Interactive Graphics Applications with OpenGL Shading Language and Qt

João Paulo Gois & Harlen C. Batagelo
Centro de Matemática Computação e Cognição
Universidade Federal do ABC
Santo André, Brazil
http://professor.ufabc.edu.br/~{joao.gois,harlen.batagelo}

Abstract—**Qt framework allows the easy development of professional cross-platform graphics applications using C++. Qt provides the *QtOpenGL Module* that makes easy the development of hardware-accelerated graphics applications using OpenGL and OpenGL Shading Language (GLSL). With Qt, matrices, vectors, vertex buffer objects, textures, shader programs and UI components are integrated by classes in the object-oriented paradigm and intercommunicate by the Qt mechanism of *signals/slots*. The goal of this survey is to detail the development of interactive graphics applications with OpenGL and Qt. Along with it, we compare features of QtOpenGL Module with those of GLU/GLUT libraries, as the latter is traditionally used in text books and computer graphics courses.**

I. INTRODUCTION

OpenGL [1] is the most popular cross-platform industry standard API for writing 2D and 3D interactive graphics applications such as for CAD, video games, scientific visualization, information visualization and virtual reality. Traditionally, course syllabus and textbooks of computer graphics are approached with OpenGL and its cross-platform window management library GLUT [2].

GLUT provides resources to control windows associated to OpenGL contexts and to perform I/O communication (mouse and keyboard devices) with the operating system. Although GLUT is broadly popular for educational purposes, it is quite limited to produce full-featured applications because its set of features is relatively restricted.

In this sense, the Qt framework [3], [4], [5] is an alternative for developing professional interactive graphics applications based on OpenGL and its shading language GLSL. Qt is open-source, cross-platform and available under different licenses, including GNU LGPL 2.1, GNU GPL 3.0 and a commercial developer license. Prominent corporations have employed Qt in their projects, *e.g.*, Autodesk, Adobe, Skype (now part of Microsoft), Wolfram, DreamWorks, Google, Lucasfilm, Samsung, Siemens, Volvo and Walt Disney Animation Studios [6].

Qt offers the *QtOpenGL Module* [7]: a full-feature set of containers to make easy the development of graphics application using OpenGL/GLSL. Qt integrates to GLSL through `QShader` and `QShaderProgram` classes that require small coding effort to compile, link and bind shader programs to applications. Matrix and vector classes such as `QMatrix2x2`, `QMatrix3x3`, `QMatrix4x4`, `QVector2D`, `QVector3D`, `QVector4D` and `QQuaternion` allow for operating with matrices, vectors and quaternions, respectively. In particular, `Qmatrix4x4` provides methods for orthogonal and perspective projections and for camera settings. The contents of these classes can be easily bound to shaders as storage qualifiers (`attribute` or `uniform`) using different shader types, *e.g.* `vec2`, `vec3`, `vec4`, `mat2`, `mat3` and `mat4`.

For texture manipulation, image files can be loaded to the application using the Qt class `QImage`. Through the Qt class `QGLWidget`, such images can be directly bound to the OpenGL texture units used in the shader programs.

The use of some recent OpenGL extensions such as *Vertex Buffer Objects*, *Frame Buffer Objects* and *Pixel Buffer Objects*, may make the source code easily cluttered and non-intuitive, specially for beginner developers. In its turn, Qt provides classes that encapsulate most details of the many extensions available for shader programming, thus producing code that is cleaner and easier to maintain.

A few requirements

There are only a few basic requirements to take the best advantages of this survey. The reader should have some familiarity with object-oriented programming, C++ preferred. Also, previous knowledge on computer graphics is required. In particular, the reader should be familiar with foundations of geometric transformations, viewing, lighting and shading [8], [9]. Since we exemplify applications with GLSL, familiarity with this shading language is desirable [10], [11].

Structure of the presentation

In the next section we guide the reader through the development of an interactive OpenGL/GLSL application with Qt. First we start up a Qt project with minimal support to OpenGL. Next we show how to visualize surface meshes loaded from a file and how to rotate and scale them with a virtual trackball. Different shader effects (Gouraud shading, Phong shading, texture mapping and normal mapping) and simple Qt UI components are also introduced. After presenting this interactive application, we then discuss how to improve its interface with richer UI components (Sec. III). Finally, we conclude our presentation (Sec. IV) and describe directions for further graphics-based Qt applications (Sec. V).

II. Creating an OpenGL/GLSL Application with Qt

A. Starting up a Qt project

The Qt libraries and tools are enclosed in the Qt SDK, available in the Qt website [3] for Windows, Linux and Mac OS X. In this work we develop our application under Qt Framework version 4.8 using the Qt Creator IDE version 2.4.1 (Figure 1) for Linux/Ubuntu. In the following we will detail this process. First we will create a Qt project to our application. After that we will design the UI, write the C++ and GLSL codes and manage the assets of our application.
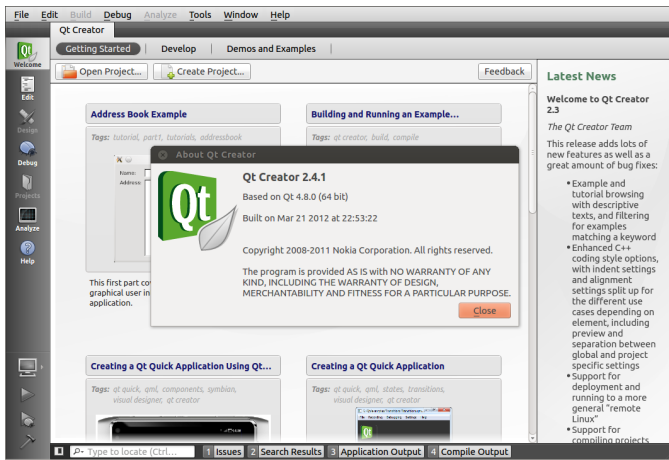

Fig. 1. Interface of the Qt Creator IDE.

Tip 1. *To maintain the coherence with the Qt style, we follow the Qt Coding Guidelines [12] in our C++ files.*

Let us start by creating a Qt project. Clicking at **File→New File or Project** on the Qt Creator menu bar, we see that Qt offers some project templates. In our case, we choose *Qt Gui Application* (Fig. 2-(a)), which creates a minimal application for desktop with a main window, a menu bar, a tool bar, a status bar and a central widget (Fig. 3-(b)).

Following steps depicted in Fig. 2, first we select the *Qt Gui Application* option (Fig. 2-(a)), second we define the project name, as well as its path (Fig. 2-(b)), third we select *Desktop* as the build target and *Use Shadow Building* to create a separate directory for storing the compiled sources, objects and executable files (Fig. 2-(c)). Finally, we specify details of the main window interface class as shown in Fig. 2-(d). Qt automatically generated four files:

- **myqtglproject.pro**: this file, depicted in Listing 1, contains the configurations of the Qt project. It lists the source files of the project, as well as the modules used. Initially, only two Qt modules are added: `core` and `gui`. In order to enable *QtOpenGL Module*, we add `opengl` in `myqtglproject.pro`, as depicted in Listing 2.
- **mainwindow.{h,cpp}**: these are the files for the `MainWindow` class (Listings 3 and 4). In the declaration of this class, we notice the Qt macro `Q_OBJECT`, which is automatically declared under the `private` access modifier.
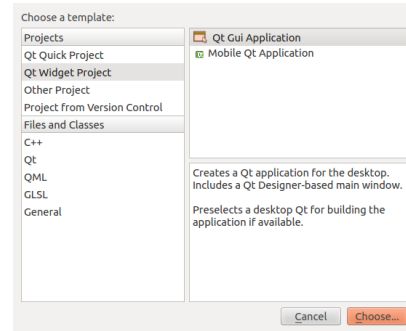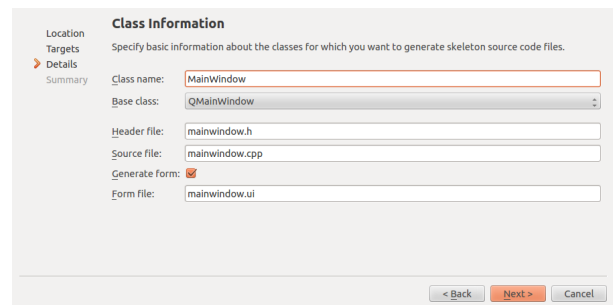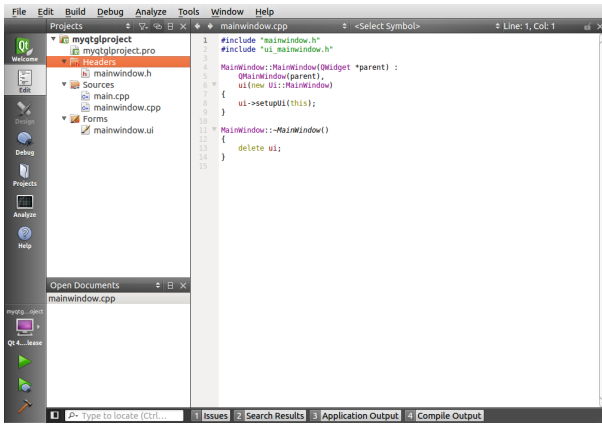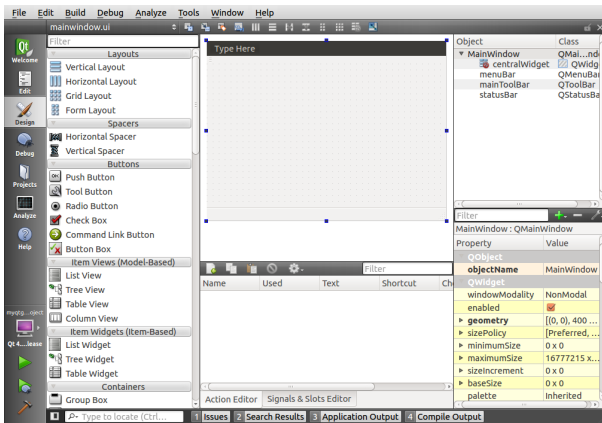

(a)
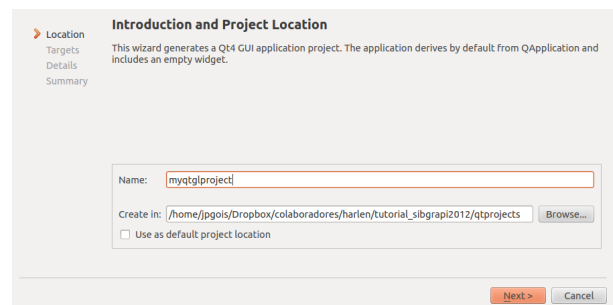
(b)
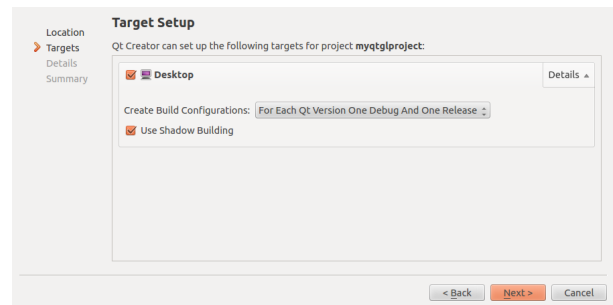
(c)

(d)

Fig. 2. Step-by-step to start up a Qt GUI Project.

(a)

(b)

Fig. 3. Qt Creator: (a) the *Edit* mode, (b) the *Design* mode.

This macro is mandatory for any class that implements the Qt concept of Signals and Slots (presented in Sec. II-D). For our application it will not be required to make changes on these files.

- **main.cpp**: this file contains the C++ main function (Listing 5). We will keep main.cpp as simple as we can. We will only insert further code on it to format details of OpenGL context, for instance, to use antialiasing. It will be detailed in Section II-N.

- **mainwindow.ui**: this file contains the XML description of the user interface. This XML file is translated by the *Qt User Interface Compiler (uic)* to a C++ header file that contains declarations of object widgets used by the application. Fortunately, we do not need to write directly the XML file. Instead, we can edit the user interface using the Qt WYSIWYG editor: the *Design* mode. When double-clicking on the mainwindow.ui file listed in the *Projects Pane*, Qt Creator switches to the *Design* mode (Fig. 3-bottom). The interface of *Design* mode contains (Fig. 4): (A) components to design the UI; (B) UI WYSIWYG editor; (C) hierarchy of currently created UI components; (D) properties of the UI components. To come back to the text editor, the *Edit* mode, click on the

Edit button in the leftmost pane of Qt Creator.

```
1  QT       += core gui
3  TARGET   = myqtglproject
5  TEMPLATE = app
7  SOURCES += main.cpp\
            mainwindow.cpp
9
   HEADERS += mainwindow.h
11
   FORMS   += mainwindow.ui
```

Listing 1. File **myqtglproject.pro**: initial configuration of the Qt project.

```
QT       += core gui opengl
```

Listing 2. File **myqtglproject.pro**: including *QtOpenGL Module* at myqtglproject.pro.

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
3
#include <QMainWindow>
5
namespace Ui {
7  class MainWindow;
}
9
class MainWindow : public QMainWindow
11 {
    Q_OBJECT
13
public:
15     explicit MainWindow(QWidget *parent = 0);
       ~MainWindow();
17
private:
19     Ui::MainWindow *ui;
};
21
#endif // MAINWINDOW_H
```

Listing 3. File **mainwindow.h**: automatically generated by Qt Creator.

```
#include "mainwindow.h"
2  #include "ui_mainwindow.h"
4  MainWindow::MainWindow(QWidget *parent) :
       QMainWindow(parent),
6      ui(new Ui::MainWindow)
{
8      ui->setupUi(this);
}
10
MainWindow::~MainWindow()
12 {
    delete ui;
14 }
```

Listing 4. File **mainwindow.cpp**: automatically generated by Qt Creator.

```
#include <QtGui/QApplication>
2  #include "mainwindow.h"
4  int main(int argc, char *argv[])
{
6      QApplication a(argc, argv);
       MainWindow w;
8      w.show();
10     return a.exec();
}
```

Listing 5. File **main.cpp**: automatically generated by Qt Creator.

Tip 2. *It is also possible to include non-Qt libraries in the Qt project. We show an example in Listing 6, where we include hypothetical libraries* foo *and* bar *in the* .pro *file. In this case, we assume they are located at* /usr/local/lib/foo *and their header files are at* /usr/loca/include/foo.

```
INCLUDEPATH += /usr/local/include/foo/

LIBS += -L/usr/local/lib/foo/ -lfoo -lbar
```

Listing 6. File **myqtglproject.pro**: including non-Qt libraries in the Qt project.

Before building the project, the qmake *[13] tool uses the configuration from the* .pro *file to automatically generate, in the shadow building directory, the Makefile file for the target platform.* qmake *is also able to generate project files for Apple Xcode and Microsoft Visual Studio. Further details about* qmake *and its configuration variables are found on [13].*

Five UI objects are automatically generated by Qt Creator (Fig. 4-(C)): MainWindow, of class QMainWindow, is the main window object and the root node of the hierarchy of UI objects. Its children are centralWidget of class QWidget, menuBar of class QMenuBar, mainToolBar of class QToolBar and statusBar of class QStatusBar. We can use the *Design* mode to create new child objects. Fig. 5 lists all UI objects of our application. They will be detailed along the next sections.
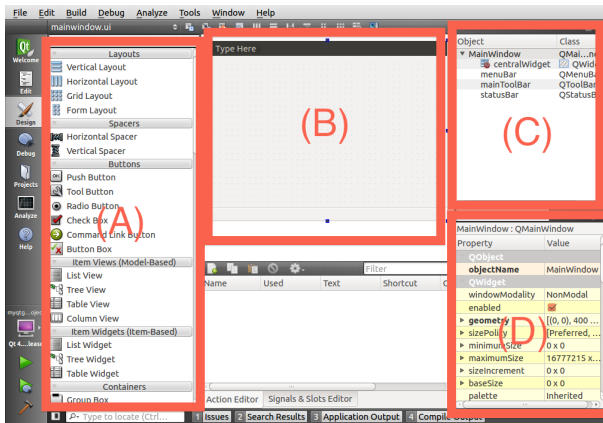

Fig. 4. Qt Creator (*Design* mode): (A) components to design the UI, (B) UI WYSIWYG editor, (C) hierarchy of currently created UI components and (D) properties of UI components.

Fig. 5. UI objects of our application: some of them were automatically generated by Qt Creator and others were added using the *Design* mode.

So far we learnt how to generate a Qt Project. Our next step is to create an OpenGL interactive application in this project. We show how to do it in a two-step process: in the first, we use

the *Edit* mode to create a *custom* class that supports OpenGL functionalities (Sec. II-B). In the second, we use the *Design* mode to associate a widget to an object of the previously created *custom* class (Sec. II-C) for displaying the OpenGL scene.

B. Edit *mode: extending the class* QGLWidget *to our OpenGL application*

The Qt class QWidget provides the widget base for all UI objects. In particular, it is the base for the widget class QGLWidget used for rendering OpenGL graphics [14]. QGLWidget provides virtual methods that should be implemented to perform common OpenGL tasks. Three of them are the most used and will be implemented in our applications:

- paintGL(): renders the OpenGL scene whenever the widget needs to be repainted. It is equivalent to the callback function registered through the GLUT function glutDisplayFunc().
- resizeGL(int width, int height): handles the resizing of the OpenGL window and repaints it. It is equivalent to the callback function registered through the GLUT function glutReshapeFunc().
- initializeGL(): it is called whenever the widget is assigned to a new OpenGL context. It is intended to contain the OpenGL initialization code that comes before the first call to paintGL() or resizeGL().

We will define these methods in our custom class GLWidget, derived from the base class QGLWidget. An easy approach to create a new C++ class in a Qt project is as follows: click at **File → New File or Project** to open a dialog window (Fig. 6-(a)). Choose the C++ class template and fill-in all fields (Fig. 6-(b)). Our base class is a QGLWidget and inherits from QWidget. In Fig. 6-(c) we finish the class creation. Qt Creator automatically creates both .h and .cpp files for the GLWidget class (Listings 7 and 8). Notice that Qt does not automatically declare virtual methods of the parent class. We add the code for paintGL(),resizeGL() and initializeGL() on the class files, as depicted in Listings 9 and 10.

```
#ifndef GLWIDGET_H
#define GLWIDGET_H

#include <QGLWidget>

class GLWidget : public QGLWidget
{
    Q_OBJECT
public:
    explicit GLWidget(QWidget *parent = 0);

signals:

public slots:

};

#endif // GLWIDGET_H
```

Listing 7. File **glwidget.h**: source code automatically generated by Qt Creator.

```
#include "glwidget.h"

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

Listing 8. File **glwidget.cpp**: source code automatically generated by Qt Creator.

```
...
class GLWidget : public QGLWidget
{
    ...
protected:
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
};
```

Listing 9.   File **glwidget.h**: declaration of three methods in class `GLWidget`.

```
void GLWidget::initializeGL()
{
}

void GLWidget::resizeGL(int width, int height)
{
}

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

Listing 10.   File **glwidget.cpp**: first implementations.

Tip 3. *There is an include file for each class of the* QtOpenGL *Module. Several of these classes are used in our project. Instead of including each one, we can simply write* #include <QtOpenGL> *at* glwidget.h. *The complete listing of* glwidget.h *is in Appendix A.*

*C.* Design *mode: promoting a* QWidget *to* GLWidget

Now we need to add to the main window a widget that displays the OpenGL graphics handled by our custom class `GLWidget`. However, `GLWidget` is not in the list of UI components available in the *Design* mode (Fig. 4-(A)). To overcome that we will initially use the base widget QWidget of GLWidget as a placeholder and then *promote* it to GLWidget.

Following the steps in Fig. 7, first we drag and drop an **Widget** onto the object `centralWidget` (Fig. 7-(a)), creating an object named `widget` of class `QWidget`. As we need that the placeholder `widget` have the specialization of `GLWidget`, we apply a *Promote Operation*: right-click on the `widget` object and click on **Promote to ...** (Fig. 7-(b)). At the new dialog window, type `GLWidget` at **Promoted class name** and select **QWidget** at **Base class name**. Click on **Add** and finally click on **Promote** (Fig. 7-(c)).

Tip 4. *By default,* widget *has a fixed size. In order to automatically adjust its size when* MainWindow *is resized, we must set the* **sizePolicy – Horizontal/Vertical** *property of the* widget *(Fig. 4-(D)) from the default option –* **Preferred** *– to* **Expanding**. *Other resize policies are also available.*

Tip 5. *UI components can be consistently arranged within widgets using Qt layout styles [15]. We can choose, for instance, the Vertical, Horizontal or Grid layouts. In our application,* centralWidget *is laid out vertically. In* Design *mode, this is done by clicking at* centralWidget *and then either pressing Ctrl+L or clicking on the icon* ▦ *in the Qt Creator toolbar.*

To run our application, either click on the icon **Run** ▶ in the leftmost Qt Creator pane or press Ctrl+R.
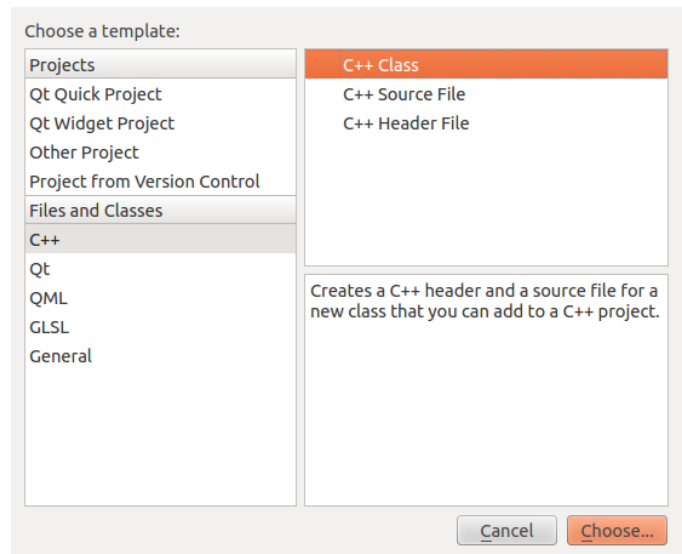
(a)
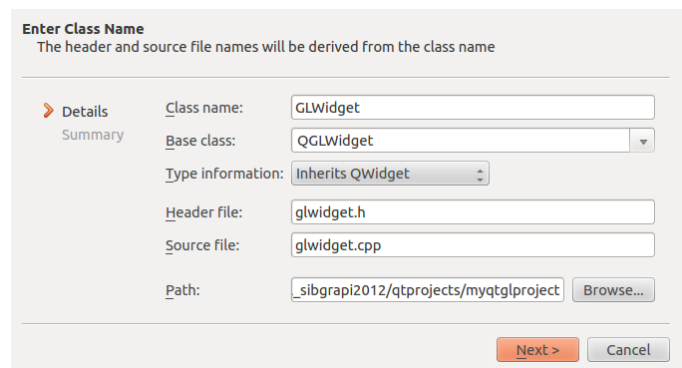
(b)
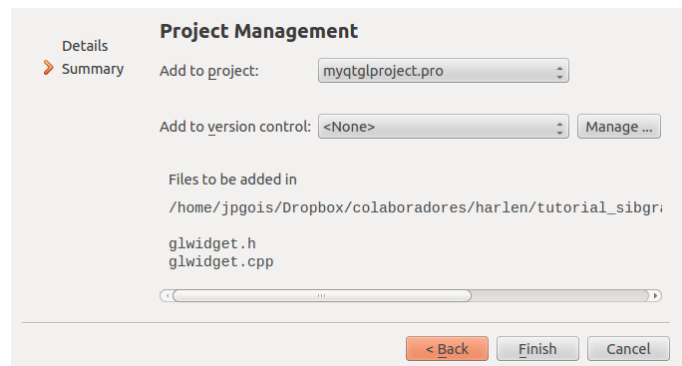
(c)

Fig. 6.   Creating the class `GLWidget` extended from `QGLWidget`: on (a) we select **C++ Class**, on (b) we name the class, define the base class `QGLWidget` and set type information **Inherits QWidget**. On (c) we finalize.

D. Introducing Signals and Slots

OpenGL applications using GLUT [2] make use of the concept of *callback functions* to handle window events, input device events and timers. With Qt, callback functions can be avoided as it provides not only *methods* to handle events (Sec. II-I) but also a mechanism for object intercommunication,
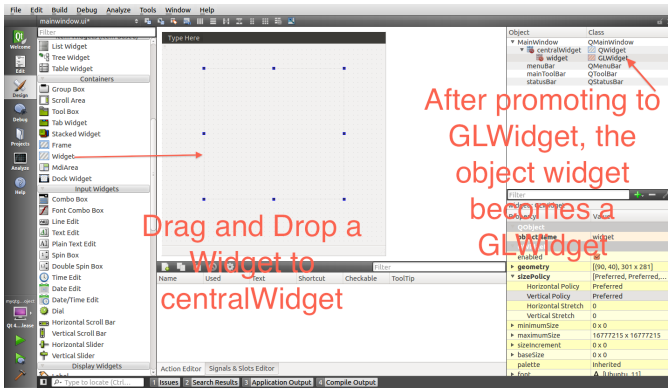
After promoting to GLWidget, the object widget becomes a GLWidget

Drag and Drop a Widget to centralWidget

(a)
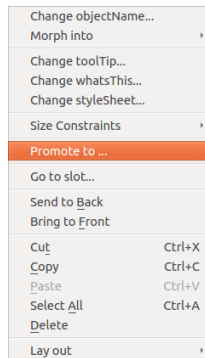
(b)
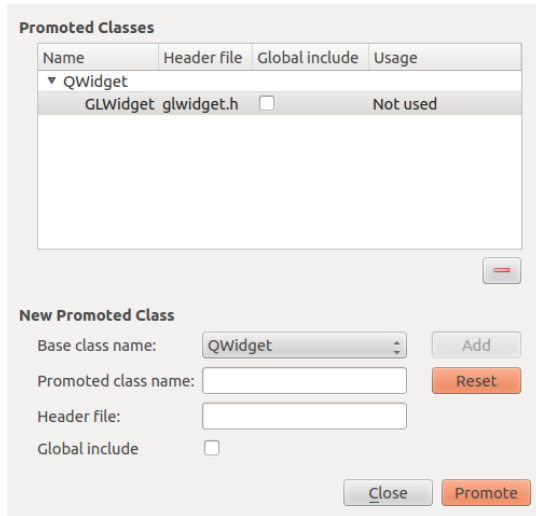
(c)

Fig. 7. Promoting the class `QWidget` to `GLWidget`.

named *Signals* and *Slots* [16].

A *signal* is *emitted* when an event associated to some *sender* object is triggered, for instance, when a push button is clicked in the UI. A *slot*, in turn, is a method of a *receiver* object which is called in response to a particular signal. Slots can be qualified as `private`, `protected` or `public`. Signals are implicitly declared as `protected`. Differently from callback functions, signal and slots are type-safe, *i.e.*, the signature of a specific sender signal must match the signature of the receiver slot. The complete description of signals and slots is found on [16].

First we will show how to connect a pre-defined signal to a pre-defined slot for creating an *Exit* push button that quits the application. At *Design* mode, drag and drop a **QPushButton** to `centralWidget` and change the **text** property of this newly created button from **PushButton** to **Exit** (Fig. 8-(a)).

The signal corresponds to the event of pressing the button. It is associated to the method `clicked()` of the object `pushButton` of `QPushButton` class. The slot quits the application. It is associated to the method `close()` of the object `MainWindow` of `QMainWindow` class. Qt Creator provides different ways to make this signal/slot connection. We will show how to do that in the *Design* mode.

Click at the **Signal & Slots Editor** tab (highlighted at Fig. 8-(b)). At this moment, there is no signal/slot connection as shown in Fig. 8-(a). To add our first signal/slot connection, click on the ![+] icon. A row with four combo boxes appears, which are changed as follows:

- **Sender** combo box: select `pushButton`;
- **Signal** combo box: select the signal `clicked()` of `pushButton`;
- **Receiver** combo box: select `MainWindow`;
- **Slot** combo box: select the signal `close()` of `MainWindow`.

At this moment we had concluded our first signal/slot connection.

Drag and Drop a Push Button

Change text to 'Exit'

(a)

Click on + to insert a connection between a signal and a slot
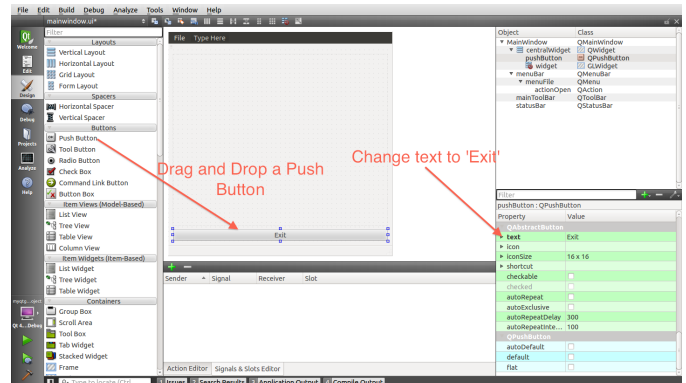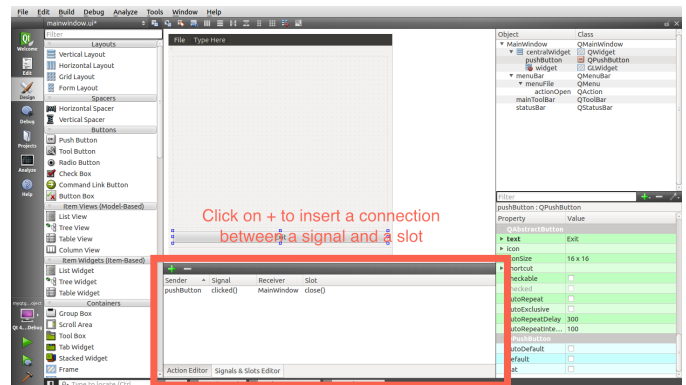
(b)

Fig. 8. Signal/slot connection: on (a) we create a button, whereas on (b) we make a signal/slot connection.

Now we will exemplify another signal/slot connection, where the signal is pre-defined but the slot is customized. We will create a checkbox button (Fig. 9) that toggles the background color of the `widget` between white and black using OpenGL commands. The signal is the action of toggling the checkbox, whereas the slot is a method of `GLWidget` that changes the OpenGL clear color. Before we make the signal/slot connection, let us create the custom slot:

1) In *Edit* mode, declare the custom slot as shown in Listing 11;
2) In *Edit* mode, define the custom slot as shown in Listing 12;
3) In *Design* mode, insert the new signature of the custom slot to the Signals/Slots dialog of the receiver object: to open this dialog, right-click on `widget` and select **Change signals/slots...** (Fig. 10-(a)). Note that this option is only available to our custom OpenGL widget. Now click at the plus button located right below the *Slots* list widget (Fig. 10-(b)) and enter the signature of the custom slot: `toggleBackgroundColor(bool)`. Click **OK** to finish.
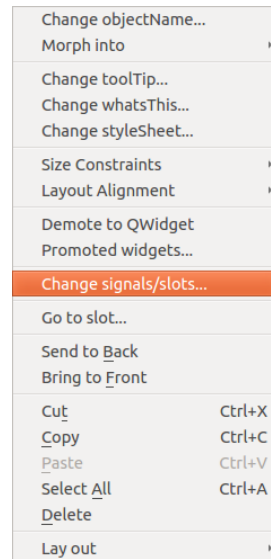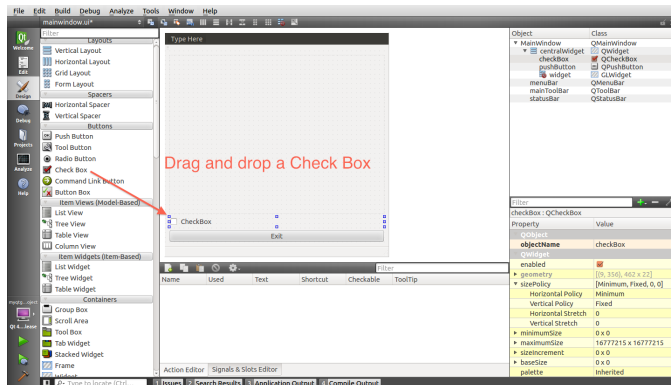

(a)
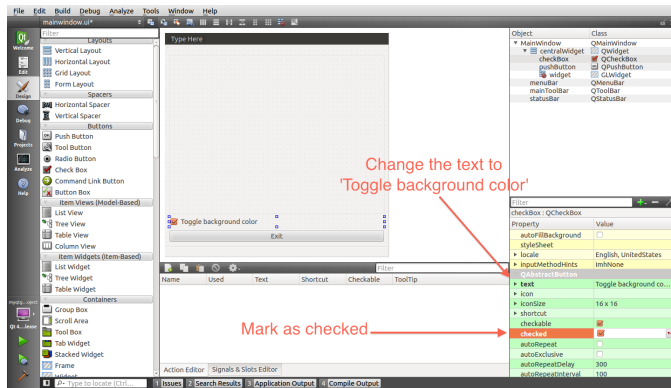
(b)

Fig. 9. Creating a checkbox button: on (a) we drag and drop a **Check Box** to the `centralWidget`. On (b) we rename it and set its initial state to **checked**.

After these steps we proceed as in the previous signal/slot connection example:

- **Sender** combo box: select `checkBox`;
- **Signal** combo box: select the signal `toggled(bool)` of `checkBox`;

(a)
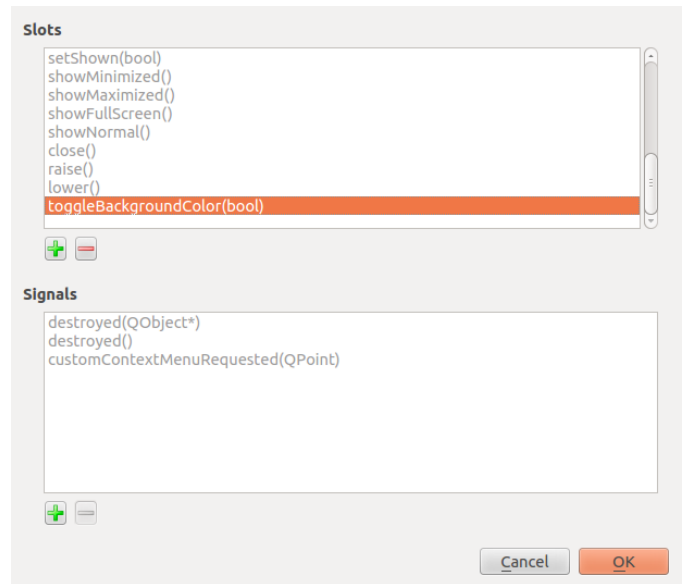
(b)

Fig. 10. Inserting the new signature of the custom slot to the Signals/Slots dialog: the menu on (a) is displayed after a right-click on the `widget`. On (b) we add the new slot signature.

- **Receiver** combo box: select `widget`;
- **Slot** combo box: select the signal `toggleBackgroundColor(bool)` of `widget`.

As type-safe methods, the signatures of both sender and receiver methods must match. The parameter of `toggleBackgroundColor()` must be of the same type of `toggled()` – the `bool` type. It is also acceptable for a signal to have a signature with more arguments than a slot has. In this case the extra arguments are ignored [16].

```
...
class GLWidget : public QGLWidget
{
    ...
    public slots:
```

```
6        void toggleBackgroundColor(bool toBlack);
    ...
8  };
   #endif // GLWIDGET_H
```
Listing 11. File **glwidget.h**: defining a slot.

```
1  void GLWidget::toggleBackgroundColor(bool toBlack)
   {
3      if (toBlack)
           glClearColor(0, 0, 0, 1);
5      else
           glClearColor(1, 1, 1, 1);
7
       updateGL();
9  }
```
Listing 12. File **glwidget.cpp**: implementing the slot.

Obs 1. *Observe in Listing 12 the* QGLWidget *command* updateGL(). *We will use it whenever we need to repaint the* widget.

E. Loading a Geometric Model

Our OpenGL application displays a triangular mesh model loaded from a file in *Object File Format* (OFF) format [17]. The user can choose the OFF file from a file selection dialog, accessed by **File → Open** on the menuBar.

To create such options **File → Open**, at *Design* mode click at **Type Here** on the menuBar and rename it to **File**. Qt Creator automatically creates the object menuFile of the class QMenu and opens a submenu. Click at **Type Here** on the submenu and change it to **Open**. Qt Creator automatically creates the object actionOpen of the class QAction.

We want that, whenever the user selects **Open** at menu bar, the emitted signal, QAction::triggered(), connects to a slot that displays a file selection dialog. We declare the slot as shown in Line 27 at Appendix A and define it as shown in Listing 13.

```
1  void GLWidget::showFileOpenDialog()
   {
3      QByteArray fileFormat = "off";
       QString fileName;
5      fileName = QFileDialog::getOpenFileName(this,
         "Open File",
7        QDir::homePath(),
         QString("%1 Files (*.%2)")
9        .arg(QString(fileFormat.toUpper()))
         .arg(QString(fileFormat)));
11     if (!fileName.isEmpty()) {
           readOFFFile(fileName);
13
           genNormals();
15         genTexCoordsCylinder();
           genTangents();
17
           createVBOs();
19         createShaders();
21         updateGL();
       }
23 }
```
Listing 13. File **glwidget.cpp**: slot showFileOpenDialog().

Before detailing the new commands in the previous listing, let us conclude this signal/slot connection:

- **Sender** combo box: select actionOpen;
- **Signal** combo box: select the signal triggered() of actionOpen;
- **Receiver** combo box: select widget;
- **Slot** combo box: select the signal showFileOpenDialog() of widget.

In Listing 13, Line 5, the method QFileDialog::getOpenFileName() displays file open dialog box and returns the string containing the complete file path. At Line 12, the string is passed to the method readOFFFile() (Listing 14), which reads the OFF file using the C++ Standard Library class std::ifstream. In the array GLWidget::vertices, the mesh vertices are stored as QVector4D objects. The face indices are stored in the array GLWidget::indices as unsigned int values. The number of vertices and faces are stored in GLWidget::numVertices and GLWidget::numFaces, respectively. This method also centers and resizes the mesh.

```
1  void GLWidget::readOFFFile(const QString &fileName)
   {
3      std::ifstream stream;
       stream.open(fileName.toAscii(), std::ifstream::in);
5
       if (!stream.is_open()) {
7          qWarning("Cannot open file.");
           return;
9      }
11     std::string line;
13     stream >> line;
       stream >> numVertices >> numFaces >> line;
15
       delete[] vertices;
17     vertices = new QVector4D[numVertices];
19     delete[] indices;
       indices = new unsigned int[numFaces * 3];
21
       if (numVertices > 0) {
23         double minLim = std::numeric_limits<double>::min();
           double maxLim = std::numeric_limits<double>::max();
25         QVector4D max(minLim, minLim, minLim, 1.0);
           QVector4D min(maxLim, maxLim, maxLim, 1.0);
27
           for (unsigned int i = 0; i < numVertices; i++) {
29             double x, y, z;
               stream >> x >> y >> z;
31             max.setX(qMax(max.x(), x));
               max.setY(qMax(max.y(), y));
33             max.setZ(qMax(max.z(), z));
               min.setX(qMin(min.x(), x));
35             min.setY(qMin(min.y(), y));
               min.setZ(qMin(min.z(), z));
37
               vertices[i] = QVector4D(x, y, z, 1.0);
39         }
41         QVector4D midpoint = (min + max) * 0.5;
           double invdiag = 1 / (max - min).length();
43
           for (unsigned int i = 0; i < numVertices; i++) {
45             vertices[i] = (vertices[i] - midpoint)*invdiag;
               vertices[i].setW(1);
47         }
       }
49
       for (unsigned int i = 0; i < numFaces; i++) {
51         unsigned int a, b, c;
           stream >> line >> a >> b >> c;
53         indices[i * 3    ] = a;
           indices[i * 3 + 1] = b;
55         indices[i * 3 + 2] = c;
       }
57
       stream.close();
59 }
```
Listing 14. File **glwidget.cpp**: method readOFFFile().

In Listing 13, Line 14, the method genNormal() (Listing 15) estimates the normals at the mesh vertices and stores them in the array GLWidget::normals of QVector3D objects.

```
1  void GLWidget::genNormals()
   {
3      delete[] normals;
       normals = new QVector3D[numVertices];
5
       for (unsigned int i = 0; i < numFaces; i++) {
```

```cpp
        unsigned int i1 = indices[i * 3    ];
        unsigned int i2 = indices[i * 3 + 1];
        unsigned int i3 = indices[i * 3 + 2];

        QVector3D v1 = vertices[i1].toVector3D();
        QVector3D v2 = vertices[i2].toVector3D();
        QVector3D v3 = vertices[i3].toVector3D();

        QVector3D faceNormal = QVector3D::crossProduct(v2 -
            v1, v3 - v1);
        normals[i1] += faceNormal;
        normals[i2] += faceNormal;
        normals[i3] += faceNormal;
    }

    for (unsigned int i = 0; i < numVertices; i++)
        normals[i].normalize();
}
```

Listing 15. File **glwidget.cpp**: method genNormals().

In Listing 13, Line 15, the method genTexCoordsCylinder()
(Listing 16) generates cylindrical texture coordinates for the
mesh vertices. They are stored in the array GLWidget::texCoords
of QVector2D objects.

```cpp
void GLWidget::genTexCoordsCylinder()
{
    delete[] texCoords;
    texCoords = new QVector2D[numVertices];

    double minLim = std::numeric_limits<double>::min();
    double maxLim = std::numeric_limits<double>::max();
    QVector2D max(minLim, minLim);
    QVector2D min(maxLim, maxLim);

    for (unsigned int i = 0; i < numVertices; i++) {
        QVector2D pos = vertices[i].toVector2D();
        max.setX(qMax(max.x(), pos.x()));
        max.setY(qMax(max.y(), pos.y()));
        min.setX(qMin(min.x(), pos.x()));
        min.setY(qMin(min.y(), pos.y()));
    }

    QVector2D size = max - min;
    for (unsigned int i = 0; i < numVertices; i++) {
        double x = 2.0 * (vertices[i].x() - min.x()) /
                        size.x() - 1.0;
        texCoords[i] = QVector2D(acos(x) / M_PI,
                        (vertices[i].y() - min.y()) /
                        size.y());
    }
}
```

Listing 16. File **glwidget.cpp**: method **genTexCoordsCylinder**().

In Listing 13, Line 16, the method genTangents() (Listing
17) estimates per-vertex tangent vectors required by Normal
Mapping [10]. Our code is based on the method described by
Lengyel [18], [19]. The tangent vectors are stored in the array
GLWidget::tangents of QVector4D objects.

```cpp
void GLWidget::genTangents()
{
    delete[] tangents;

    tangents = new QVector4D[numVertices];
    QVector3D *bitangents = new QVector3D[numVertices];

    for (unsigned int i = 0; i < numFaces; i++) {
        unsigned int i1 = indices[i * 3    ];
        unsigned int i2 = indices[i * 3 + 1];
        unsigned int i3 = indices[i * 3 + 2];

        QVector3D E = vertices[i1].toVector3D();
        QVector3D F = vertices[i2].toVector3D();
        QVector3D G = vertices[i3].toVector3D();

        QVector2D stE = texCoords[i1];
        QVector2D stF = texCoords[i2];
        QVector2D stG = texCoords[i3];

        QVector3D P = F - E;
        QVector3D Q = G - E;

        QVector2D st1 = stF - stE;
        QVector2D st2 = stG - stE;
```

```cpp
        QMatrix2x2 M;
        M(0,0) =  st2.y(); M(0,1) = -st1.y();
        M(1,0) = -st2.x(); M(1,1) =  st1.x();
        M *= (1.0 / (st1.x()*st2.y() - st2.x()*st1.y()));

        QVector4D T = QVector4D(M(0,0)*P.x()+M(0,1)*Q.x(),
                                M(0,0)*P.y()+M(0,1)*Q.y(),
                                M(0,0)*P.z()+M(0,1)*Q.z(),
                                0.0);

        QVector3D B = QVector3D(M(1,0)*P.x()+M(1,1)*Q.x(),
                                M(1,0)*P.y()+M(1,1)*Q.y(),
                                M(1,0)*P.z()+M(1,1)*Q.z());

        tangents[i1] += T;
        tangents[i2] += T;
        tangents[i3] += T;

        bitangents[i1] += B;
        bitangents[i2] += B;
        bitangents[i3] += B;
    }

    for (unsigned int i = 0; i < numVertices; i++) {
        const QVector3D& n = normals[i];
        const QVector4D& t = tangents[i];

        tangents[i] = (t - n * QVector3D::dotProduct(n,
                    t.toVector3D())).normalized();

        QVector3D b = QVector3D::crossProduct(n,
                    t.toVector3D());
        double hand = QVector3D::dotProduct(b,
                    bitangents[i]);
        tangents[i].setW((hand < 0.0) ? -1.0 : 1.0);
    }

    delete[] bitangents;
}
```

Listing 17. File **glwidget.cpp**: Method genTangents().

The remaining methods, createVBOs() and createShaders(),
will be detailed in due course.

F. Encapsulating Resources

Graphics applications commonly make use of external asset
files such as textures, icons, XML data and other text or binary
files.

Qt provides a mechanism called *Resource System* [20]
that stores external binary and text files into the application
executable, encapsulating both application and their resources
into a single binary file. This embedment of resource files is
done during the build process. In our application, we will use
this mechanism to store textures and GLSL programs.

In order to use the Resource System, we first add to our Qt
project a *resource collection file* [20]. This is an XML file, with
extension .qrc, that lists the resource files to be embedded.
Again, we do not have to edit this XML file directly because
Qt Creator provides a friendly interface, the *Resource* Editor,
for managing resources (Fig. 12).

In Qt Creator, first select **File**→**New File or Project** then
select **Qt** and **Qt Resource file** (Fig. 11-(a)). We name it
resources.qrc (Fig. 11-(b)).

Double click the file resources.qrc at Qt Creator to open the
Resource Editor. Before we include the resource files, we must
create a *path prefix* to organize the lists of files. We will create
two prefixes, one for GLSL programs and other for texture
files. Click at **Add** → **Add Prefix** and type /shaders. Repeat
the process and type /textures. Now, to include resource files,
click at the corresponding prefix and at **Add** → **Add Files**
(Fig. 12). In Sections II-G and II-L we show how to access
these resources.
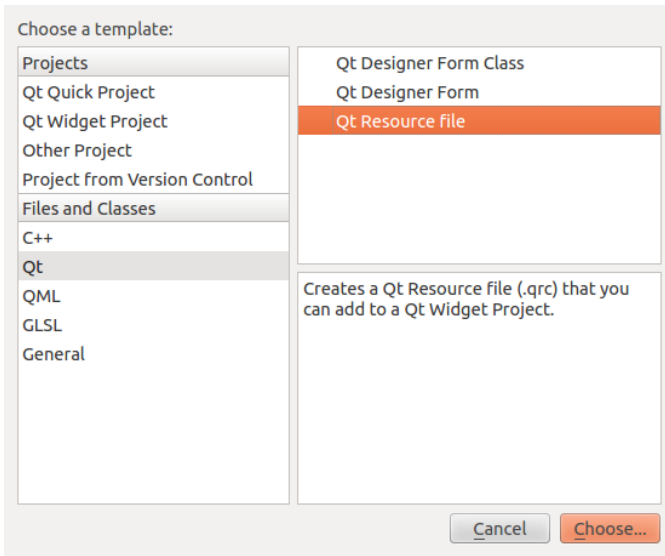
(a)
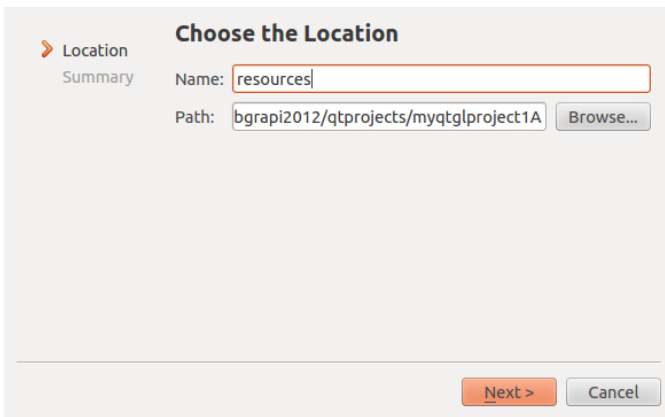
(b)

Fig. 11. Creating a *Resource System* using Qt Creator: on (a) we select the option **Qt Resource file**. On (b) we name our *Resource System* as `resources`.

Obs 2. *Resource files embedded into the application executable are only accessible by Qt classes. For instance, if we want to open a text file managed by the Resource System, we cannot do that using* `std::ifstream`*, instead we should use* `QFile` *[21].*

Obs 3. *If we try to run the application but Qt Creator produces an error message related to resource files, make sure to run* **Qmake** *by selecting, on the Qt Creator menu bar,* **Build** → **Run qmake** *before trying to run the application again.*

G. OpenGL Shaders and Qt

Qt provides facilities to handle GLSL shaders and shader programs. Qt Creator features a built-in GLSL editor, while the *QtOpenGL* Module implements the classes `QGLShader` and `QGLShaderProgram`. The former class allows OpenGL shaders to be compiled and the later class allows OpenGL shader programs to be linked and bound to the OpenGL pipeline. In the Qt Creator, GLSL shaders can be included into the Qt project similarly as adding C++ classes and resources.
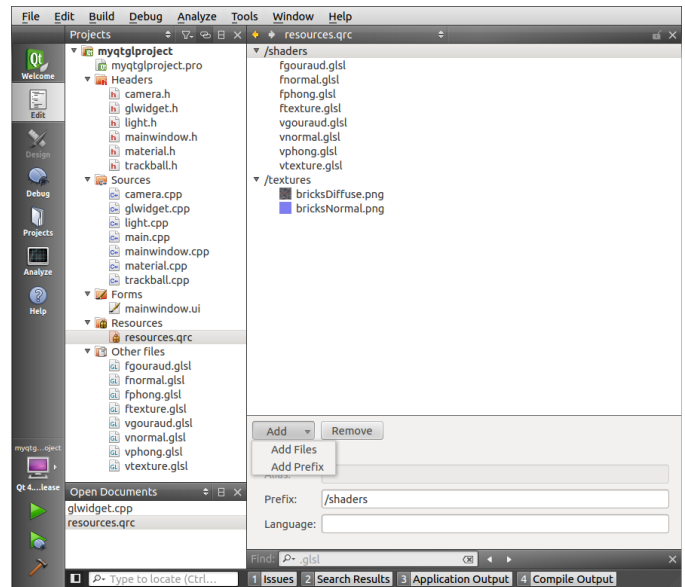

Fig. 12. Using the Resource System in our application: there are two prefixes, `/shaders` and `/textures`.

In our application, the user can use the keyboard to select among four shader effects that render a:

- Gouraud shaded model;
- Phong shaded model;
- Phong shaded model with texture mapping;
- Phong shaded model with normal mapping.

The source code of these shaders are listed in Appendix C.

When the user selects a shader effect, the previously selected is firstly *released* from the OpenGL pipeline in order to the shaders of the current effect be *compiled*, *linked* and *bound* to the OpenGL application in run time. Details about compiling, linking and binding GLSL shaders are found in [8], [9], [10].

In Listing 18, we present our methods responsible to release and destroy the previous shader program and to compile and link the new one. The binding will be done in the `paintGL()` method (Sec. II-M, Listing 27).

It is worth to mention that we opt to use Qt Resource System to store our GLSL shaders. However, we could instead load them as text strings into the C++ sources or from external files [8], [9], [10].

In order to load the shader files managed by the Qt Resource System to the application, their paths must begin with a colon followed by the resource prefix that we specified in the *Resource* Editor, *i.e.*, `:/shaders` (Listing 18, Lines 5 and 11). On Lines 18 and 22 we create `QGLShader` objects for vertex and fragment shaders. On Lines 19 and 23 the corresponding GLSL source files stored in the Resource System are compiled. In our application, only the vertex and fragment shaders of the currently selected effect are compiled.

After compiling the shaders (using `compileSourceFile()`), we add (`addShader()`) them to the object `shaderProgram` of class `QGLShaderProgram` (Lines 27 and 28) and link (`link()`)

the shader program (Line 30).

```cpp
void GLWidget::createShaders()
{
    destroyShaders();

    QString vertexShaderFile[] = {
        ":/shaders/vgouraud.glsl",
        ":/shaders/vphong.glsl",
        ":/shaders/vtexture.glsl",
        ":/shaders/vnormal.glsl"
    };
    QString fragmentShaderFile[] = {
        ":/shaders/fgouraud.glsl",
        ":/shaders/fphong.glsl",
        ":/shaders/ftexture.glsl",
        ":/shaders/fnormal.glsl"
    };

    vertexShader = new QGLShader(QGLShader::Vertex);
    if (!vertexShader->compileSourceFile(vertexShaderFile[
        currentShader]))
        qWarning() << vertexShader->log();

    fragmentShader = new QGLShader(QGLShader::Fragment);
    if (!fragmentShader->compileSourceFile(
        fragmentShaderFile[currentShader]))
        qWarning() << fragmentShader->log();

    shaderProgram = new QGLShaderProgram;
    shaderProgram->addShader(vertexShader);
    shaderProgram->addShader(fragmentShader);

    if (!shaderProgram->link())
        qWarning() << shaderProgram->log() << endl;
}

void GLWidget::destroyShaders()
{
    delete vertexShader;
    vertexShader = NULL;

    delete fragmentShader;
    fragmentShader = NULL;

    if (shaderProgram) {
        shaderProgram->release();
        delete shaderProgram;
        shaderProgram = NULL;
    }
}
```

Listing 18. File **glwidget.cpp**: methods destroyShaders() and createShaders().

In Sec. II-M, where the method paintGL() is detailed, we will present how to set up the input *attributes* and *uniform* variables of the shader program currently in use.

H. Buffer Objects

For a long time, the OpenGL's *immediate mode* and *display lists* [9] have been the standard approach to upload rendering data to the graphics server, but both features are now deprecated. Another option is to use OpenGL *vertex arrays*, but they are stored in the client side, which means that they must be sent to the server every time the scene is updated.

The concept of *Vertex Buffer Objects* (VBOs) [8], [22], introduced in OpenGL 1.5, allows the direct manipulation of the data stored in the server side. This is the approach used in our examples.

Qt provides the class QGLBuffer [23] to handle buffer objects. In our application we make use of four vertex buffers, according to Listing 19:

- vboVertices: vertex positions of type QVector4D (Line 5);
- vboNormals: vertex normals of type QVector3D (Line 13);
- vboTexCoords: texture coordinates of type QVector2D (Line 21);
- vboTangents: tangent vectors of type QVector4D (Line 29).

For each vertex buffer, we first create a buffer object using the parameter QGLBuffer::VertexBuffer in the constructor of QGLBuffer. We then create (create()) the buffer in the server and associate (bind()) it to the current OpenGL context. Since our vertex data will be set only once and used many times for rendering, we set the QGLBuffer::StaticDraw usage pattern at setUsagePattern(). Additional patterns can be found in the documentation [23].

We use the method allocate() to send the vertex data to the VBO as a contiguous untyped data. According to the specified usage pattern, OpenGL will decide where the data must be stored and when it must be sent [22]. Lastly, we can delete the array of vertex attributes passed to allocate() because it was already copied to the VBO.

We also create a buffer object to the indices of the mesh triangles (vboIndices, Lines 37–43). It only differs from vertex buffers, on the use of QGLBuffer::IndexBuffer in the constructor.

```cpp
void GLWidget::createVBOs()
{
    destroyVBOs();

    vboVertices = new QGLBuffer(QGLBuffer::VertexBuffer);
    vboVertices->create();
    vboVertices->bind();
    vboVertices->setUsagePattern(QGLBuffer::StaticDraw);
    vboVertices->allocate(vertices, numVertices *
                          sizeof(QVector4D));
    delete[] vertices;
    vertices = NULL;

    vboNormals = new QGLBuffer(QGLBuffer::VertexBuffer);
    vboNormals->create();
    vboNormals->bind();
    vboNormals->setUsagePattern(QGLBuffer::StaticDraw);
    vboNormals->allocate(normals, numVertices *
                         sizeof(QVector3D));
    delete[] normals;
    normals = NULL;

    vboTexCoords = new QGLBuffer(QGLBuffer::VertexBuffer);
    vboTexCoords->create();
    vboTexCoords->bind();
    vboTexCoords->setUsagePattern(QGLBuffer::StaticDraw);
    vboTexCoords->allocate(texCoords, numVertices *
                           sizeof(QVector2D));
    delete[] texCoords;
    texCoords = NULL;

    vboTangents = new QGLBuffer(QGLBuffer::VertexBuffer);
    vboTangents->create();
    vboTangents->bind();
    vboTangents->setUsagePattern(QGLBuffer::StaticDraw);
    vboTangents->allocate(tangents, numVertices *
                          sizeof(QVector4D));
    delete[] tangents;
    tangents = NULL;

    vboIndices = new QGLBuffer(QGLBuffer::IndexBuffer);
    vboIndices->create();
    vboIndices->bind();
    vboIndices->setUsagePattern(QGLBuffer::StaticDraw);
    vboIndices->allocate(indices, numFaces * 3 *
                         sizeof(unsigned int));
    delete[] indices;
    indices = NULL;
}

void GLWidget::destroyVBOs()
{
    if (vboVertices) {
        vboVertices->release();
        delete vboVertices;
        vboVertices = NULL;
    }

    if (vboNormals) {
        vboNormals->release();
        delete vboNormals;
        vboNormals = NULL;
    }
```

```
       if (vboTexCoords) {
66         vboTexCoords ->release();
           delete vboTexCoords;
68         vboTexCoords = NULL;
       }
70
       if (vboTangents) {
72         vboTangents ->release();
           delete vboTangents;
74         vboTangents = NULL;
       }
76
       if (vboIndices) {
78         vboIndices ->release();
           delete vboIndices;
80         vboIndices = NULL;
       }
82 }
```

Listing 19.   File **glwidget.cpp**: methods `createVBOs()` and `destroyVBOs()`.

Obs 4. *A key feature of the buffer objects is their capability to map their data into the client side. These allow us to efficiently update VBO data. In Listing 20 we show a simple example, where the methods* `map()`/`unmap()` *are employed to this purpose. Further options of buffer object mappings are found in [23].*

```
  vboVertices ->setUsagePattern(QGLBuffer::DynamicDraw);
2 ...
  vboVertices ->bind();
4 QVector4D* pt = (QVector4D*) vboVertices ->map(QGLBuffer::
      WriteOnly);
  //change here the VBO data by directly manipulating pt array
6 ...
  vboVertices ->unmap();
```

Listing 20.   An example of mapping VBO data into client side.

I. Mouse and Keyboard Events

So far we learnt with signals and slots how to communicate between Qt objects. In this section, we will see that Qt also offers the capability to handle *events* from the window system [24]. In particular, we are interested in events generated by mouse and keyboard. Recalling GLUT, the mouse and keyboard events are handled by callback functions [2]. Qt, in turn, provides virtual methods in `QGLWidget` which are called in response to mouse or keyboard events.

In Listing 21 we present the method `keyPressEvent()` that handles keyboard events. This is the method responsible to switch among the shader effects (Sec. II-G) by pressing the keys 0 (Gouraud shading), 1 (Phong shading), 2 (Phong + texture) and 3 (normal mapping). Also, pressing the escape key, the application quits (`qApp` is a Qt global pointer to the unique application object).

```
  void GLWidget::keyPressEvent(QKeyEvent *event)
2 {
      switch(event->key())
4     {
      case Qt::Key_0:
6         currentShader = 0;
          createShaders();
8         updateGL();
          break;
10    case Qt::Key_1:
12        currentShader = 1;
          createShaders();
          updateGL();
14        break;
      case Qt::Key_2:
16        currentShader = 2;
          createShaders();
18        updateGL();
          break;
20    case Qt::Key_3:
          currentShader = 3;
```

```
22        createShaders();
          updateGL();
24        break;
      case Qt::Key_Escape:
26        qApp->quit();
      }
28 }
```

Listing 21.   File **glwidget.cpp**: method `keyPressEvent()`.

Obs 5. *By default, widgets are set to not receive the keyboard or mouse focus. Therefore they will not catch any events from such input devices. In order to ensure that* GLWidget *will receive focus, we change (in* Design *mode) the* **focusPolicy** *property to a policy differing from the default* **Qt::NoFocus***. In our application, we use* **Qt::StrongFocus***, as it allows the widget to receive focus by both tabbing and clicking.*

We also handle mouse events (Listing 22) for manipulating a *virtual trackball*. Appendix D presents the class `TrackBall`. This class implements the methods `mouseMove()`, `mousePress()` and `mouseRelease()` which are called in the event handlers in Listing 22. The only parameter of these trackball class methods is the current mouse position (`event->posF()`). For the `wheelEvent()` we simply implement a zooming operation, where `event->delta()` provides the distance that the mouse wheel is rotated (in eights of a degree).

```
  void GLWidget::mouseMoveEvent(QMouseEvent *event)
2 {
      trackBall.mouseMove(event ->posF());
4 }
6 void GLWidget::mousePressEvent(QMouseEvent *event)
  {
8     if (event ->button() & Qt::LeftButton)
          trackBall.mousePress(event ->posF());
10 }
12 void GLWidget::mouseReleaseEvent(QMouseEvent *event)
  {
14     if (event ->button() == Qt::LeftButton)
          trackBall.mouseRelease(event ->posF());
16 }
18 void GLWidget::wheelEvent(QWheelEvent *event)
  {
20     zoom += 0.001 * event ->delta();
  }
```

Listing 22.   File **glwidget.cpp**: methods for mouse events used by the virtual trackball and for zooming.

J. Matrix and Vector classes

Qt provides a set of classes of vectors (`QVector2D`, `QVector3D`, and `QVector4D`) and matrices (`QMatrix2x2`, `QMatrix2x3`, ..., `QMatrix4x3`, `QMatrix4x4`) to work with geometric transformations and camera settings. Their contents can be bound to shader attributes declared as native data types `vec2`, `vec3`, `vec4`, `mat2`, `mat3`, `mat4` or shader arrays, using methods of the class `QGLShaderProgram`. This binding procedure is shown in Sec. II-M.

In `GLWidget`, we declare two `QMatrix4x4` objects, `modelViewMatrix` and `projectionMatrix` (App. A). In our application, they will replace the deprecated OpenGL matrix modes `GL_MODELVIEW` and `GL_PROJECTION`, respectively. However, we are now under the object-oriented paradigm, instead of the traditional state-machine OpenGL.

In the method `resizeGL()`, in Listing 23, we set up `projectionMatrix`. First we initialize the projection matrix to the identity matrix (equivalent to a call of `glLoadIdentity()` under `glMatrixMode(GL_PROJECTION)`). We then right-multiply itself by the matrix corresponding to the perspective projection, using the method `QMatrix4x4::perspective()`. This method has the same parameters of `gluPerspective()`.

```
void GLWidget::resizeGL(int width, int height)
{
    glViewport(0, 0, width, height);

    projectionMatrix.setToIdentity();
    projectionMatrix.perspective(60.0,
            static_cast<qreal>(width) /
            static_cast<qreal>(height), 0.1, 20.0);

    trackBall.resizeViewport(width, height);

    updateGL();
}
```
Listing 23.   File **glwidget.cpp**: method `resizeGL()`.

In the method `paintGL()` (Listing 27), we set up `modelViewMatrix`. As `projectionMatrix`, we first set it to the identity matrix. After that, we use the method `QMatrix4x4::lookAt()` which, similarly to `gluLookAt()`, creates a viewing matrix derived from the triple: a observer point, a look-at point and an up-vector. We also apply two geometric transformations: first a rotation, provided by the `trackball` object (App. D), and second a translation. Notice that, as the deprecated OpenGL geometric transformations (`glTranslate`/`glRotate`/`glScale`), Qt matrix transformations correspond to right-multiplications.

Obs 6. `QMatrix4x4` *provides other matrix transformation methods,* e.g., *projection matrices* `frustum()`, `ortho()` *and the geometric transformation* `scale()`. *Overloaded operators for matrix-matrix and matrix-vector operations are also provided. Further methods as well as their details are found in the* `QMatrix4x4` *documentation [25].*

K. Signals/Slots – Part 2

So far we learnt how to work with signals/slots using Qt Creator. In this section we show other objects, not directly related to UI, that are also able to intercommunicate using signals/slots.

We exemplify with an instance of the class `QTimer`, `timer`, declared in `GLWidget`. It emits signals to `widget` in order to repaint the OpenGL window during the application's idle time.

Instead of defining the signal/slot connection in *Design* mode, we do that directly in the source code. In Listing 26, Line 12, the Qt command `connect()` connects the pre-defined signal `timeout()` emitted by `timer` to the custom slot `animate()` (Listing 24) of the `widget`. In Line 13 of Listing 26, we start the `timer` with a timeout interval of zero milliseconds, which means that a timeout signal will be emitted whenever the application is idle.

```
void GLWidget::animate()
{
    updateGL();
}
```
Listing 24.   File **glwidget.cpp**: slot method `animate()`.

We also exemplify a custom signal defined in our class `GLWidget` that connects to a pre-defined slot. This signal will be connected to a slot of the status bar that shows on it the number of vertices and faces of the loaded mesh. When the mesh model is successfully loaded in `readOFFFile()` (Listing 14), `widget` emits the `statusBarMessage()` signal. This is done by inserting Lines 4-6 to the end of `readOFFFile()`, as shown in Listing 25. We create the connection in *Design* mode, associating our custom signal `statusBarMessage()` from `widget` to the pre-defined slot `showMessage()` of `statusBar`.

```
void GLWidget::readOFFFile(const QString &fileName)
{
...
    emit statusBarMessage(QString("Samples %1, Faces %2")
                          .arg(numVertices)
                          .arg(numFaces));
}
```
Listing 25.   File **glwidget.cpp**: inserting a signal emitted from the method `readOFFFile()`.

Obs 7. *Signals do not need to be defined, only declared: in this last example we do not define the method* `statusBarMessage()`*; it is only declared in* `GLWidget` *(Listing 30, Lines 22 and 23) using the reserved Qt keyword* `signals`.

L. Texture Mapping

We will use two texture maps: a diffuse map and a normal map [8]. In Listing 26, Lines 5-6, we create the objects `texColor` and `texNormal` of class `QImage` by loading image files from the resource collection file. After that, in Lines 7-10 we generate corresponding 2D GL textures for the first two texture units (`GL_TEXTURE0` and `GL_TEXTURE1`). The method `QGLWidget::bindTexture()` calls the OpenGL command `glGenTextures()`, binds the new texture and returns the texture identifier for future usage.

```
void GLWidget::initializeGL()
{
    glEnable(GL_DEPTH_TEST);

    QImage texColor= QImage(":/textures/bricksDiffuse.png");
    QImage texNormal= QImage(":/textures/bricksNormal.png");
    glActiveTexture(GL_TEXTURE0);
    texID[0] = bindTexture(texColor);
    glActiveTexture(GL_TEXTURE1);
    texID[1] = bindTexture(texNormal);

    connect(&timer,SIGNAL(timeout()), this,SLOT(animate()));
    timer.start(0);
}
```
Listing 26.   File **glwidget.cpp**: method `initializeGL()`.

Obs 8. *The concept of* pixel buffer objects *is another option to work with textures in OpenGL. The class* `QGLBuffer` *supports pixel buffer objects by using the Qt buffer types* `QGLBuffer::PixelPackBuffer` *and* `QGLBuffer::PixelUnpackBuffer`. *One of the advantages of pixel buffer objects is the asynchronous communication between the CPU and the GPU, which is useful, for instance, in applications where the textures need to be changed in run time [22].*

M. The method `paintGL()`

The method `paintGL()` (Listing 27) is called whenever the OpenGL scene must be redisplayed. For each call, `paintGL()` performs the following tasks:

1) The model-view transformations;
2) The binding of the shader program;
3) The uploading of the `uniform` data to the GPU;
4) The binding of the buffer objects and textures to the GPU;
5) The releasing of the buffer objects and the shader program.

The model-view transformations (Lines 8-11) were aforementioned in Sec. II-J. After binding the shader program (Line 13), we upload the matrices `modelViewMatrix` and `projectionMatrix` to the `mat4` uniform shader variables `modelViewMatrix` and `projectionMatrix` (see the vertex shaders in App. C). We also upload the transpose of the inverse of the top-left 3×3 part of `modelViewMatrix` to the `mat3` uniform variable `normaMatrix` (Line 17). This matrix is used to apply the model-view transformation to the vertex normals. In Lines 19-28, we upload the coefficients of the Phong lighting model (we assume only one light source).

The texturing code is shown in Lines 33-37. First we assign the sampler shader variables `texColorMap` and `texNormalMap` to the first two texture units (Lines 33-34). We see in the fragment shader **ftexture.glsl** (Listings 36) the use of `texColorMap`, whereas in the fragment shader **fnormal.glsl** (Listing 38) the use of both `texColorMap` and `texNormalMap`. Finally, in Lines 33-36 we bind both textures to the corresponding texture units. Here we use the native OpenGL command `glBindTexture()`, passing as parameters the texture identifiers that we kept from the last call to `QGLWidget::bindTexture()` when the textures were created (as shown in Sec. II-L, Listing 26).

In Lines 38-53 we bind the different buffer objects and assign them to the corresponding shader program attributes. For instance, in Lines 38-40, `vboVertices` is bound. `shaderProgram` enables the array attribute `vPosition` (see its declaration on the vertex shaders in App. C) and finally, the method `QGLShaderProgram::setAttributeBuffer()` formats `vPosition`, where its parameters are: the type of elements in the vertex array, the starting position in the bound buffer object, the number of per vertex components and the stride between consecutive vertices.

Just before calling the native OpenGL command `glDrawElements()` to draw the mesh, we bind the index buffer in Line 54.

The method `paintGL()` ends with the releasing of all buffer objects as well as of the shader program.

Obs 9. *In our application, the uploading of shader variables and the binding of shader attributes in* `paintGL()` *is done irrespective of the shader effect currently set. This is acceptable because any variables and attributes not referenced in the shader program are simply ignored. For instance, the shader program for rendering a non-textured model with Phong shading does not require texture coordinates or tangent vectors passed as attributes. In such case, the binding of the corresponding buffer objects and textures are ignored.*

Obs 10. *The set up of shader attributes could be done outside* `paintGL()` *just after the model is loaded and the shader program is created, assuming that our application is composed of only one mesh model rendered with only one shader effect. Also, since the contents of our textures do not change along the execution of our application, we could perform the texturing work only once. However, we opt to leave these commands at* `paintGL()` *as it is done in more sophisticate applications with several mesh models, shader effects and textures.*

```cpp
void GLWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    if (!vboVertices)
        return;

    modelViewMatrix.setToIdentity();
    modelViewMatrix.lookAt(camera.eye,camera.at,camera.up);
    modelViewMatrix.translate(0, 0, zoom);
    modelViewMatrix.rotate(trackBall.getRotation());

    shaderProgram->bind();

    shaderProgram->setUniformValue("modelViewMatrix",
        modelViewMatrix);
    shaderProgram->setUniformValue("projectionMatrix",
        projectionMatrix);
    shaderProgram->setUniformValue("normalMatrix",
        modelViewMatrix.normalMatrix());

    QVector4D ambientProduct = light.ambient * material.
        ambient;
    QVector4D diffuseProduct = light.diffuse * material.
        diffuse;
    QVector4D specularProduct = light.specular * material.
        specular;

    shaderProgram->setUniformValue("lightPosition", light.
        position);
    shaderProgram->setUniformValue("ambientProduct",
        ambientProduct);
    shaderProgram->setUniformValue("diffuseProduct",
        diffuseProduct);
    shaderProgram->setUniformValue("specularProduct",
        specularProduct);
    shaderProgram->setUniformValue(
            "shininess", static_cast<GLfloat>(material.
                shininess));

    shaderProgram->setUniformValue("texColorMap", 0);
    shaderProgram->setUniformValue("texNormalMap", 1);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texID[0]);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, texID[1]);

    vboVertices->bind();
    shaderProgram->enableAttributeArray("vPosition");
    shaderProgram->setAttributeBuffer("vPosition", GL_FLOAT,
        0, 4, 0);

    vboNormals->bind();
    shaderProgram->enableAttributeArray("vNormal");
    shaderProgram->setAttributeBuffer("vNormal", GL_FLOAT,
        0, 3, 0);

    vboTexCoords->bind();
    shaderProgram->enableAttributeArray("vTexCoord");
    shaderProgram->setAttributeBuffer("vTexCoord", GL_FLOAT,
        0, 2, 0);

    vboTangents->bind();
    shaderProgram->enableAttributeArray("vTangent");
    shaderProgram->setAttributeBuffer("vTangent", GL_FLOAT,
        0, 4, 0);

    vboIndices->bind();

    glDrawElements(GL_TRIANGLES, numFaces * 3,
        GL_UNSIGNED_INT, 0);

    vboIndices->release();
    vboTangents->release();
    vboTexCoords->release();
    vboNormals->release();
    vboVertices->release();
    shaderProgram->release();
}
```

Listing 27. File **glwidget.cpp**: method `paintGL()`.

N. Formatting the OpenGL context

Qt provides the class `QGLFormat` [26] that allows to specify the display format of an OpenGL context. For instance, it allows to set double or single buffering (double buffering by default), alpha channel (disabled by default), stereo buffers (disabled by default), the color mode (rgba by default), the depth buffer (enabled by default) and antialiasing (disabled by default). The functionalities of this class are similar to those of the function `glutInitDisplayMode()`.

We must format the OpenGL context prior to the creation of `GLWidget`. In our application, we first create a `QGLFormat` object `format` based on the default Qt format (Listing 28, Line 7) and enable the support to antialiasing (Lines 8-9). After that we set `format` to the new default OpenGL context format (Line 12). As can be noticed, this process is done in the `main()` function before the creation of the `MainWindow` object. When the `MainWindow` object is defined, our `GLWidget` automatically creates an OpenGL context using the new default format.

```
#include <QtGui/QApplication>
#include <QGLFormat>
#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QGLFormat format = QGLFormat::defaultFormat();
    format.setSampleBuffers(true);
    format.setSamples(8);
    if (!format.sampleBuffers())
        qWarning("Multisample buffer is not supported.");
    QGLFormat::setDefaultFormat(format);

    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

Listing 28.   File **main.cpp**: setting the default format for the OpenGL context.

III. IMPROVING THE USER INTERFACE

So far the UI of our application is quite simple, as shown in Fig. 13. Qt provides several other UI components to improve the graphical interface of our application [4], [5]. In Fig. 14 we present a richer GUI with several tabs containing different sets of components. On (a) we can select the shader either by keyboard or by a combo box. On (b) we can change the colors of the light and the materials in the Phong lighting model by sliding dials. On (c) we can take a screenshot of the OpenGL scene by pressing a push button (see App. F for the source code of the slot associated to this button), we can set the background color using a `QColorDialog` object, as shown on (d), and we can toggle a check box to display the frame rate as an overlaid text string rendered by the method `QGLWidget::renderText()`.

IV. FINAL CONSIDERATIONS

In this work we present how to develop an interactive graphics application with the Qt framework, OpenGL and GLSL using the Qt SDK. We show how such an application is built with reasonably small effort while keeping coding effectiveness. In our exposition we exploit programmable

Fig. 13.   Simple Qt with OpenGL application.

(a)   (b)

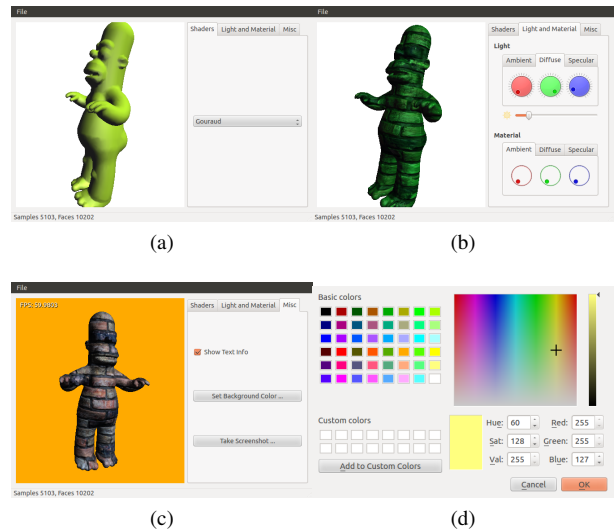(c)   (d)

Fig. 14.   An improved Qt application with OpenGL: several tabs and parameters can be tuned by the UI.

OpenGL pipeline features with Qt, *e.g.*, buffer objects and shader programs. In particular, we do not use OpenGL 3.0 deprecated functions. We employ the Qt object-oriented program framework, allowing to encapsulate the newest OpenGL functionalities while maintaining the source code clean and organized.

We believe that OpenGL with Qt is a reasonable choice not only to develop academic and professional applications, but also to educational purposes as we have successfully applied this combination on both graduate and undergraduate computer graphics courses.

V. DIRECTIONS FOR FURTHER GRAPHICS-BASED QT APPLICATIONS

The power of Qt for interactive graphics applications is not limited to the scope of this presentation which relied on the

Qt OpenGL Module.

For instance, for 2D graphics applications, Qt provides the classes `QGraphicsScene` and `QPainter`, which allow to draw and organize complex 2D vector elements, text fonts and pixmaps [4]. Furthermore, `QPainter` can be combined with `QGLWidget` to render scenes with 2D and 3D graphics elements (see, for instance, *Qt Boxes Demo* [27]).

Qt framework 5.0 will be released soon. This new major release will feature the *Qt3D Module* [28], which implements a scene graph using the Qt Meta Language QML. The scene can be described by nodes that correspond to geometry, material, effects and transformations.

Acknowledgments

APPENDIX A
GLWidget CLASS DECLARATION

```
#ifndef GLWIDGET_H
#define GLWIDGET_H

#include <QtOpenGL>

#include <iostream>
#include <fstream>
#include <limits>

#include "camera.h"
#include "light.h"
#include "material.h"
#include "trackball.h"

class GLWidget : public QGLWidget
{
    Q_OBJECT
public:
    explicit GLWidget(QWidget *parent = 0);
    virtual ~GLWidget();

signals:
    void statusBarMessage(QString ns);

public slots:
    void toggleBackgroundColor(bool toBlack);
    void showFileOpenDialog();
    void animate();

protected:
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
    void mouseMoveEvent(QMouseEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
    void wheelEvent(QWheelEvent *event);
    void keyPressEvent(QKeyEvent *event);

private:
    void readOFFFile(const QString &fileName);
    void genNormals();
    void genTexCoordsCylinder();
    void genTangents();
    void createVBOs();
    void destroyVBOs();
    void createShaders();
    void destroyShaders();

    QPointF pixelPosToViewPos(const QPointF &p);

    unsigned int numVertices;
    unsigned int numFaces;
    QVector4D *vertices;
    QVector3D *normals;
    QVector2D *texCoords;
    QVector4D *tangents;
    unsigned int *indices;

    QGLBuffer *vboVertices;
    QGLBuffer *vboNormals;
```

```
    QGLBuffer *vboTexCoords;
    QGLBuffer *vboTangents;
    QGLBuffer *vboIndices;

    QGLShader *vertexShader;
    QGLShader *fragmentShader;
    QGLShaderProgram *shaderProgram;
    unsigned int currentShader;

    int texID[2];

    QMatrix4x4 modelViewMatrix;
    QMatrix4x4 projectionMatrix;

    Camera camera;
    Light light;
    Material material;

    TrackBall trackBall;

    double zoom;

    QTimer timer;
};
```

Listing 29. Class `GLWidget`.

APPENDIX B
CONSTRUCTOR AND DESTRUCTOR OF GLWIDGET

```
GLWidget::GLWidget(QWidget *parent) :
    QGLWidget(parent)
{
    vertices = NULL;
    normals = NULL;
    texCoords = NULL;
    tangents = NULL;
    indices = NULL;

    vboVertices = NULL;
    vboNormals = NULL;
    vboTexCoords = NULL;
    vboTangents = NULL;
    vboIndices = NULL;

    shaderProgram = NULL;
    vertexShader = NULL;
    fragmentShader = NULL;
    currentShader = 0;

    zoom = 0.0;

    fpsCounter = 0;
}

GLWidget::~GLWidget()
{
    destroyVBOs();
    destroyShaders();
}
```

Listing 30. File **glwidget.cpp**: constructor and destructor of `GLWidget`.

APPENDIX C
GLSL SHADERS

```
attribute vec4 vPosition;
attribute vec3 vNormal;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform mat3 normalMatrix;

uniform vec4 ambientProduct;
uniform vec4 diffuseProduct;
uniform vec4 specularProduct;
uniform float shininess;
uniform vec4 lightPosition;

void main()
{
    vec4 eyePosition = modelViewMatrix * vPosition;
    vec3 N = normalMatrix * vNormal;
    vec3 L = lightPosition.xyz - eyePosition.xyz;
    vec3 E = -eyePosition.xyz;
    vec3 R = reflect(-E, N);

    N = normalize(N);
    L = normalize(L);
    E = normalize(E);
```

```glsl
    float NdotL = dot(N, L);
    float Kd = max(NdotL, 0.0);
    float Ks = (NdotL < 0.0) ? 0.0 : pow(max(dot(R, E), 0.0)
        , shininess);

    vec4 diffuse  = Kd * diffuseProduct;
    vec4 specular = Ks * specularProduct;
    vec4 ambient  = ambientProduct;

    gl_Position = projectionMatrix * eyePosition;
    gl_FrontColor = ambient + diffuse + specular;
    gl_FrontColor.a = 1.0;
}
```

Listing 31. File **vgouraud.glsl**.

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

Listing 32. File **fgouraud.glsl**.

```glsl
attribute vec4 vPosition;
attribute vec3 vNormal;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform mat3 normalMatrix;

uniform vec4 lightPosition;

varying vec3 fN;
varying vec3 fE;
varying vec3 fL;

void main()
{
    vec4 eyePosition = modelViewMatrix * vPosition;

    fN = normalMatrix * vNormal;
    fL = lightPosition.xyz - eyePosition.xyz;
    fE = -eyePosition.xyz;

    gl_Position = projectionMatrix * eyePosition;
}
```

Listing 33. File **vphong.glsl**.

```glsl
varying vec3 fN;
varying vec3 fE;
varying vec3 fL;

uniform vec4 ambientProduct;
uniform vec4 diffuseProduct;
uniform vec4 specularProduct;
uniform float shininess;

void main()
{
    vec3 N = normalize(fN);
    vec3 E = normalize(fE);
    vec3 L = normalize(fL);
    vec3 R = normalize(2.0 * dot(L, N) * N - L);

    float NdotL = dot(N, L);
    float Kd = max(NdotL, 0.0);
    float Ks = (NdotL < 0.0) ? 0.0 : pow(max(dot(R, E), 0.0)
        , shininess);

    vec4 diffuse  = Kd * diffuseProduct;
    vec4 specular = Ks * specularProduct;
    vec4 ambient  = ambientProduct;

    gl_FragColor = ambient + diffuse + specular;
    gl_FragColor.a = 1.0;
}
```

Listing 34. File **fphong.glsl**.

```glsl
attribute vec4 vPosition;
attribute vec3 vNormal;
attribute vec2 vTexCoord;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform mat3 normalMatrix;

uniform vec4 lightPosition;
```

```glsl
varying vec3 fN;
varying vec3 fE;
varying vec3 fL;
varying vec2 fTexCoord;

void main()
{
    vec4 eyePosition = modelViewMatrix * vPosition;

    fN = normalMatrix * vNormal;
    fL = lightPosition.xyz - eyePosition.xyz;
    fE = -eyePosition.xyz;
    fTexCoord = vTexCoord;

    gl_Position = projectionMatrix * eyePosition;
}
```

Listing 35. File **vtexture.glsl**.

```glsl
varying vec3 fN;
varying vec3 fE;
varying vec3 fL;
varying vec2 fTexCoord;

uniform vec4 ambientProduct;
uniform vec4 diffuseProduct;
uniform vec4 specularProduct;
uniform float shininess;

uniform sampler2D texColorMap;

void main()
{
    vec3 N = normalize(fN);
    vec3 E = normalize(fE);
    vec3 L = normalize(fL);
    vec3 R = normalize(2.0 * dot(L, N) * N - L);

    float NdotL = dot(N, L);
    float Kd = max(NdotL, 0.0);
    float Ks = (NdotL < 0.0) ? 0.0 : pow(max(dot(R, E), 0.0)
        , shininess);

    vec4 diffuse  = Kd * diffuseProduct;
    vec4 specular = Ks * specularProduct;
    vec4 ambient  = ambientProduct;

    gl_FragColor = (ambient + diffuse + specular) *
                   texture2D(texColorMap, fTexCoord);
    gl_FragColor.a = 1.0;
}
```

Listing 36. File **ftexture.glsl**.

```glsl
attribute vec4 vPosition;
attribute vec3 vNormal;
attribute vec2 vTexCoord;
attribute vec4 vTangent;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform mat3 normalMatrix;

uniform vec4 lightPosition;

varying vec3 fE;
varying vec3 fL;
varying vec2 fTexCoord;

void main()
{
    vec3 bitangent = vTangent.w * cross(vNormal, vTangent.
        xyz);
    vec3 T = normalMatrix * vTangent.xyz;
    vec3 B = normalMatrix * bitangent;
    vec3 N = normalMatrix * vNormal;

    mat3 TBN = mat3(T.x, B.x, N.x,
                    T.y, B.y, N.y,
                    T.z, B.z, N.z);

    vec4 eyePosition = modelViewMatrix * vPosition;
    fL = TBN * (lightPosition.xyz - eyePosition.xyz);
    fE = TBN * (-eyePosition.xyz);

    fTexCoord = vTexCoord;

    gl_Position = projectionMatrix * eyePosition;
}
```

Listing 37. File **vnormal.glsl**.

```glsl
varying vec3 fE;
varying vec3 fL;
varying vec2 fTexCoord;

uniform vec4 ambientProduct;
uniform vec4 diffuseProduct;
uniform vec4 specularProduct;
uniform float shininess;

uniform sampler2D texColorMap;
uniform sampler2D texNormalMap;

void main()
{
    vec3 N = normalize(texture2D(texNormalMap, fTexCoord).
        rgb * 2.0 - 1.0);
    vec3 E = normalize(fE);
    vec3 L = normalize(fL);
    vec3 R = normalize(2.0 * dot(L, N) * N - L);

    float NdotL = dot(N, L);
    float Kd = max(NdotL, 0.0);
    float Ks = (NdotL < 0.0) ? 0.0 : pow(max(dot(R, E), 0.0)
        , shininess);

    vec4 diffuse  = Kd * diffuseProduct;
    vec4 specular = Ks * specularProduct;
    vec4 ambient  = ambientProduct;

    gl_FragColor = (ambient + diffuse + specular) *
                texture2D(texColorMap, fTexCoord);
    gl_FragColor.a = 1.0;
}
```

Listing 38. File **fnormal.glsl**.

APPENDIX D
VIRTUAL TRACKBALL CLASS

Our implementation of the virtual trackball was based on both the book by Angel and Shreiner [8] and on the *Boxes Demo* available on Qt framework 4.8 [27]. We use the class `QQuaternion` to represent rotations in 3D space using *quaternions*.

```cpp
#ifndef TRACKBALL_H
#define TRACKBALL_H

#include <QVector3D>
#include <QQuaternion>
#include <QTime>

#include <cmath>

class TrackBall
{
public:
    TrackBall();
    void mouseMove(const QPointF& p);
    void mousePress(const QPointF& p);
    void mouseRelease(const QPointF& p);
    void resizeViewport(int width, int height);
    QQuaternion getRotation();

private:
    QQuaternion rotation;
    QVector3D axis;
    double velocity;

    QVector3D lastPos3D;
    QTime lastTime;
    bool trackingMouse;

    double viewportWidth;
    double viewportHeight;

    const double rad2deg;

    QVector3D mousePosTo3D(const QPointF& p);
};

#endif
```

Listing 39. File **trackball.h**.

```cpp
#include "trackball.h"

TrackBall::TrackBall() : rad2deg(180.0 / M_PI)
```

```cpp
{
    velocity = 0.0;
    trackingMouse = false;
    lastTime = QTime::currentTime();
}

void TrackBall::mouseMove(const QPointF &p)
{
    if (!trackingMouse)
        return;

    QTime currentTime = QTime::currentTime();
    int msecs = lastTime.msecsTo(currentTime);
    if (msecs) {
        QVector3D vp = mousePosTo3D(p);
        QVector3D currentPos3D = QVector3D(vp.x(), vp.y(),
            0.0);
        double lenSqr = currentPos3D.lengthSquared();
        (lenSqr >= 1.0) ? currentPos3D.normalize() :
                    currentPos3D.setZ(sqrt(1.0 -
                        lenSqr));

        axis = QVector3D::crossProduct(lastPos3D,
            currentPos3D);
        double angle = rad2deg * axis.length();
        velocity = angle / msecs;
        axis.normalize();
        rotation = QQuaternion::fromAxisAndAngle(axis, angle
            ) * rotation;

        lastPos3D = currentPos3D;
        lastTime = currentTime;
    }
}

void TrackBall::mousePress(const QPointF &p)
{
    rotation = getRotation();
    trackingMouse = true;
    lastTime = QTime::currentTime();

    lastPos3D = mousePosTo3D(p);
    double lenSqr = lastPos3D.lengthSquared();
    (lenSqr >= 1.0) ? lastPos3D.normalize() :
                lastPos3D.setZ(sqrt(1.0 - lenSqr));

    velocity = 0.0;
}

void TrackBall::mouseRelease(const QPointF &p)
{
    mouseMove(p);
    trackingMouse = false;
}

void TrackBall::resizeViewport(int width, int height)
{
    viewportWidth = static_cast<double>(width);
    viewportHeight = static_cast<double>(height);
}

QQuaternion TrackBall::getRotation()
{
    if (trackingMouse)
        return rotation;

    QTime currentTime = QTime::currentTime();
    double angle = velocity * lastTime.msecsTo(currentTime);
    return QQuaternion::fromAxisAndAngle(axis, angle) *
        rotation;
}

QVector3D TrackBall::mousePosTo3D(const QPointF &p)
{
    return QVector3D(2.0 * p.x() / viewportWidth - 1.0,
                1.0 - 2.0 * p.y() / viewportHeight,
                0.0);
}
```

Listing 40. File **trackball.cpp**.

APPENDIX E
AUXILIARY CLASSES FOR PHONG LIGHTING MODEL

```cpp
#ifndef CAMERA_H
#define CAMERA_H

#include <QVector3D>

class Camera
{
public:
```

```
10        Camera();

          QVector3D eye;
12        QVector3D at;
          QVector3D up;
14    };
      #endif // CAMERA_H
```

Listing 41.   File **camera.h**.

```
1     #include "camera.h"

3     Camera::Camera()
      {
5         eye = QVector3D(0.0, 0.0, 1.0);
          at  = QVector3D(0.0, 0.0, 0.0);
7         up  = QVector3D(0.0, 1.0, 0.0);
      }
```

Listing 42.   File **camera.cpp**

```
      #ifndef LIGHT_H
2     #define LIGHT_H

4     #include <QVector4D>

6     class Light
      {
8     public:
          Light();
10
          QVector4D position;
12        QVector4D ambient;
          QVector4D diffuse;
14        QVector4D specular;
      };
16
      #endif // LIGHT_H
```

Listing 43.   File **light.h**.

```
1     #include "light.h"

3     Light::Light()
      {
5         position = QVector4D(3.0, 3.0, 3.0, 0.0);
          ambient  = QVector4D(0.1, 0.1, 0.1, 1.0);
7         diffuse  = QVector4D(0.9, 0.9, 0.9, 1.0);
          specular = QVector4D(0.9, 0.9, 0.9, 1.0);
9     }
```

Listing 44.   File **light.cpp**.

```
1     #ifndef MATERIAL_H
      #define MATERIAL_H
3
      #include <QVector4D>
5
      class Material
7     {
      public:
9         Material();

11        QVector4D ambient;
          QVector4D diffuse;
13        QVector4D specular;
          double shininess;
15    };

17    #endif // MATERIAL_H
```

Listing 45.   File **material.h**.

```
1     #include "material.h"

3     Material::Material()
      {
5         ambient   = QVector4D(1.0, 1.0, 1.0, 1.0);
          diffuse   = QVector4D(0.6, 0.6, 0.6, 1.0);
7         specular  = QVector4D(0.4, 0.4, 0.4, 1.0);
          shininess = 200.0;
9     }
```

Listing 46.   File **material.cpp**.

APPENDIX F
TAKING A SCREENSHOT OF THE OPENGL SCENE

Listing 47 presents the slot method that takes a screenshot of the OpenGL Scene. The method `QGLWidget::grabFrameBuffer()` returns the contents of the frame buffer as a `QImage` object, which is saved to disk using the method `QImage::save()`.

```
1     void GLWidget::takeScreenshot()
      {
3         QImage screenshot = grabFrameBuffer();

5         QString fileName;
          fileName = QFileDialog::getSaveFileName(this,
7                 "Save File As", QDir::homePath(),
                  QString("PNG Files (*.png)"));
9         if (fileName.length()) {
              if (!fileName.contains(".png"))
11                fileName += ".png";
              if (screenshot.save(fileName, "PNG")) {
13                QMessageBox::information(this, "Screenshot",
                          "Screenshot taken!",
15                        QMessageBox::Ok);
              }
17        }
      }
```

Listing 47.   Slot responsible to take a screenshot of the OpenGL scene.

REFERENCES

[1] "OpenGL – The Industry Standard for High Performance Graphics," 2012. [Online]. Available: http://www.opengl.org/

[2] "GLUT – The OpenGL Utility Toolkit," 2012. [Online]. Available: http://www.opengl.org/resources/libraries/glut/

[3] Qt, "Qt–Cross-Platform application and UI framework," 2012. [Online]. Available: http://qt.nokia.com/

[4] J. Blanchette and M. Summerfield, *C++ GUI Programming with Qt 4*, 2nd ed.   Prentice Hall, 2008.

[5] M. Summerfield, *Advanced Qt Programming: Creating Great Software with C++ and Qt 4*.   Prentice Hall, 2010.

[6] Qt, "Qt in use," 2012. [Online]. Available: http://qt.nokia.com/qt-in-use

[7] ——, "QtOpenGL Module," 2012. [Online]. Available: http://qt-project.org/doc/qt-4.8/qtopengl.html

[8] E. Angel and D. Shreiner, *Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL*, 6th ed.   Addison Wesley, 2011.

[9] M. P. B. Donald D Hearn and W. Carithers, *Computer Graphics with OpenGL*, 4th ed.   Prentice Hall, 2010.

[10] D. Wolff, *OpenGL 4.0 Shading Language Cookbook*.   Pactk Publishing, 2011.

[11] R. Marroquim and A. Maximo, "Introduction to GPU Programming with GLSL," in *Computer Graphics and Image Processing (SIBGRAPI TUTORIALS), 2009 Tutorials of the XXII Brazilian Symposium on*, oct. 2009, pp. 3 –16.

[12] Qt, "Developing Qt," 2012. [Online]. Available: http://qt-project.org/wiki/Category:Developing_Qt

[13] ——, "Qmake Variable Reference," 2012. [Online]. Available: http://qt-project.org/doc/qt-4.8/qmake-variable-reference.html

[14] ——, "QGLWidget Class Reference," 2012. [Online]. Available: http://qt-project.org/doc/qt-4.8/QGLWidget.html

[15] ——, "Layout Management," 2012. [Online]. Available: http://qt-project.org/doc/qt-4.8/layout.html

[16] ——, "Signals & Slots," 2012. [Online]. Available: http://qt-project.org/doc/qt-4.8/signalsandslots.html

[17] "OFF File Format," 2012. [Online]. Available: http://www.geomview.org/docs/html/OFF.html

[18] E. Lengyel, *Mathematics for 3D Game Programming and Computer Graphics*, 3rd ed.   Course Technology PTR, 2011.

[19] ——, "Computing Tangent Space Basis Vectors for an Arbitrary Mesh. Terathon Software 3D Graphics Library," 2001. [Online]. Available: http://www.terathon.com/code/tangent.html

[20] Qt, "The Qt Resource System," 2012. [Online]. Available: http://qt-project.org/doc/qt-4.8/resources.html

[21] ——, "QFile Class Reference," 2012. [Online]. Available: http://doc-snapshot.qt-project.org/4.8/qfile.html

[22] M. J. Kilgard, "Modern OpenGL usage: using vertex buffer objects well," in *ACM SIGGRAPH ASIA 2008 courses*, ser. SIGGRAPH Asia '08. New York, NY, USA: ACM, 2008, pp. 49:1–49:19. [Online]. Available: http://doi.acm.org/10.1145/1508044.1508093

[23] Qt, "QGLBuffer Class Reference," 2012. [Online]. Available: http://qt-project.org/doc/qt-4.8/qglbuffer.html

[24] ——, "The Event System," 2012. [Online]. Available: http://qt-project.org/doc/qt-4.8/eventsandfilters.html

[25] ——, "QMatrix4x4 Class Reference," 2012. [Online]. Available: http://qt-project.org/doc/qt-4.8/qmatrix4x4.html

[26] ——, "QGLFormat Class Reference," 2012. [Online]. Available: http://qt-project.org/doc/qt-4.8/QGLFormat.html

[27] ——, "Boxes | Documentation | Qt Developer Network," 2012. [Online]. Available: http://qt-project.org/doc/qt-4.8/demos-boxes.html

[28] ——, "Qt3D Module," 2012. [Online]. Available: http://doc-snapshot.qt-project.org/5.0/qt3d-reference.html