера. Отрывок из **mainwindow.h**:

```
class View;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void redrawOnValueChange(double);

private:
    View* view;
};
```



Конструктор класса выглядит иначе, чем прежде. Приведём реализации функций (**mainwindow.cpp**):

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    Panel* pan(new Panel);

    QDockWidget* dock(new QDockWidget("Controls"));
    dock->setWidget(pan);

    addDockWidget(Qt::RightDockWidgetArea, dock);

    view = new View;
    view->setControlPanel(pan);

    setCentralWidget(view);

    connect(pan, SIGNAL(radiusChanged(double)),
            this, SLOT(redrawOnValueChange(double)));
    connect(pan, SIGNAL(stepChanged(double)),
            this, SLOT(redrawOnValueChange(double)));
}

void MainWindow::redrawOnValueChange(double)
{
    view->update();
}
```

В отличие от обычных окон, с которыми мы работали до этого (класс `QWidget`), окно класса `QMainWindow` устроено несколько по-другому. У главного окна есть центральная область (`central widget`), которая несёт основную смысловую нагрузку, четыре внутренних области (верх, правая сторона, низ, левая сторона) для привязки плавающих окон (`dock widgets`), четыре внешних области для панелей инструментов (`toolbars`), область для строки меню (`menu bar` – сверху) и для строки состояния (`status bar` – снизу). На рисунке 3.5.8 показана соответствующая схема из документации Qt.

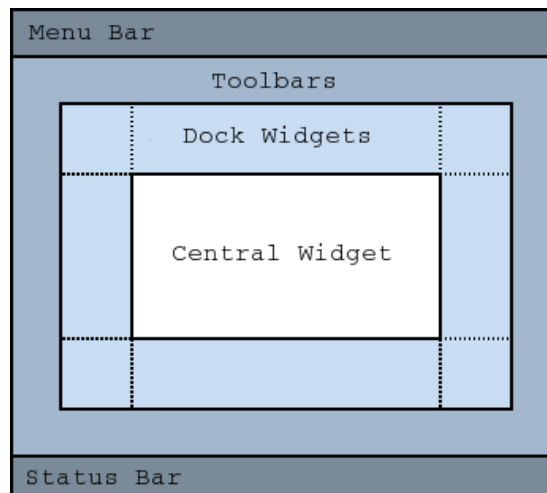


Рис. 3.5.8. Расположение областей в главном окне

Класс `QDockWidget` представляет плавающее окно, которое можно привязать к главному. Окно, представленное `QDockWidget`, является лишь оболочкой, функция `QDockWidget::setWidget` задаёт его содержимое. Функция `QMainWindow::addDockWidget` передаёт вспомогательное окно главному, первый параметр задаёт начальную область привязки (в данном случае – правая). Плавающему окну можно также задать области главного окна, к которым его разрешено привязывать.

Функция `QMainWindow::setCentralWidget` помещает окно в центральную область главного.

Все динамически созданные объекты переданы своим владельцам, удалять их будет библиотека.

Снимки работающего приложения показаны на рисунке 3.5.9. Теперь пользователь может давать панели любой размер, помещать её с любой стороны окна, вытаскивать из него, делая отдельным окном, и даже закрывать.

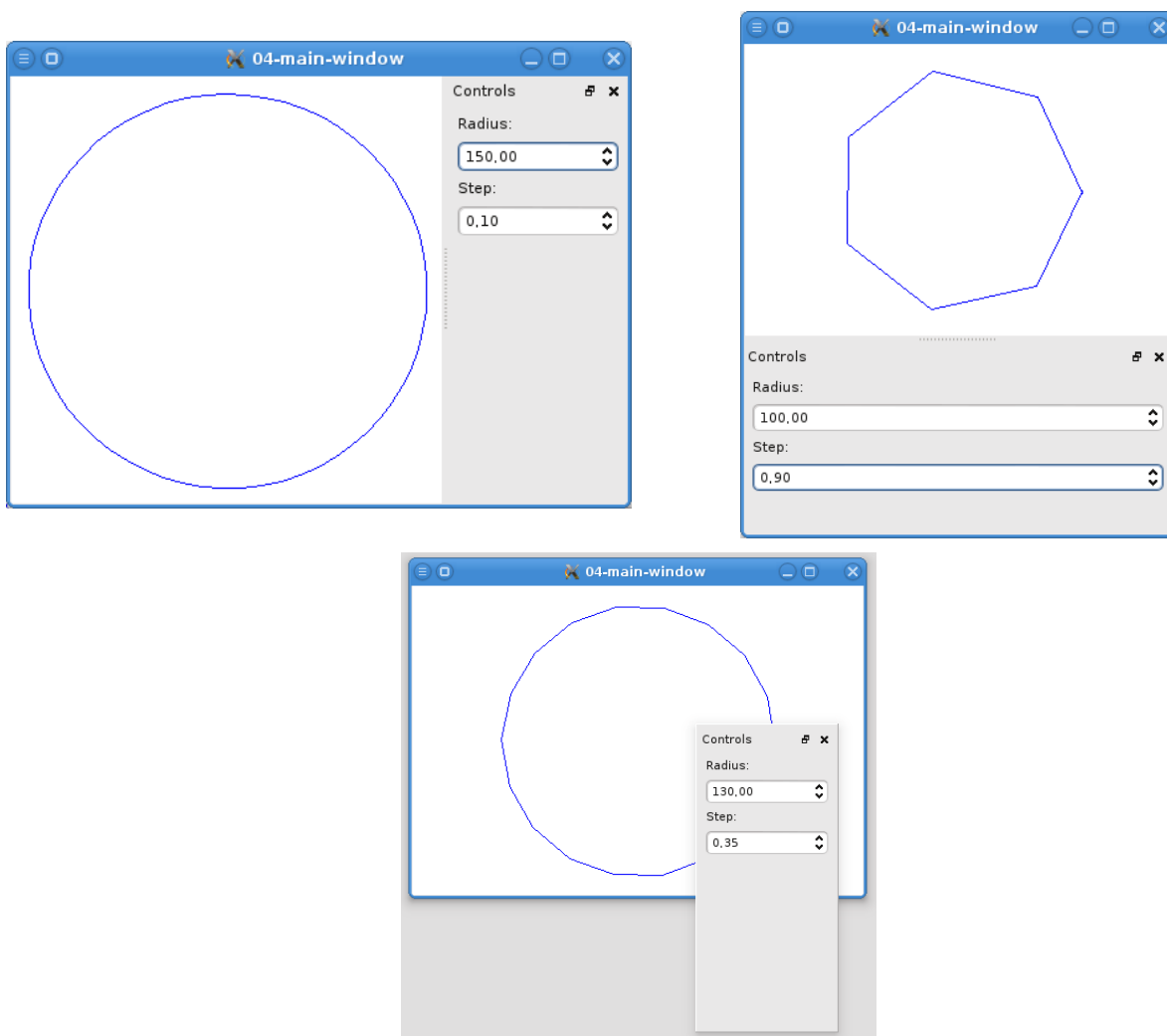


Рис. 3.5.9. Примеры работы программы

### 3.5.4. Создаём основное меню

В Qt пользовательские команды являются самостоятельными объектами и называются *действиями* (action). Действие можно поместить в меню или на панель инструментов, и оно будет показано как команда или кнопка с названием и значком. Действию можно назначить

комбинацию клавиш, работающую в окне, к которому оно привязано. Пользователь получает несколько способов для вызова команды, но в любом случае действие остаётся одним и не зависит от них, и в любом случае вызовется один и тот же сигнал, с которым можно соединить нужные функции.

Изменения коснутся только конструктора `MainWindow`. Также не забудьте включить нужные заголовочные файлы. Дополнения в `mainwindow.cpp`:

```
// ...


QAction* quitAct(new QAction(QIcon("quit.png"),
    "&Quit", this));
quitAct->setShortcut(QKeySequence("Ctrl+Q"));
quitAct->setToolTip("Quit application");
quitAct->setStatusTip("Closes the application");

connect(quitAct, SIGNAL(triggered()), qApp, SLOT(quit()));

QMenu* fileMenu(menuBar()->addMenu("&File"));
fileMenu->addAction(quitAct);

QAction* panelAct(dock->toggleViewAction());
panelAct->setStatusTip("Toggle panel");

QMenu* viewMenu(menuBar()->addMenu("&View"));
viewMenu->addAction(panelAct);
}
```

Класс `QAction` представляет действие. При создании ему передаются значок (в данном случае – **quit.png**: ) , текст команды и объект-владелец. Амперсанд (&) в строке стоит перед буквой, которая окажется подчёркнутой в имени пункта меню (быстрая клавиша). Файл со значком необходимо поместить в каталог проекта.

Функция `QAction::setShortcut` устанавливает комбинацию клавиш. Можно задать комбинацию явно или указать стандартный тип команды (см. документацию на `QKeySequence`); последний способ позволяет установить комбинацию клавиш, которая является стандартной для данной операции в данной оконной системе (например, **F1** в Windows и **Command+?** в Mac OS X для вызова справочной системы).

Функция `QAction::setToolTip` устанавливает текст всплывающей подсказки. Она появляется, например, когда пользователь наводит курсор мыши на кнопку панели инструментов. Функция `QAction::setStatusTip` устанавливает текст подсказки, которая появляется в строке состояния, если навести курсор мыши на пункт меню или кнопку на панели.

Сигнал `QAction::triggered` вызывается, когда пользователь вызывает это действие (любым способом, например через комбинацию клавиш). `qApp` является указателем на единственный (в приложении) объект класса `QApplication`, который создаётся в функции `main`. Функция-слот `QApplication::quit` завершает работу приложения.

В основном окне может быть одна строка меню. Она представлена классом `QMenuBar`. Функция `QMainWindow::menuBar` создаёт строку меню (если она ещё не создана) и возвращает указатель на соответствующий объект. Чтобы добавить пункт с выпадающим подменю, мы вызываем `QMenuBar::addMenu` и передаём ей название пункта меню. Класс

QMenu представляет всплывающее меню, с помощью `QMenu::addAction` мы добавляем в него пункты; разделитель можно вставить с помощью `QMenu::addSeparator`.

Чтобы пользователь легко мог скрывать и отображать панель с элементами управления, мы ввели ещё одно подменю (View) и команду, которая будет включать и отключать панель. У плавающих окон и панелей инструментов есть функция `toggleViewAction`, которая возвращает указатель на готовый объект-действие, служащий для данной цели.

### 3.5.5. Создаём панель инструментов и строку состояния

Наполнение для панели инструментов и строки состояния у нас уже готово. Осталось создать их. Приведём соответствующий отрывок конструктора `MainWindow`:

```
// ...

QToolBar* bar(addToolBar("Application"));
bar->addAction(quitAct);

statusBar()->showMessage("Ready");
}
```

Функция `QMainWindow::addToolBar` создаёт панель инструментов с указанным названием и возвращает указатель на объект `QToolBar`. Функция `QToolBar::addAction` добавляет элемент на панель инструментов.

Функция `QMainWindow::statusBar` аналогична `QMainWindow::menuBar`. Она создаёт строку состояния в данном окне (если её ещё нет) и возвращает указатель на объект класса `QStatusBar`. В нашем случае достаточно было бы просто её вызвать, но мы, кроме этого, сразу показываем в строке состояния временное сообщение о готовности нашего приложения к работе (`QStatusBar::showMessage`).

### 3.5.6. Результат

На рисунке 3.5.10 показаны снимки окна работающего приложения и его элементов.

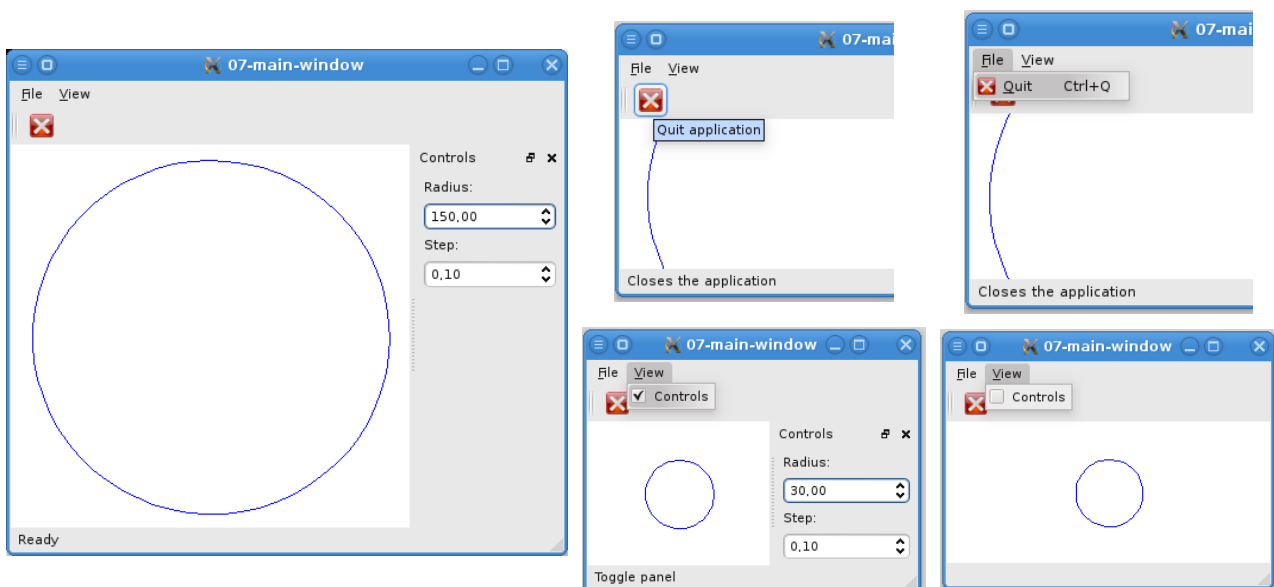


Рис. 3.5.10. Снимки готового приложения

Если выбрать пункт меню **File → Quit** или нажать кнопку на панели инструментов, или нажать на клавиатуре **Control+Q**, вызовется сигнал `triggered` соответствующего объекта `QAction`. К сигналу подключена функция `QApplication::quit`, которая завершит работу программы. Такая последовательность типична при обработке пользовательского ввода в приложениях, использующих Qt.

## 3.6. Система ресурсов Qt

В предыдущем примере используется картинка для кнопки на панели инструментов и пункта меню. В программе мы указали обычное имя файла, поэтому этот файл должен всегда находиться в одном каталоге с приложением. Можно избавиться от этой зависимости, поместив картинку в сам исполняемый файл. Для этого служит система ресурсов Qt (resource system).

В данном примере необходим проект, созданный в предыдущем разделе.

Основные технологии:

- система ресурсов.

### 3.6.1. Создаём файл ресурсов

Чтобы создать файл для описания ресурсов в Qt Creator:

- если предыдущий проект не открыт, его нужно открыть. Для этого нажмите **Control+O** (**Файл → Открыть**), найдите каталог с проектом и выберите файл проекта (с расширением `pro`);
- нажмите **Control+N** (**Файл → Новый**) – появится окно **Новый**. Выберите в нём в разделе **Qt** пункт **Файл ресурсов Qt** (рис. 3.6.1) и нажмите кнопку **ОК**;
- в окне **Выбор размещения** введите название файла в поле **Имя** (рис. 3.6.2) и нажмите **Вперед**;

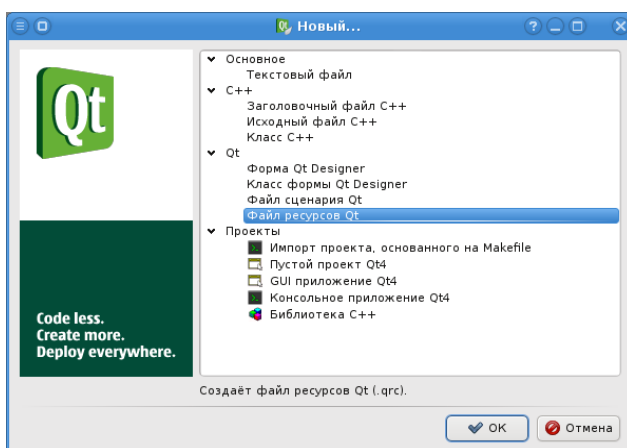


Рис. 3.6.1. Создание файла ресурсов

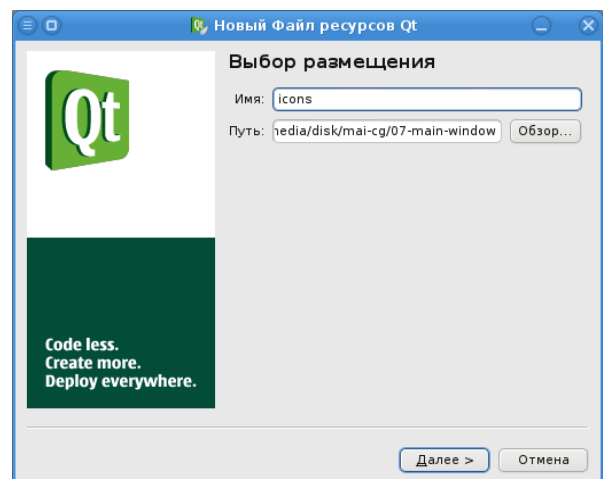


Рис. 3.6.2. Ввод параметров файла

- в окне **Управление проектом** (рис. 3.6.3) нажмите кнопку **Финиш**. Среда создаст пустой файл ресурсов, добавит его в проект и откроет редактор ресурсов.

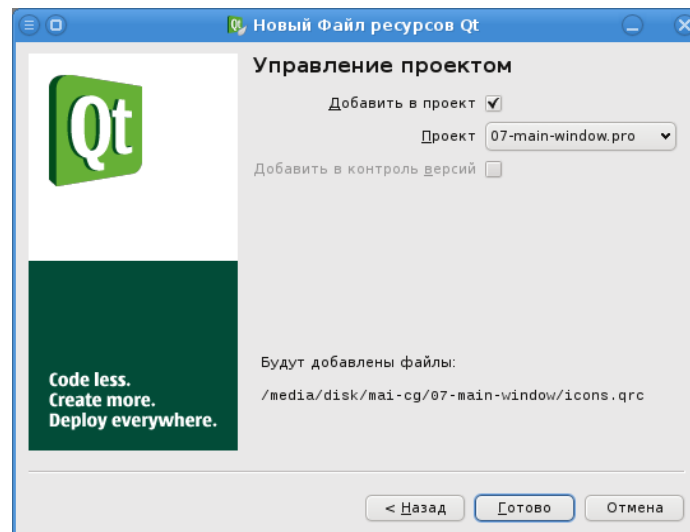


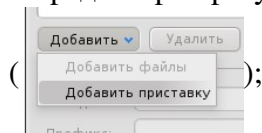
Рис. 3.6.3. Сведения о файле

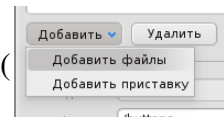
### 3.6.2. Добавляем картинку в проект

В файле ресурсов перечислены файлы, которые нужно поместить внутрь приложения. Также в нём описывается виртуальная иерархия этих файлов, к которой обращается программа.

Чтобы добавить изображение:

- в редакторе ресурсов выберите команду **Добавить** → **Добавить приставку**



- в поле **Префикс** введите название виртуального каталога для значков кнопок, например `buttons`;
- выберите команду **Добавить** → **Добавить файлы** (  ) – появится окно открытия файлов;
- выберите файл с рисунком в каталоге проекта. В итоге должно получиться нечто похожее на рисунок 3.6.4.

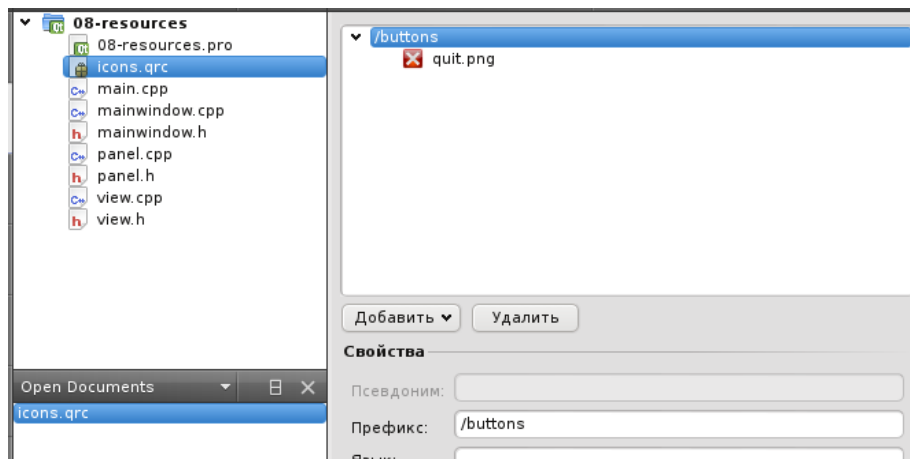


Рис. 3.6.4. Структура ресурсов проекта

### 3.6.3. Загружаем картинку из ресурсов

Чтобы в приложении использовать файл из ресурсов, нужно к его полному (с путём) имени в виртуальной иерархии ресурсов прибавить спереди двоеточие. В нашем случае (отрывок из **mainwindow.cpp**):

```
// ...
QAction* quitAct(new QAction(
    QIcon(":/buttons/quit.png"), "&Quit", this));
// ...
```

Программа должна работать, как раньше, но теперь носить файл с изображением вместе с ней не нужно. В ресурсы можно помещать любые файлы.

## 4. Графическая библиотека OpenGL

### 4.1. Готовим машину к работе

В курсе компьютерной графики рассматриваются OpenGL 2.1 и язык GLSL 1.20. Эта версия библиотеки, во-первых, включает в себя все основные средства программирования современного графического оборудования и, во-вторых, доступна на многих современных машинах.

Программа из следующего раздела (4.2) выводит версию OpenGL и список расширений. Если версия библиотеки ниже 2.1, то есть два выхода

- обновить драйвер видеокарты (мы рекомендуем сделать это в любом случае). Для этого зайдите на сайт производителя и загрузите последнюю версию драйвера для своей графической карты и системы. Узнать название карты можно в окне свойств рабочего стола (в Windows XP: контекстное меню рабочего стола (щелчок правой кнопкой по рабочему столу) → **Свойства** → вкладка **Параметры**, надпись в поле **Дисплей**, относящаяся к видеокарте). Ниже приведены сайты загрузки драйверов основных производителей микросхем видеокарт:

NVIDIA: <http://www.nvidia.ru/Download/index.aspx?lang=ru>;

ATI (AMD): <http://www.amd.com/ru/Pages/AMDHomePage.aspx> (окошко **Загрузка драйверов** в правой верхней части страницы);

Intel: [http://downloadcenter.intel.com/default.aspx?lang=rus&iid=gg\\_work-RU+downloads](http://downloadcenter.intel.com/default.aspx?lang=rus&iid=gg_work-RU+downloads) (в списке **Выберите семейство продукции** выберите пункт **Графические адаптеры**, далее – в зависимости от вашей системы).

После установки нового драйвера и перезагрузки системы проверьте версию OpenGL (или присутствие нужных расширений) ещё раз;

- проверить наличие нужных расширений. В списке расширений (вывод программы из раздела 4.2) должны присутствовать названия: GL\_ARB\_vertex\_buffer\_object, GL\_ARB\_shader\_objects, GL\_ARB\_vertex\_shader, GL\_ARB\_fragment\_shader и GL\_ARB\_shading\_language\_100.

## 4.2. Простой пример. Основы работы с OpenGL в Qt

Основные технологии:

- использование OpenGL в Qt, модуль QtOpenGL;
- получение основных сведений о реализации OpenGL.

### 4.2.1. Создаём проект

Чтобы создать в Qt Creator проект для приложения, работающего с OpenGL:

- нажмите **Control+N** (**Файл → Новый**), появится окно **Новый**. Выберите в нём пункт **GUI приложение Qt4** в разделе **Проекты** и нажмите кнопку **ОК** (рис. 4.2.1);
- появится окно **Введение и размещение проекта**. В поле **Название** введите название проекта, а в поле **Создать в** – каталог, в котором будет храниться проект. Нажмите кнопку **Вперед** (рис. 4.2.2);

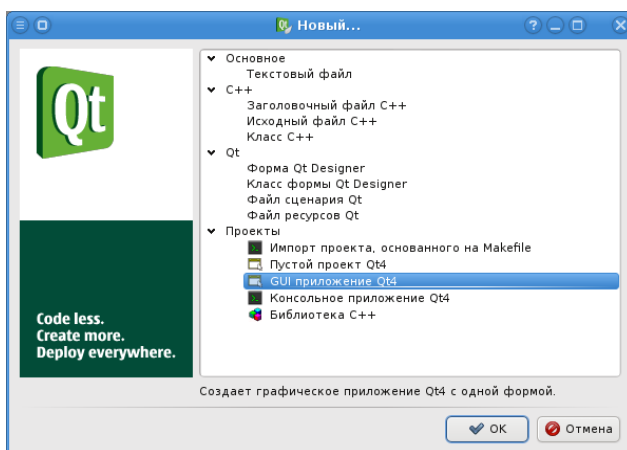


Рис. 4.2.1. Выбор вида проекта

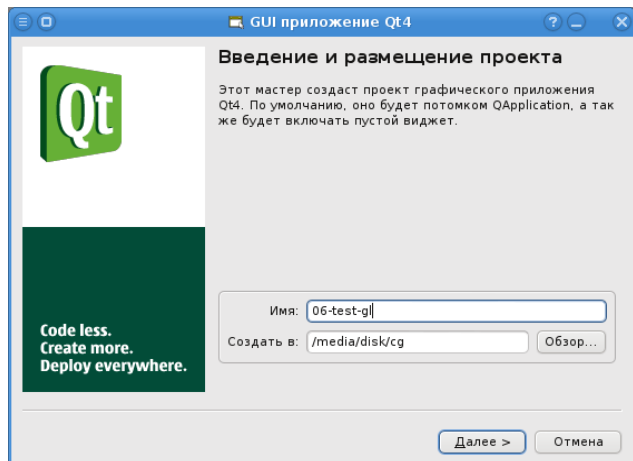


Рис. 4.2.2. Выбор расположения проекта

- в следующем окне (**Выбор необходимых модулей**) включите флажок **Модуль QtOpenGL** и нажмите **Вперед** (рис. 4.2.3);
- в окне **Информация о классе** в списке **Базовый класс** выберите **QWidget** и снимите галочку **Создать форму**. Нажмите **Вперед** (рис. 4.2.4);
- в последнем окне мастера нажмите кнопку **Финиш**, и Qt Creator создаст проект нового приложения (рис. 4.2.5).



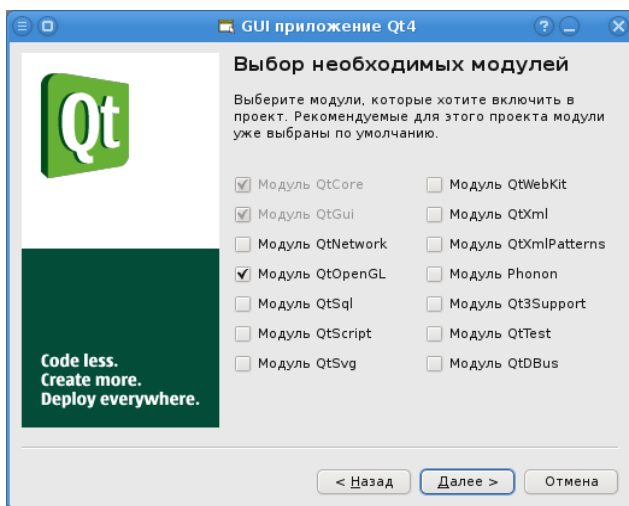


Рис. 4.2.3. Подключение QtOpenGL

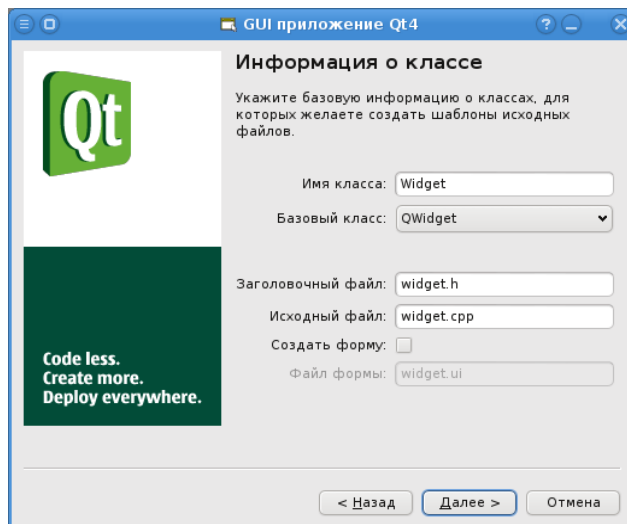


Рис. 4.2.4. Создание класса окна программы

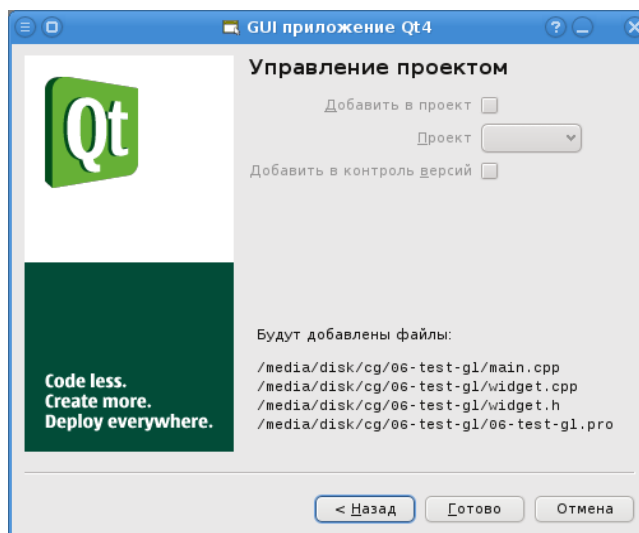


Рис. 4.2.5. Завершение создания проекта

## 4.2.2. Пишем программу

В модуле QtOpenGL есть класс окна (QGLWidget), в которое можно рисовать с помощью OpenGL. Базовым классом для QGLWidget, как и для всех окон Qt, является QWidget, поэтому с ним можно работать, как и с любым другим окном. Именно он будет базовым классом нашего окна. Некоторые системные события (например, перерисовка) в нём обрабатываются иначе, чем в обычном QWidget, поэтому для них введены свои функции и переопределять нужно именно их.

Приведём фрагмент файла **widget.h**:

```
#include <QGLWidget>

class Widget : public QGLWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

protected:
    void initializeGL();
    void paintGL();
};
```

Класс Widget переопределяет функции initializeGL и paintGL. Именно в функции paintGL (не paintEvent) необходимо рисовать при использовании класса QGLWidget. Функция initializeGL будет вызвана один раз до первого рисования, в ней можно провести начальную настройку, связанную с OpenGL. Сделать это в конструкторе не получится, потому что QGLWidget связывается с графической библиотекой позже.

Приведём реализацию основных функций (из файла **widget.cpp**):

```
Widget::Widget(QWidget *parent)
    : QGLWidget(parent)
{
}
```

Ещё раз обратим внимание на то, что мы изменили базовый класс на QGLWidget, поэтому вызывается именно его конструктор.

```
void Widget::initializeGL()
{
    using std::cout;
    cout << "version: ";
    cout << glGetString(GL_VERSION) << '\n';
    cout << "extensions:\n";
    QString ext(reinterpret_cast<const char*>(
        glGetString(GL_EXTENSIONS)));
    ext.replace(' ', '\n');
    cout << ext.toAscii().data() << '\n';
    glClearColor(0, 1.0, 0, 1.0);
}
```

Мы выдаём сведения об OpenGL на стандартный вывод программы (объект `std::cout`), сначала версию библиотеки, потом список расширений. С помощью функции `glGetString` можно получить эти и некоторые другие сведения, её параметр указывает, какую строку вернуть. Другие его значения:

- `GL_VENDOR` – сведения о производителе драйвера;
- `GL_RENDERER` – сведения о видеокарте (драйвере);
- `GL_SHADING_LANGUAGE_VERSION` – версия языка GLSL (языка раскраски OpenGL). Чтобы использовать эту константу, нужно включить в программу файл **glext.h** из раздела 4.3.

Список расширений выдаётся одной строкой, названия разделены пробелами. Чтобы читать его было удобнее, мы перед выводом заменяем пробелы переходами на новую строку с помощью `QString::replace`.

Функция `glClearColor` устанавливает цвет, которым GL будет заполнять видеобуфер (в нашем случае – окно) при его очистке (его можно считать цветом фона). Цвет задаётся четырёхмерным вектором  $(R, G, B, A)$ , каждый элемент которого лежит в интервале  $[0; 1]$ . Цвет по умолчанию –  $(0, 0, 0, 0)$ .

```
void Widget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT);
}
```

Функция `glClear` очищает выбранные буферы. В данном случае очищается только буфер цвета, то есть окно заполняется цветом очистки.

Чтобы использовать `std::cout`, не забудьте включить файл **iostream**:

```
#include <iostream>
```

Начальный размер окна можно задать в функции `main` или конструкторе `Widget` (см. раздел 3.3.4).

Чтобы в Windows можно было посмотреть текстовый вывод программы, откройте файл проекта на редактирование и в конце добавьте строку

```
CONFIG += console
```

Окно программы показано на рисунке 4.2.6. В Windows откроется отдельное консольное окно с текстовым выводом (рис. 4.2.7), в Linux текстовый вывод программы можно посмотреть после её завершения в окне **Вывод приложения** внизу окна среды Qt Creator (рис. 4.2.8).

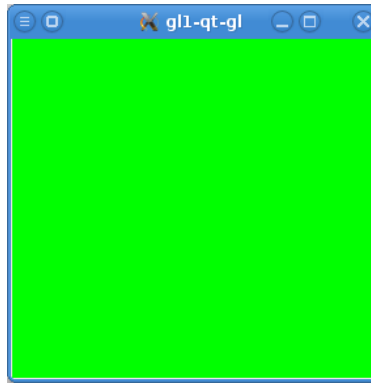


Рис. 4.2.6. Окно, закрасненное OpenGL

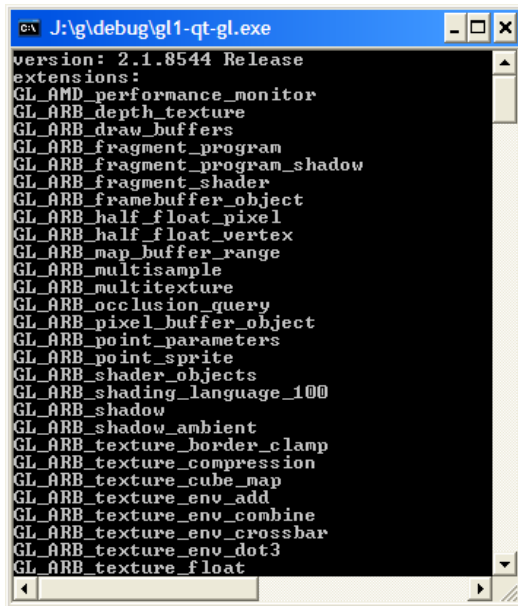


Рис. 4.2.7. Сведения об OpenGL на карте ATI Radeon 9600 (Windows)

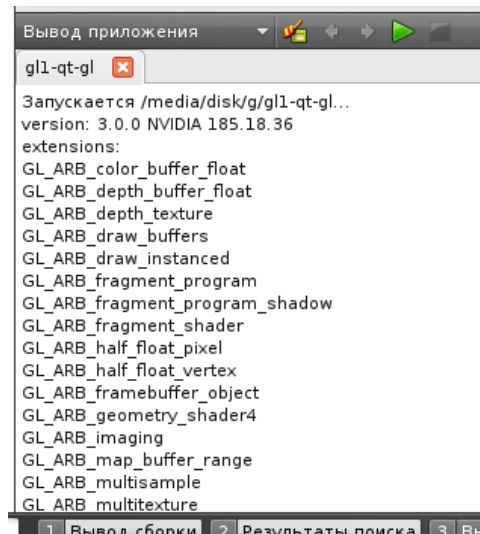


Рис. 4.2.8. Сведения об OpenGL на карте NVIDIA GeForce 8600 GT (Linux)

## 4.3. Основы рисования в OpenGL

В данном разделе мы рассмотрим, как нарисовать с помощью OpenGL простую геометрическую фигуру. Проект Qt в данном примере аналогичен проекту из раздела 4.2.1, более того, этот пример можно выполнить на основе предыдущего.

Основные технологии:

- загрузка исходных данных для рисования в видеопамять (вершинный буфер);
- загрузка и выполнение программ раскраски (шейдеров);
- язык раскраски (GLSL).

### 4.3.1. Проверяем возможности OpenGL

В данном курсе рассматривается OpenGL 2.1, поэтому программа должна проверять наличие необходимых возможностей и получать адреса новых функций, которые есть в драйвере видеокарты, но отсутствуют в реализации OpenGL корпорации Microsoft.

К данному учебнику прилагаются два исходных файла (**gl-labs.h** и **gl-labs.cpp**), которые решают описанную проблему. Их нужно положить в каталог с файлами проекта и добавить в проект (как это сделать описано в разделе 3.5.2). Также вместе с ними идёт файл **glext.h**,

который содержит описание нужных функций. Его можно просто положить в каталог проекта.

В файле **gl-labs.h** описана функция `checkCapabilities`, которая проделывает всю необходимую работу. Ей нужно передать указатель на `QGLWidget` (наше окно), а она возвратит `true` или `false` (в зависимости от успешности проверок). Использовать её можно, например, так (файл **widget.cpp**):

```
//...
#include "gl-labs.h"
#include <cstdlib>
#include <QMessageBox>

//...

void Widget::initializeGL()
{
    if (checkCapabilities(this))
    {
        // здесь можно работать с GL
    }
    else
    {
        QMessageBox::critical(this, "Error",
            "OpenGL features needed are unsupported.");
        exit(0);
    }
}
```

Программа проверяет наличие в драйвере необходимых возможностей. Если их нет, она выдаёт сообщение об ошибке и завершает свою работу (для этого в данном случае используется функция `exit` из стандартной библиотеки Си++, поскольку здесь `QApplication::quit` не работает).

### 4.3.2. Создаём программы раскраски

OpenGL берёт часть работы по рисованию сцены на себя. В числе прочего, она хранит геометрические данные, передаёт их на видеокарту, собирает из вершин линии и треугольники, растеризует их (разбивает на пиксели) и выводит пиксели в видеобuffer. Часть работы должен проделать программист. Программист должен преобразовать вершины объектов в пространстве, вычислить их цвет (например, с учётом освещения), а также (при необходимости) преобразовать пиксели до их рисования.

Вершины преобразуют с помощью *вершинных программ раскраски*, пиксели – с помощью *фрагментных*. Для этих программ существует специальный язык – GLSL (OpenGL Shading Language).

Приведём вершинную программу для данного проекта (она хранится в отдельном файле **simple.vert**):

```
#version 120

attribute vec3 pos;
attribute vec3 color;

void main()
{
    gl_Position = vec4(pos, 1.0);
    gl_FrontColor = vec4(color, 1.0);
}
```

Язык раскраски очень напоминает Си++. Глобальные переменные `pos` и `color` имеют встроенный тип `vec3` (трёхмерный вектор). Свойство `attribute` означает, что это – входные параметры программы. Программа обрабатывает одну вершину за раз. *Атрибутами* вершины называются её данные, в нашем примере у каждой вершины два атрибута: положение в пространстве и цвет.

Функция `main` вызывается первой. В данном примере она лишь передаёт данные дальше. В выходную переменную `gl_Position` программа должна записать преобразованное положение вершины (четырёхмерный вектор) в однородных координатах OpenGL. В этой системе координат ось `X` идёт слева направо (относительно плоскости экрана, рис. 4.3.1), ось `Y` – снизу вверх, ось `Z` – из экрана на наблюдателя, центр системы совпадает с центром окна. Величина каждой координаты ограничена интервалом  $[-1; 1]$ .

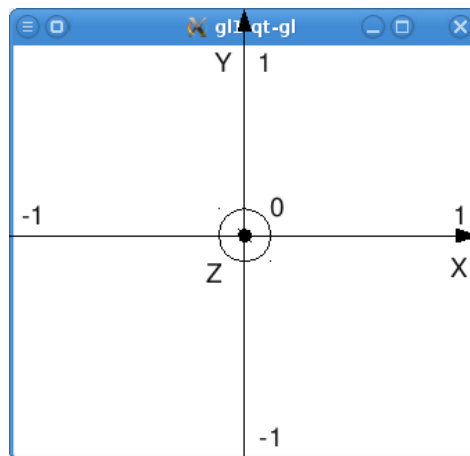


Рис. 4.3.1. Система координат OpenGL. Ось `Z` направлена на наблюдателя

В выходную переменную `gl_FrontColor` программа записывает окончательный цвет вершины для *лицевой* стороны грани. Лицевая сторона определяется порядком обхода вершин (порядком, в котором программист передаёт вершины треугольника библиотеке). По умолчанию лицевая сторона обходится против часовой стрелки. Цвет вершины для обратной стороны можно записать в выходную переменную `gl_BackColor`.

В примере также показано, как создать четырёхмерный вектор из трёхмерного вектора и скаляра.

Директива `#version` в начале программы указывает версию GLSL (1.20).

Фрагментная программа тоже очень проста (файл **simple.frag**):

```
#version 120

void main()
{
    gl_FragColor = gl_Color;
}
```

Переменная `gl_Color` является для фрагментной программы входной и имеет тип `vec4` (четырёхмерный вектор). Она хранит цвет *фрагмента* (пиксела), рассчитанный методом интерполяции во время растеризации объекта. Фрагментная программа может его преобразовать и должна записать результат в выходную переменную `gl_FragColor`. Как и вершинная, фрагментная программа обрабатывает по одному фрагменту за раз.

Программы раскраски выполняются на графическом процессоре видеокарты и не занимают центральный процессор. Мы передаём их исходный текст драйверу видеокарты, который компилирует его и загружает в видеопамять. Для этого в классе `Widget` мы создадим отдельную функцию:

```
void Widget::setupShaders()
{
    vshId = glCreateShader(GL_VERTEX_SHADER);

    QFile f("../simple.vert");
    f.open(QFile::ReadOnly | QFile::Text);
    QByteArray buf(f.readAll());
    f.close();

    const char* src(buf.data());
    glShaderSource(vshId, 1, &src, 0);

    glCompileShader(vshId);

    fshId = glCreateShader(GL_FRAGMENT_SHADER);

    f.setFileName("../simple.frag");
    f.open(QFile::ReadOnly | QFile::Text);
    buf = f.readAll();

    src = buf.data();
    glShaderSource(fshId, 1, &src, 0);

    glCompileShader(fshId);

    progId = glCreateProgram();

    glAttachShader(progId, vshId);
    glAttachShader(progId, fshId);

    glBindAttribLocation(progId, 0, "pos");
    glBindAttribLocation(progId, 1, "color");
}
```

```

        glLinkProgram(progId);

        glUseProgram(progId);
    }

```

Функция `glCreateShader` создаёт специальный объект для одной части графической программы и возвращает его идентификатор (целое число). Текст программы мы загружаем с помощью Qt: `QFile` читает текст и возвращает его в виде массива байтов (`QByteArray`). Исполняемый файл лежит в каталоге **debug**, а графические программы – вместе с другими исходными файлами (выше по иерархии каталогов), поэтому пути к файлам выглядят несколько необычно. Далее мы загружаем его в созданный объект OpenGL с помощью функции `glShaderSource`, которой нужно передать указатель на массив символьных строк. Мы храним весь текст программы одной строкой. В последнем параметре передаётся указатель на массив длин строк; если передать ноль, библиотека посчитает длины сама. Функция `glCompileShader` компилирует программу.

Все части графической программы нужно собрать в один объект. Такой объект создаёт функция `glCreateProgram`. Как и `glCreateShader`, она возвращает уникальный числовой идентификатор. Функция `glAttachShader` добавляет модуль в единую программу. Функция `glLinkProgram` компоует программу, а `glUseProgram` делает её текущей.

В вершинной программе атрибуты вершин являются переменными и у них есть имена. В нашем примере это `pos` и `color`. В основной же программе (на Си++) мы будем обращаться к ним по номерам. Номера атрибутов задаются программистом, но обычно в атрибут 0 помещают положение вершины (в OpenGL 2.1 это правило обязательно, в более поздних версиях – нет). Функция `glBindAttribLocation` связывает номер атрибута и его имя, сделать это нужно до компоновки программы.

Переменные `vshId`, `fshId` и `progId` объявлены внутри класса `Widget` (раздел 4.3.6).

### 4.3.3. Загружаем данные в видеопамять

*Вершинный буфер* – это область видеопамати, куда можно загрузить данные о вершинах объекта и откуда потом их будет читать OpenGL.

Мы будем рисовать цветной треугольник. Каждой вершине мы дадим свой цвет, а библиотека сама рассчитает переходы цвета внутри фигуры.

Для удобства заведём в классе `Widget` отдельную функцию, которая готовит данные для рисования:

```

void Widget::setupData()
{
    glGenBuffers(1, &bufId);
    glBindBuffer(GL_ARRAY_BUFFER, bufId);

    const GLfloat triang[] =
    {
        // координаты
        -0.5f, -0.5f, 0,
        0.5f, -0.5f, 0,
        0, 0.5f, 0,

```



```

    // цвета
    0, 1.0f, 0,
    0, 0, 1.0f,
    1.0f, 0, 0
};
glBufferData(GL_ARRAY_BUFFER, 18 * sizeof(GLfloat),
             triang, GL_STATIC_DRAW);

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
                      reinterpret_cast<GLvoid*>(9 * sizeof(GLfloat)));
}

```

Функция `glGenBuffers` создаёт  $n$  буферов (в данном случае – один). Ей надо передать указатель на массив целых чисел типа `GLuint`, в который она запишет  $n$  идентификаторов. У каждого буфера есть число-идентификатор, по которому к нему можно обратиться. Мы создаём один буфер, поэтому передаём указатель на обычную переменную `bufId`, которая объявлена внутри класса `Widget` (раздел 4.3.6).

Чтобы работать с буфером, его надо сделать *текущим* (активным). Есть несколько видов буферов, которые различаются тем, какие данные они хранят. Например, буфер может хранить сами данные вершин (`GL_ARRAY_BUFFER`) или номера (индексы) вершин некоторого объекта, данные которого лежат в другом буфере (`GL_ELEMENT_ARRAY_BUFFER`). Для каждого такого вида данных текущим может быть не более одного буфера. Функция `glBindBuffer` делает буфер текущим. Первый параметр задает вид данных, второй – идентификатор буфера.

Функция `glBufferData` загружает данные в буфер указанного типа. Мы передаём ей объём данных в байтах и указатель на массив с данными. Параметр `GL_STATIC_DRAW` означает, что мы будем использовать буфер для рисования и не будем менять данные в нём. Функция создаёт буфер указанного размера и копирует туда данные из массива.

Функция `glEnableVertexAttribArray` разрешает передачу атрибута с указанным номером вершинной программе. С помощью `glVertexAttribPointer` мы описываем, как данные (значения атрибутов для каждой вершины объекта) хранятся в буфере. Первый параметр – номер атрибута. Второй – его размерность (от одного до четырёх). Третий – тип чисел. Четвёртый параметр управляет нормализацией целых чисел (у нас числа вещественные). Пятый необходим, когда разные атрибуты лежат в буфере в перемешку. Шестой параметр – начало данных для указанного атрибута в байтах. По историческим причинам тип шестого параметра – «указатель на `void`», поэтому нам приходится преобразовать число явно.

#### 4.3.4. Начальная настройка и рисование

Вся основная работа уже проделана. В функции `Widget::initializeGL` нам остаётся вызвать функции из предыдущих разделов, а в `Widget::paintGL` – нарисовать треугольник. Приведём тексты обеих функций:

```

void Widget::initializeGL()
{
    if (checkCapabilities(this))
    {
        setData();
        setupShaders();
    }
    else
    {
        QMessageBox::critical(this, "Error",
            "OpenGL features needed are unsupported.");
        exit(0);
    }
}

void Widget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, 3);
}

```

Функция `glDrawArrays` передаёт вершины из текущего вершинного буфера на графический вывод. Мы просим её нарисовать треугольники. Ноль – номер первой вершины, три – их количество. Номера вершин определяются порядком, в котором они лежат в буфере.

Также при изменении размеров окна необходимо передать OpenGL его новые размеры. В `QWidget` есть функция `resizeEvent`. Для обычного окна необходимо перегрузить её, чтобы получать сообщения о том, что размер окна изменился. В случае `QGLWidget` опять же есть специальная функция `resizeGL`, и перегружать нам надо именно её:

```

void Widget::resizeGL(int width, int height)
{
    glViewport(0, 0, width, height);
}

```

Функция `glViewport` задаёт границы «области просмотра». На самом деле никакого отсечения по данным границам не происходит. `glViewport` лишь задаёт геометрическое преобразование трёхмерных координат пиксела в оконные. Мы передаём координаты левого нижнего угла области, её ширину и высоту, все значения измеряются в пикселах.

### 4.3.5. Освобождение захваченных ресурсов

В классе `QWidget` есть событие `closeEvent`, которое вызывается при закрытии окна. Мы перегружаем эту функцию, чтобы освободить ресурсы OpenGL:

```

void Widget::closeEvent(QCloseEvent*)
{
    glUseProgram(0);
    glDeleteProgram(progId);
    glDeleteShader(vshId);
    glDeleteShader(fshId);
    glDeleteBuffers(1, &bufId);
}

```

Сначала мы отключаем программу раскраски, затем удаляем все созданные объекты.

### 4.3.6. Класс Widget

Наконец приведём определение класса `Widget` для данного примера:

```
class Widget : public QGLWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

protected:
    void initializeGL();
    void paintGL();
    void closeEvent(QCloseEvent*);
    void resizeGL(int width, int height);

private:
    void setupData();
    void setupShaders();

    GLuint bufId;
    GLuint vshId, fshId, progId;
};
```

### 4.3.7. Запускаем программу

На рисунке 4.3.2 показано окно работающей программы.

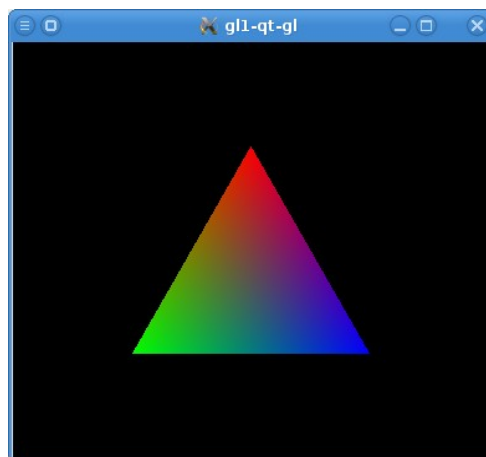


Рис. 4.3.2. Снимок окна программы

## 4.4. Более сложный пример

В данном примере показано, как с помощью OpenGL нарисовать трёхмерный объект из нескольких граней (куб), создать простую анимацию (вращение) и рассчитать цвет вершин объекта по некоторой математической модели.

Данный пример основан на предыдущем (из раздела 4.3).

Основные технологии:

- описание граней объекта через номера вершин (буферы индексов);
- передача вершинной программе произвольных параметров, не связанных с вершинами;
- использование таймера в Qt;
- язык раскраски (GLSL);
- отсечение невидимых граней с помощью буфера глубины (z-буфера).

### 4.4.1. Пишем программы раскраски

На этот раз вершинная программа будет немного сложнее. Она должна преобразовать вершину (повернуть её вокруг оси Y на заданный угол) и рассчитать её цвет. Математическая модель этого расчёта будет проста: в пространстве поместим «источник света» с заданным цветом, цвет вершины будет прямо пропорционален цвету источника и обратно пропорционален расстоянию от вершины до источника:

$$C_v = \frac{C_s}{d}.$$

Приведём текст вершинной программы, а потом его опишем:

```
#version 120

attribute vec3 pos;

uniform float angle = 0.0;

uniform vec3 lightColor = vec3(1.0, 1.0, 1.0);
uniform vec3 lightPos = vec3(0.0, 0.0, -1.7);

void main()
{
    float c = cos(angle);
    float s = sin(angle);

    gl_Position =
        vec4(dot(pos, vec3(c, 0.0, s)),
            pos.y,
            dot(pos, vec3(-s, 0.0, c)),
            1.0);

    float d = distance(gl_Position.xyz, lightPos);
    gl_FrontColor = vec4(lightColor, 1.0) / d;
}
```

У вершины теперь только один атрибут – её положение. Глобальные переменные со свойством `uniform` тоже являются входными параметрами программы, но они не зависят от вершин и не меняются в пределах одного геометрического примитива (например, треугольника). Значение `uniform`-переменных можно менять из основной программы, при создании они получают значения по умолчанию, заданные в вершинной программе.

Сначала вершина поворачивается вокруг оси `Y` на угол `angle`. Функция `dot` вычисляет скалярное произведение двух векторов одинаковой размерности.

Затем мы вычисляем цвет вершины. Функция `distance` вычисляет расстояние между двумя векторами ( $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$ ). Заметьте, что при вычислении расстояния мы используем преобразованное значение положения (переменную `gl_Position`, а не `pos`). Из компонентов вектора в GLSL можно как угодно составлять новый вектор. В данном случае взяты компоненты `x`, `y` и `z` в прямом порядке. `gl_Position` – четырёхмерный вектор, а `lightPos` – трёхмерный, поэтому преобразование необходимо.

На GLSL можно записывать все основные действия над векторами и матрицами (например, при расчёте цвета мы делим вектор на скаляр). К компонентам вектора можно обращаться по именам: либо `x`, `y`, `z`, `w`, либо `r`, `g`, `b`, `a`. Также к ним можно обращаться по индексам, как к элементам массива. Смешивать разные способы обращения к компонентам нельзя. Примеры:

```
// Правильно:
v.xу // новый двумерный вектор
v.zyx // новый трёхмерный вектор в обратном порядке
v[2] // обращение к одному ...
v.y // ... и тому же ...
v.g // ... компоненту

// Неправильно:
v.ryb // мешать компоненты нельзя
```

Фрагментная программа в данном примере мало отличается от предыдущей (раздел 4.3.2):

```
#version 120

void main()
{
    gl_FragColor = gl_Color;
    gl_FragDepth = gl_FragCoord.z;
}
```

Теперь, кроме цвета фрагмента (будущего пиксела), мы передаём дальше и его глубину (`z`-координату, преобразованную в систему координат окна) для проверки с помощью буфера глубины (см. ниже). `gl_FragDepth` – выходная переменная для значения глубины пиксела, `gl_FragCoord` – его входные координаты.

Функция `Widget::setupShaders` изменилась не сильно:

```
void Widget::setupShaders()
{
    vshId = glCreateShader(GL_VERTEX_SHADER);

    QFile f("../complex.vert");
    f.open(QFile::ReadOnly | QFile::Text);
    QByteArray buf(f.readAll());
    f.close();

    const char* src(buf.data());
    glShaderSource(vshId, 1, &src, 0);

    glCompileShader(vshId);

    fshId = glCreateShader(GL_FRAGMENT_SHADER);

    f.setFileName("../complex.frag");
    f.open(QFile::ReadOnly | QFile::Text);
    buf = f.readAll();

    src = buf.data();
    glShaderSource(fshId, 1, &src, 0);

    glCompileShader(fshId);

    progId = glCreateProgram();

    glAttachShader(progId, vshId);
    glAttachShader(progId, fshId);

    glBindAttribLocation(progId, 0, "pos");

    glLinkProgram(progId);

    angleAddr = glGetUniformLocation(progId, "angle");
    posAddr = glGetUniformLocation(progId, "lightPos");
    colorAddr = glGetUniformLocation(progId, "lightColor");

    glUseProgram(progId);
}
```

Функция `glGetUniformLocation` возвращает числовой идентификатор `uniform`-переменной (после сборки всей программы раскраски). Позже мы сможем с помощью этих идентификаторов менять значения соответствующих переменных.

## 4.4.2. Загружаем данные

Приведём текст функции `Widget::setupData`:

```
void Widget::setupData()
{
    glGenBuffers(2, bufId);
    glBindBuffer(GL_ARRAY_BUFFER, bufId[0]);

    const GLfloat cube[] =
    {
        // координаты вершин
        -0.5f, -0.5f, 0.5f,
        0.5f, -0.5f, 0.5f,
        0.5f, 0.5f, 0.5f,
        -0.5f, 0.5f, 0.5f,
        -0.5f, -0.5f, -0.5f,
        -0.5f, 0.5f, -0.5f,
        0.5f, 0.5f, -0.5f,
        0.5f, -0.5f, -0.5f
    };
    glBufferData(GL_ARRAY_BUFFER, 24 * sizeof(GLfloat),
                 cube, GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, bufId[1]);
    const GLuint cubeFaces[] =
    {
        // лицевая грань
        0, 1, 2,
        0, 2, 3,
        // правая грань
        1, 6, 2,
        1, 7, 6,
        // задняя грань
        4, 6, 7,
        4, 5, 6,
        // левая грань
        0, 3, 4,
        4, 3, 5,
        // верхняя грань
        3, 2, 6,
        3, 6, 5,
        // нижняя грань
        0, 4, 7,
        0, 7, 1
    };
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, 36 * sizeof(GLuint),
                 cubeFaces, GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
}
```

Обратите внимание на то, что в этот раз мы создаём два буфера сразу. Теперь `bufId` – массив из двух идентификаторов.

В первый буфер мы заносим восемь вершин куба. Вторым же назначается как текущий `GL_ELEMENT_ARRAY_BUFFER`, то есть *буфер индексов*. В него мы помещаем описания всех треугольников куба. Числа в нём – номера вершин (по порядку) в буфере вершин (рис. 4.4.1).

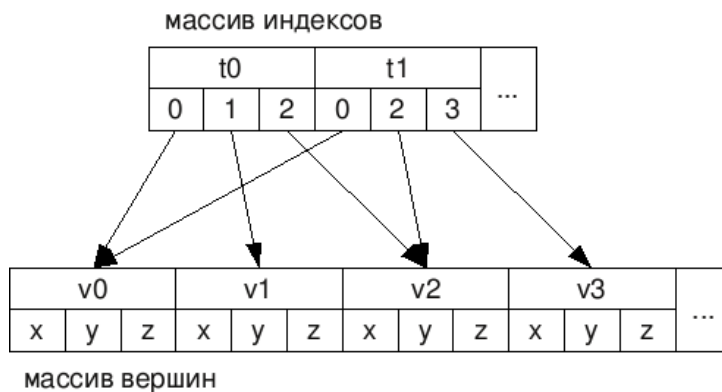


Рис. 4.4.1. Буфер вершин и буфер индексов

## 4.4.3. Обработываем системные события

### 4.4.3.1. Начальная настройка

Текст функции `Widget::initializeGL`:

```
void Widget::initializeGL()
{
    if (checkCapabilities(this))
    {
        setupData();
        setupShaders();

        glClearDepth(100);
        glEnable(GL_DEPTH_TEST);

        angle = 0;
        timer = startTimer(30);
    }
    else
    {
        QMessageBox::critical(this, "Error",
            "OpenGL features needed are unsupported.");
        exit(0);
    }
}
```

Здесь есть два нововведения. Во-первых, мы используем *буфер глубины* (z-буфер). Функция `glClearDepth` задаёт число для его очистки (начального заполнения). Функция `glEnable` с параметром `GL_DEPTH_TEST` включает проверку z-координат фрагментов (пикселей).



Во-вторых, мы задаём начальный угол для вращения и запускаем таймер. Функция `QObject::startTimer` принимает длину интервала в миллисекундах и возвращает числовой идентификатор таймера. Также она включает сам таймер, теперь каждые 30 мс библиотека будет вызывать событие `timerEvent` для нашего объекта окна.

#### 4.4.3.2. Рисование

Приведём функцию `Widget::paintGL`:

```
void Widget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
}
```

Одной командой `glClear` мы очищаем буфер цвета (экран) и буфер глубины.

Функция `glDrawElements` рисует примитивы по набору индексов (из буфера индексов). Мы передаём ей количество индексов, их тип и начало массива внутри буфера (в байтах, как и в функции `glVertexAttribPointer`).

#### 4.4.3.3. Анимация

Текст функции `Widget::timerEvent` (обработка событий от таймера):

```
void Widget::timerEvent(QTimerEvent*)
{
    angle += 2;
    if (angle >= 360)
        angle = 0;

    const float k(3.1415926535897932 / 180);
    glUniform1f(angleAddr, angle * k);
    update();
}
```

Здесь мы увеличиваем угол, переводим его в радианы и записываем в `uniform`-переменную `angle` вершинной программы. Идентификатор (адрес) переменной мы получили в функции `Widget::initializeGL` (раздел 4.4.3.1). Также мы обновляем картинку в окне.

#### 4.4.3.4. Освобождение системных ресурсов

Функция `Widget::closeEvent` теперь, кроме прочего, останавливает таймер и удаляет индексный буфер:

```
void Widget::closeEvent(QCloseEvent*)
{
    killTimer(timer);
    glUseProgram(0);
    glDeleteProgram(progId);
    glDeleteShader(vshId);
    glDeleteShader(fshId);
    glDeleteBuffers(2, bufId);
}
```

#### 4.4.4. Класс Widget

Класс Widget теперь «умеет» обрабатывать события таймера и хранит дополнительные числовые идентификаторы (переменная bufId теперь является массивом и хранит сразу два идентификатора: для вершинного и индексного буферов; это удобно, поскольку, как мы видели, пара glGenBuffers/glDeleteBuffers может работать с несколькими буферами сразу):

```
class Widget : public QGLWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

protected:
    void initializeGL();
    void paintGL();
    void resizeGL(int width, int height);
    void closeEvent(QCloseEvent*);
    void timerEvent(QTimerEvent*);

private:
    void setupData();
    void setupShaders();

    GLuint bufId[2];
    GLuint vshId, fshId, progId;
    GLint angleAddr, colorAddr, posAddr;
    int timer;
    GLfloat angle;
};
```

## 4.4.5. Запускаем программу

На рисунке 4.4.2 показаны снимки окна программы.

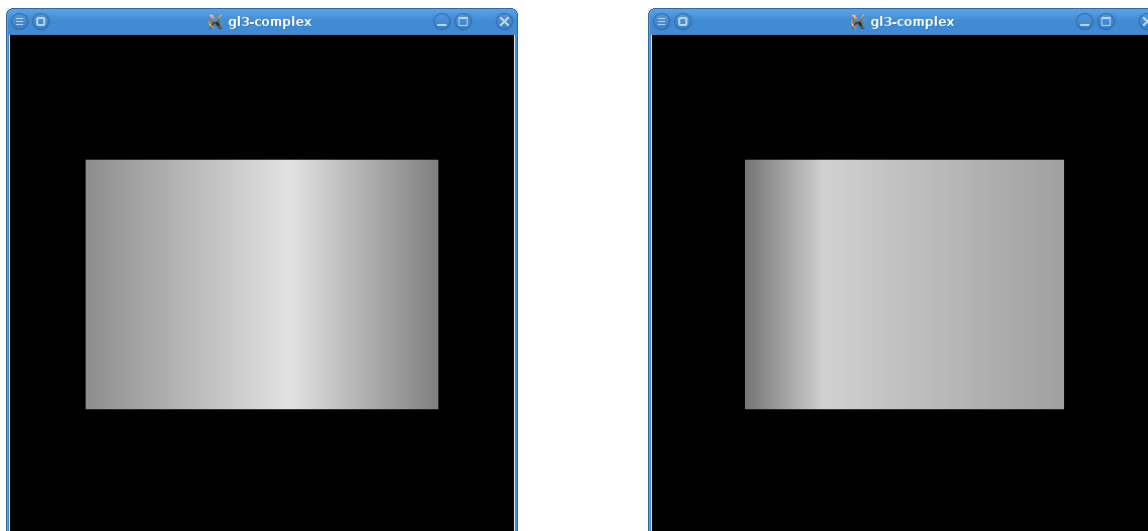


Рис. 4.4.2. Работающая программа с кубом

## 4.5. Дополнительные возможности и приёмы работы

В данном разделе мы рассмотрим ещё несколько средств OpenGL, которые могут пригодиться в курсе лабораторных работ.

### 4.5.1. Проверка ошибок

В некоторых случаях необходимо проверять, всё ли в порядке, правильно ли работает программа. В частности, в примерах выше может потребоваться проверить, получилось ли выделить память для буфера вершин или индексов и правильно ли написаны программы раскраски.

Основной способ проверки ошибок в OpenGL – функция `glGetError`. Если в какой-то функции библиотеки случается ошибка, её код запоминается в специальной внутренней переменной. Любые последующие ошибки в любых других функциях OpenGL уже не могут перезаписать этот код, то есть запоминается только первая ошибка. `glGetError` возвращает код ошибки и сбрасывает внутреннюю переменную, чтобы новая ошибка могла записать свой код. Если ошибок не было, `glGetError` возвращает код `GL_NO_ERROR`. В таблице 4.5.1 перечислены основные коды ошибок OpenGL.

Реализация библиотеки может хранить не один код ошибки, а несколько. В таком случае `glGetError` будет возвращать коды по одному, пока не вернёт все (если ошибки закончились, функция будет возвращать `GL_NO_ERROR`). Порядок хранения и выдачи кодов не определён, то есть библиотека не обязана возвращать коды ошибок в том порядке, в котором ошибки возникали.

Таблица 4.5.1

**Коды ошибок OpenGL**

<i>Код ошибки</i>	<i>Описание</i>
GL_INVALID_ENUM	Программист передал команде неправильную именованную константу
GL_INVALID_VALUE	Программист передал команде число в неверном диапазоне
GL_INVALID_OPERATION	Программист вызвал неверную команду (в данном состоянии системы)
GL_OUT_OF_MEMORY	Недостаточно памяти для выполнения команды

В качестве примера покажем, как проверить, смогла ли система создать буфер вершин. В данном случае мы делаем проверку «наверняка». То есть сначала очищаем все сохранённые коды ошибок, затем пытаемся создать буфер. Фрагмент текста программы:

```
// ...

while (glGetError() != GL_NO_ERROR)
{
}

glBufferData(GL_ARRAY_BUFFER, /* ... */);

if (glGetError() == GL_OUT_OF_MEMORY)
{
    // Памяти не хватило. Обрабатываем ошибку.
}

// ...
```

#### 4.5.1.1. Проверка ошибок в программах раскраски

В программах раскраски могут быть ошибки. Чтобы проверить это, существует набор специальных команд.

Сначала мы проверяем, правильно ли скомпилировалась программа. Если в ней есть ошибки, можно получить их описание. Оно напоминает сообщения об ошибках обычного компилятора. Пример:

```
// ...

glCompileShader(vshId);

GLint status;
glGetShaderiv(vshId, GL_COMPILE_STATUS, &status);

if (!status)
{
    // В программе раскраски есть ошибки.
    // Получим их описание
    GLint length;
    glGetShaderiv(vshId, GL_INFO_LOG_LENGTH, &length);
    if (length > 0)
    {
        // Если описание доступно, получаем его
        char* info(new char[length]);
        glGetShaderInfoLog(vshId, length, 0, info);

        // выводим описание куда-нибудь, где его можно прочесть
        // ...

        delete[] info;
    }
}

// ...
```

`glGetShaderiv` возвращает через последний параметр целое число – запрошенную характеристику программы раскраски. Запрос `GL_COMPILE_STATUS` выдаст результат компиляции (`GL_TRUE` или `GL_FALSE`), `GL_INFO_LOG_LENGTH` – длину сообщения об ошибках (считая завершающий нулевой символ).

`glGetShaderInfoLog` записывает сообщения об ошибках в переданный ей буфер (последний параметр). Результат является обычной строкой в стиле Си с нулевым символом в конце.

Если вы выводите строку из `info` в какой-нибудь из стандартных потоков (например, `cout`, `cerr` или `clog`) в Windows, не забудьте указать в файле проекта, что ваше приложение использует консоль (см. раздел 4.2.2).

Ошибки сборки программы проверяются аналогично с точностью до названий функций:

```
// ...

glLinkProgram(progId);

GLint status;
glGetProgramiv(progId, GL_LINK_STATUS, &status);

if (!status)
{
    // После сборки были ошибки.
    // Получим их описание
    GLint length;
    glGetProgramiv(progId, GL_INFO_LOG_LENGTH, &length);
    if (length > 0)
    {
        // Если описание доступно, получаем его
        char* info(new char[length]);
        glGetProgramInfoLog(vshId, length, 0, info);

        // выводим описание куда-нибудь, где его можно прочесть
        // ...

        delete[] info;
    }
}

// ...
```

## 4.5.2. Рисование «больших» объектов

Иногда требуется вывести объект с большим количеством вершин и треугольников. Такие объекты занимают много памяти и её может попросту не хватить (объём видеопамати обычно существенно меньше оперативной). Такой объект надо рисовать по частям. Если объект известен, можно учесть особенности его геометрии (например, симметрию). Иначе его приходится разбивать на произвольные порции. Мы покажем, как реализовать второй способ.

При создании буфера вершин мы не будем указывать его содержимое, передаём только размер. Размер буфера меняться не будет, его выбирают из практических соображений один раз либо статически, при написании программы, либо динамически, при её выполнении. Мы создадим буфер на 100 треугольников. Индексный буфер нам не нужен вовсе. Также мы говорим OpenGL, что содержимое буфера будет изменяться (режим GL\_DYNAMIC\_DRAW):

```
// ...

glBufferData(GL_ARRAY_BUFFER, 100 * 3 * 3 * sizeof(float),
             0, GL_DYNAMIC_DRAW);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);

// ...
```

При рисовании мы выводим данные частями по следующему алгоритму:

- отображаем вершинный буфер в оперативную память, получаем указатель на него;
- заполняем его новой порцией данных;
- разблокируем буфер;
- выводим его содержимое.

В примере ниже предполагается, что у нас есть массив координат вершин и массив треугольников, которые описаны индексами вершин. При других способах описания объекта вывод можно осуществить аналогично.

```
// ...

// Структуры данных. Предполагаем, что они хранят полное
// описание конкретного объекта

struct Vertex
{
    float x, y, z;
};
Vertex vertices[NUM_VER];

struct Triangle
{
    int v0, v1, v2;
};
Triangle faces[NUM_TR];

// ...

// Вывод

glBindBuffer(GL_ARRAY_BUFFER, bufId);

for (int face(0); face < NUM_TR; )
{
    Vertex* buffer = reinterpret_cast<Vertex*>(
        glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY));

    int tr(0), ver(0);

    for (; tr < 100 && face < NUM_TR;
        ver += 3, ++face, ++tr)
    {
        buffer[ver] = vertices[faces[face].v0];
        buffer[ver + 1] = vertices[faces[face].v1];
        buffer[ver + 2] = vertices[faces[face].v2];
    }

    glUnmapBuffer(GL_ARRAY_BUFFER);

    glDrawArrays(GL_TRIANGLES, 0, tr * 3);
}

// ...
```

Функция `glBindBuffer` делает буфер текущим (раздел 4.3.3). `glMapBuffer` отображает буфер в оперативную память, чтобы с ним можно было работать, как с обычным массивом. В данном примере мы отображаем буфер для записи туда данных. `glMapBuffer` возвращает указатель на тип `void`, поэтому мы приводим его к типу, с которым работаем. `glUnmapBuffer` отключает отображение, сделанное через `glMapBuffer`. `glDrawArrays` выводит последовательный блок данных из буфера вершин (раздел 4.3.4).

### 4.5.3. Рисование прозрачных объектов

В OpenGL прозрачность реализуется с помощью *смешивания* цветов. Сначала выводятся объекты, которые находятся в пространстве за прозрачным (фон). Затем выводится прозрачный объект, причём его пиксели смешиваются с пикселями фона, которые лежат под объектом, по заданному закону.

Сначала необходимо задать закон смешивания. По умолчанию включена следующая формула расчёта цвета (и она нас устраивает):

$$C = C_s S + C_d D,$$

где  $C$  – результат смешивания (новый цвет пикселя);

$C_s$  – цвет пикселя прозрачного объекта;

$C_d$  – цвет пикселя фона под ним;

$S, D$  – коэффициенты смешивания.

Цвет в OpenGL состоит из четырёх компонентов в интервале  $[0;1]$  (раздел 4.2.2). Первые три хранят собственно цвет в системе RGB. Четвёртый называется *альфа-компонентом* и хранит дополнительную информацию о цвете. Именно его обычно используют для описания прозрачности. Мы будем хранить в нём степень видимости объекта, единица будет обозначать полностью непрозрачный цвет. Тогда формула, приведённая выше, приобретает следующий вид:

$$C = C_s A_s + C_d (1 - A_s),$$

где  $A_s$  – альфа-компонент пикселя выводимого (прозрачного) объекта.

Функция `glBlendFunc` задаёт вид коэффициентов  $S$  и  $D$  для уравнения смешивания, вызвать её можно один раз при начальной настройке программы. В нашем случае (`SRC_ALPHA` обозначает  $A_s$  из нашей формулы):

```
// ...  
  
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);  
  
// ...
```

Перед рисованием прозрачного объекта надо включить смешивание. Также надо отключить запись в буфер глубины (но не проверку глубины), потому что у прозрачного объекта должны быть нарисованы все пиксели. После вывода прозрачного объекта мы возвращаем обе настройки в исходное состояние:



```
// ...

glEnable(GL_BLEND);
glDepthMask(GL_FALSE);

// рисуем прозрачный объект
// ...

glDisable(GL_BLEND);
glDepthMask(GL_TRUE);

// ...
```

Программы раскраски должны правильно рассчитать цвета вершин и пикселей, включая их альфа-компоненты.