Karlsruhe Institute of Technology
Computer Graphics Group

# Graphics Programming Lab

# Hello Cube!

## Abstract

The first assignment of the graphics programming lab introduces you to the development of basic interactive 3D applications, using *OpenGL* and the *Qt application framework*. You will learn how to render and transform simple geometry and how to implement a simple 3D user interaction interface. Eventually, you will write your own GPU shaders using *GLSL*, the *OpenGL shading language*.

## 1 Introduction

Let's have a quick look at the technologies used for this assignment. A short description of the *Hello Cube* application follows.

### 1.1 OpenGL

OpenGL is a powerful cross-platform API for graphics programming. It is the standard de facto in the industry. You will use OpenGL for everything related to drawing 3D graphics.

### 1.2 Qt Application Framework

Qt is a cross-platform application and UI framework. It is widely used for the development of GUI programs and provides a wide set of widgets. In addition, Qt provides a tight integration with OpenGL.

Qt's framework is particularly helpful to implement interactive systems, as it provides integrated key routines for handling user input, for example, widget, mouse, and keyboard events.

### 1.3 GLSL

GLSL, the OpenGL Shading Language, is a specific language for writing shaders. These are small programs which are executed directly on the GPU.

### 1.4 The Hello Cube Application

In this assignment, you are asked to program a small application called "Hello Cube". It features common elements like a menu bar, a tool bar, a main rendering widget, and a status bar.

In the main rendering widget, your application should display a simple colored cube. The user should be able to rotate and translate the cube using the mouse. Additionally, menu items should allow to switch between different shading modes.

## 2 First Steps

This assignment sheet guides you through its subtasks step by step. Eventually, you should end up with the described application.

First of all, you have to set up your programming environment. In case you accomplished all exercises of our computer graphics introductory lecture, you should already have all necessary OpenGL development packages installed. If not, catch up on everything.

Feel free to use your favourite operating system: Windows, Linux, or Mac OS X, it's your choice. Both OpenGL and Qt are cross-platform. However, please ensure not to use excessive OS-specific features; this helps us a lot when we are to compile your solution.

### 2.1 Install Qt Creator

Download *Qt Creator* from the Qt website [3]. Current Linux distributions most likely feature it already as a package, for example, `qtcreator` in Ubuntu, any version is fine down to 2.0.1.

Note that you can use any development environment of your own choice. Qt features integration into Eclipse and also provides a Visual Studio add-in. For purists, any text editor will do. However, we encourage you to give Qt Creator a try. It is nicely written, features code completion, and offers emacs-style key bindings.

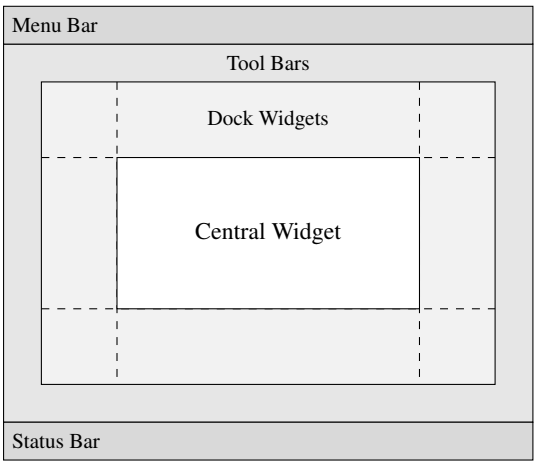
### 2.2 CG Submission System Registration

To submit your code use ILIAS submission system. To upload multiple files just zip them, multiple files upload is not well supported.

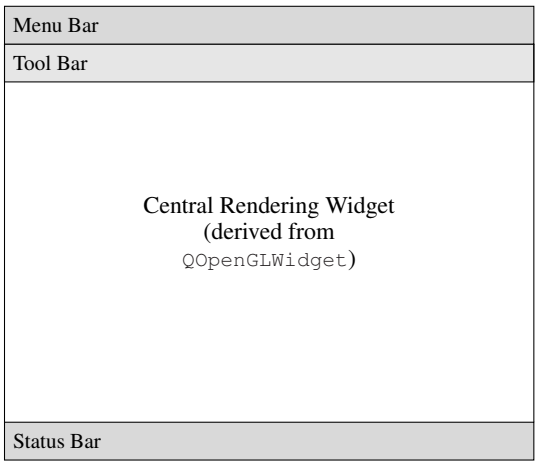
*Always use our Submission System for uploading your solutions.* Log in and sign up for the graphics programming lab. Note that there are strict deadlines for each assignment and the system won't let you upload anything any more after the deadline has passed. If this happens to you, contact us directly. Be aware that there is a *late penalty* of 2 points plus additional 2 points per day behind schedule (including the first day)!

## 3 Create Your GUI with Qt

Let's begin. Create a new project in Qt Creator and select *Qt GUI Application*. Name it "hellocube". Do not generate a UI form! You are asked to code your GUI from scratch; it is important to do so at least once to understand what is going on behind the scenes. You will see a couple of files: `hellocube.pro` defines everything the *qmake* tool has to know in order to create a makefile; `main.cpp` runs the actual application by creating and displaying the *main window* which is defined in `mainwindow.cpp` and `mainwindow.h`.




(a) The `QMainWindow` layout,



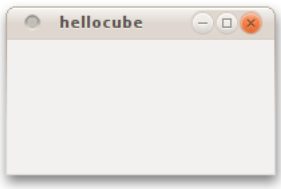

(b) The Hello Cube layout.

**Figure 1:** *Default Qt main window layout and our Hello Cube derivation. See the Qt reference documentation for the* `QMainWindow` *class.*

If your project contains a `.ui` file, you generated a UI form: remove it from the project and code your GUI manually!

Have a look at `main.cpp` and note how it differs from an ordinary console application. Here, we are not simply taking input data, processing it, and returning the result to the user. Instead, the *event loop* is executed, which waits for user input to be processed. Subsequently, the application does not terminate, but waits for further input again.

Compile and run your program. It is an empty main window yet:




However, Qt was already taking care of a lot of things for you. Consider that your code runs on a variety of platforms which differ significantly when it comes to window handling.

## 3.1 Basic Layout

Have a look at Figure 1. It shows the canonical layout of a Qt application; see the reference manual for `QMainWindow` for more information. We are not going to use dock widgets in this assignment, and the status bar is not yet useful. However, you can output information on the status bar if you wish to do so.

You can set custom window titles – the default is simply the project name. Add `setWindowTitle("Hello Cube");` to the initialization of your main window.

Now, get to grips with your user interface.

## 3.2 The Menu Bar

Note the include in `mainwindow.h`:

```
#include <QMainWindow>
```

In order to add further widgets to your form, you need to include their corresponding headers. However, you can simply change the above line to:

```
#include <QtWidgets>
```

Then, you don't have to worry about this issue any more. Note that some GUI systems (like MacOS X and some configurations of Ubuntu Linux) display the menu bar on the very top bar of the screen, instead of your window. Let's learn to include the menu bar with its only menu "File" to your form.

### 3.2.1 The File Menu

Add the following to the private section of your main window class:

```
QMenuBar *menuBar;

QMenu *fileMenu;

QAction *exitAction;
```

Obviously, `menuBar` is a pointer to the menu bar of the main window, and `fileMenu` is a pointer to the "File" menu. We will also set up a `QAction` called `exitAction`. First of all, read the "detailed description" of `QAction` in the Qt documentation: it is important to understand the concept. Most probably, you already figured out that we will use `exitAction` to enable the user to quit the application — in the way it is supposed, not having to exploit the typical "close window" button of the window.

In the main window constructor, add:

```
menuBar = new QMenuBar();
fileMenu = new QMenu("&File");
exitAction = new QAction("E&xit", fileMenu);

exitAction->setShortcut(QKeySequence::Quit);
fileMenu->addAction(exitAction);
menuBar->addMenu(fileMenu);

setMenuBar(menuBar);
```

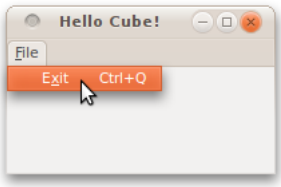
By adding an ampersand symbol into the caption of a menu element, the subsequent character is used for a shortcut. For instance,

setting the “File” menu’s caption to `"&File"` will display the menu as “File”, and typically enables the user to access it by pressing `Alt+F`.

Note that things are different on the Mac, and especially the menu behaves differently. Consult the reference documentation of `QMenu` for details. As a programmer, you don’t have to take care of it though. Qt and the operating system will manage menus, shortcuts, layouts, etc.

There are many ways to specify shortcuts for actions with overloaded functions. Using `QKeySequence::Quit` is the obvious way to enable the user to press `"Ctrl+Q"` (on the Mac `"Command+Q"`) to quit the application. Alternatively, Qt can interpret strings where you specify shortcuts as well.

Now run your program and enjoy your first menu:




Unfortunately, nothing happens when the user clicks on the “Exit” menu element. Add the following line to your constructor:

```
connect(exitAction, SIGNAL(triggered()),
        this, SLOT(close()));
```

Now, it should work as expected. If you are not yet familiar with Qt’s *signals and slots*, then this is a good opportunity to read the reference documentation. In a nutshell, Qt widgets emit *signals* when anything of interest happens. Here, `exitAction` emits its signal `triggered` after the user clicked on its corresponding menu element or pressed its shortcut. We *connect* these signals to *slots* for processing events. The signals and slots mechanism is basically an extension of the C++ standard. (Behind the scenes, the *meta object compiler* translates everything to plain callback functions.) A `QMainWindow` already features a slot called `close`. In our case, we connect the `triggered` signal of `exitAction` to the `close` slot of our main window.

We encourage you to use your own signals and slots when coding Qt applications! It’s powerful, convenient, and secure; you are less likely to screw things up.

#### 3.2.2 The Shading Menu

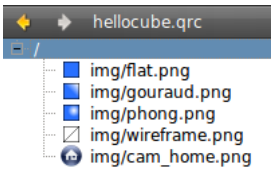
Add a “Shading” menu to the menu bar. Add the following items:

- None (Wireframe) *(Shortcut: `Ctrl+1`)*,
- Flat *(`Ctrl+2`)*,
- Gouraud *(`Ctrl+3`)*,
- Phong *(`Ctrl+4`)*.

Using this menu, the shading mode for the cube will be selected. Therefore, the user most probably expects that a single shading mode is currently selected. Use `setCheckable(true)` to make the menu elements checkable. Unfortunately, every element can now be selected individually, but they should be mutually exclusive. To solve this problem, create a `QActionGroup` and add these elements to this group! Then, set *flat shading* to “checked”. It should be the initial state after the program started.

Create some fancy icons for your actions or use the icons we provided you. Create a new resource file in Qt Creator by clicking “File”, “New File or Project...”, “Files and Classes”, “Qt”, “Qt Resource file”.

Find your new resource file `hellocube.qrc` in the “Resources” directory. Add your icons to this file:




Here, the prefix of the file group is `"/"`, and the icons are in the `img` folder. Now, you can set the icon for the wireframe action:

```
setIcon(QIcon(":/img/wireframe.png"));
```

Starting with `:/` tells Qt to access the internal resources instead of the file system. Add the remaining three icons to the shading actions.

#### 3.2.3 The About Message Box

Add an “About” action to the menu bar. In the main window class, you have to create your own slot:

```
public slots:
    void showAboutBox();
```

Implement this slot as an ordinary C++ method in `mainwindow.cpp` and use a message box to display your name:

```
void MainWindow::showAboutBox()
{
    QMessageBox msgBox;
    msgBox.setWindowTitle("About Hello Cube!");
    msgBox.setText("Written by 1337 H4Xx0r!");
    msgBox.exec();
}
```

As you can see, implementing your own slots is not difficult at all. Neither is implementing your own signals. Don’t forget to connect your ”About” action to the slot you just created!

### 3.3 Provide a Tool Bar

Create a `QToolBar` and add it to your main form. Add the four shading actions to this tool bar as well. Note the benefit of Qt’s actions: no matter whether the user clicks on the menu elements, on the tool bar, or uses shortcuts – all states are consistent and synchronized, and the user can always rely on the same behavior.

### 3.4 Add the Status Bar

Add an empty `QStatusBar` to your application. We are not going to use it, but you may provide additional information to the user.

## 4 Derive Your GLWidget

Derive your own rendering widget from `QOpenGLWidget`. See Qt’s “Hello GL” example [2] on how to accomplish this. In your project file `hellocube.pro`, you will encounter this line:

```
QT       += core gui
```

Append `opengl` in order to link your program correctly:

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

If everything compiles smoothly, use `setCentralWidget()` in `mainwindow.cpp` to set your derived OpenGL widget as your main rendering widget!

For implementing this assignment you only need to call the OpenGL functions, which are declared in `<GL/gl.h>`, which is already included by the `QOpenGLWidget`. However, in the following assignments you will need some OpenGL extension functions, which are not straightforward to use. OpenGL provides a mechanism to query function pointers to these function if they are supported by your graphics device. In order to get around the tedious procedure of querying function pointers one can use libraries that initialize those pointers, so you can directly call the extension functions. One option is to use `GLEW`. In this case we strongly recommend to download the latest version of the library's source and compile it on your own and not rely on your environment's installed library version. Another option is to include `qopenglfunctions_4_5_core.h`, which is provided by the QT environment. Note that both libraries has to be initialized before calling the extension functions.

## 4.1 Core Functionality

Implement these **`protected`** methods for the core functionality of your OpenGL widget:

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

### 4.1.1 Rendering

The initial state of the OpenGL state machine is defined in `initializeGL()`. Furthermore, you can set all OpenGL parameters that will not change during program execution here as well, for example, the background color.

Enable *lighting* and an OpenGL light in `initializeGL()`, and set the position of the light source to $(0.5, 0.0, 2.0)$. (Hint: `glLightfv()`).

After the size of the widget has changed, `resizeGL()` is called. Implement this function to set up both the viewport and the projection matrix to reflect the widget's size! Hint: use `glViewport()`, `gluPerspective()`. Use a *field of view* of 45.0 degrees and some small but positive value for near z plane.

The fundamental method is `paintGL()`: its task is to draw the scene. Therefore, it is called each time the window is going to be refreshed. Implement your rendering here. For example, this code renders a quadrilateral:

```
glBegin(GL_QUADS);

glVertex3f( 1.0,  1.0, 0.0);
glVertex3f(-1.0,  1.0, 0.0);
glVertex3f(-1.0, -1.0, 0.0);
glVertex3f( 1.0, -1.0, 0.0);

glEnd();
```

If you are not familiar with this, consult your favorite OpenGL tutorial or the slides of our basic lecture.

Implement the rendering of a unit cube! It has the vertices $(\pm 0.5, \pm 0.5, \pm 0.5)$. Color each face with a distinct color. For instance, use RGB for the faces intersected by the positive X-, Y-, and Z-axes, and CMY for the faces intersected by the negative axes, respectively.

Implement the wireframe, flat, and Gouraud shading modes. For this, create corresponding slots in your OpenGL widgets. Then, connect the shading actions of the main window to these slots. Within the slots, set the corresponding OpenGL states. Hint: use `glPolygonMode()`, `glShadeModel()` and `GL_WIREFRAME` for wireframe mode, set up material parameters (diffuse and specular color, shininess) with `glMaterial`.

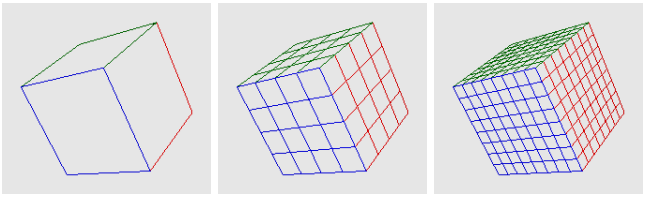
You know that Phong shading is not a part of the fixed-function pipeline of OpenGL. Leave the corresponding slot empty for now; you will implement GLSL shader for this task later on.

Add a `QSlider` to your tool bar which will be used to control the tessellation of the cube. (Hint: use `setSizePolicy()` if you think that its extent is too large.) Set the range to meaningful values. Add a slot to your OpenGL widget to receive changes for the tessellation value:

```
void setTessellation(int t);
```

Connect this slot to the `valueChanged(int)` signal of your tessellation slider. Note that the number of parameters and their types of a signal and a slot have to match exactly! Here, both the signal and the slot feature a single integer parameter. After the user changes the value of the tessellation slider, you can fetch and store this value within your `setTessellation(int)` slot.

To react on user input that changes the state of the current scene, always call `updateGL()` to refresh the displayed scene. Never call `paintGL()` directly! That is, call `updateGL()` after setting your new tessellation value; then, implement tessellation for your cube:



The higher the value of the slider, the higher the tessellation of the cube. Come up with a meaningful mapping.

### 4.1.2 User Input Handling

By dragging the left mouse button, the user should be able to rotate the cube. Moreover, the cube should be translated with the right mouse button. Eventually, using the mouse wheel, the user can zoom in and out.

Zooming is probably the easiest to implement. Catch the `wheelEvent` and process `event->delta()`. Then, use `glTranslatef()` before drawing your cube for zooming.

Translating requires a little bit more effort. Within `mousePressEvent()`, store the position where the user clicked inside the OpenGL widget. On subsequent `mouseMoveEvent()` calls, evaluate the distance of the mouse cursor to its former position from the previous event. Set translation variables accordingly. Only apply translations when the right mouse button is pressed!

Rotating the cube using a virtual trackball is not as straightforward. Consult the *OpenGL wiki* [5] for the theory behind the virtual track-

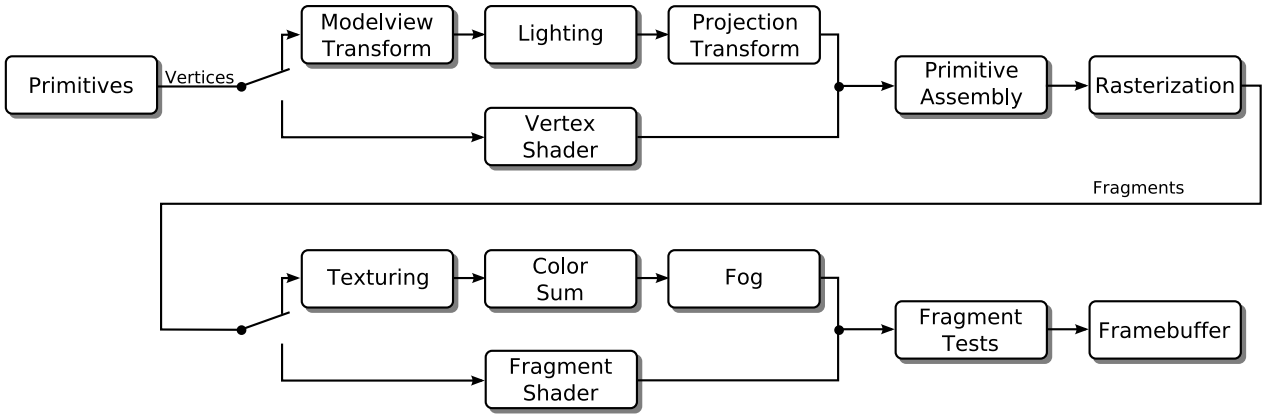



**Figure 2:** *A brief description of the OpenGL pipeline and which parts are bypassed by shaders.*

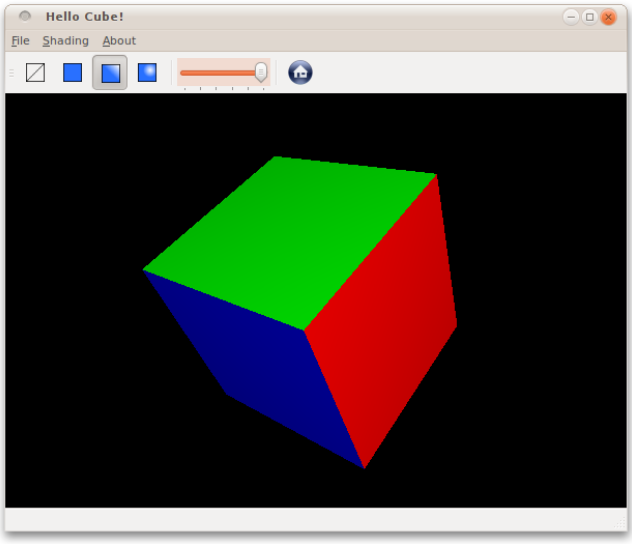
ball. In `mouseMoveEvent`(), map the mouse movement to a meaningful rotation by implementing a virtual trackball.

Use a `QQuaternion` to store the rotation information inside a quaternion in order to avoid *gimbal locks*. Before rendering, convert your quaternion to a `QMatrix4x4`. Qt already provides the "rotate" function for `QMatrix4x4` that accepts a quaternion. Use `glMultMatrixf`() with the matrix elements to rotate your cube accordingly.

Consult the "Boxes" demo [1] for reference implementations of trackballs. You will find other trackball implementations on the web as well. Do not simply copy and paste code to your project! It is important that you understand what a quaternion rotation is and that you implemented a trackball at least once. If we get the impression that you cheated by copy-pasting external code, or took code from your fellow students, we will question you about your solution!

Finally, add a "Reset Camera" action to your tool bar. Implement a corresponding slot, and reset the rotation, translation, and zoom values there. Do not forget to call `updateGL`() afterward, as always.

By now, you should have a nice program running:




Phong shading is the only thing still missing—let's learn how to implement it in the final section.

## 5 Write Your GLSL Shaders

In the previous tasks, all vertices were transformed with the help of the *fixed-function pipeline*. We will replace parts of this pipeline by implementing small self-written shaders, both a vertex and a fragment shader. (We will not cover geometry or tessellation shaders in this assignment.) Figure 2 shows a sketch of the OpenGL pipeline. As you can see, all transformations of the vertices are bypassed when a vertex shader is used.

You will find a large body of GLSL tutorials and introductions in the Internet. Be sure to consult the *OpenGL Shading Language Specification* [4] as needed!

### 5.1 The Vertex Shader

Qt 4.6 introduced the `QGLShader` and `QGLShaderProgram` classes, which will make your task much more enjoyable. It is much easier to compile shaders and to pass variables to them now!

Create a new `QGLShaderProgram` and add a vertex shader to it. Apply the *model-view-projection matrix* to the vertices, and the *normal matrix* to the normals in this shader. Moreover, add a fragment shader to your shader program. Simply set the color of each fragment to, say, white.

Link your shader program using its `link`() method. Recall that you might already have set up an empty slot which you connected to the "triggered" signal of the "Phong" shading action. In this slot, bind the shader program using its `bind`() method. In all other slots, release the shader program using `release`().

You can bypass tessellation if your shader program is bound, as Phong shading will not further increase in quality. It is sufficient to render six single quads then, the cube's faces.

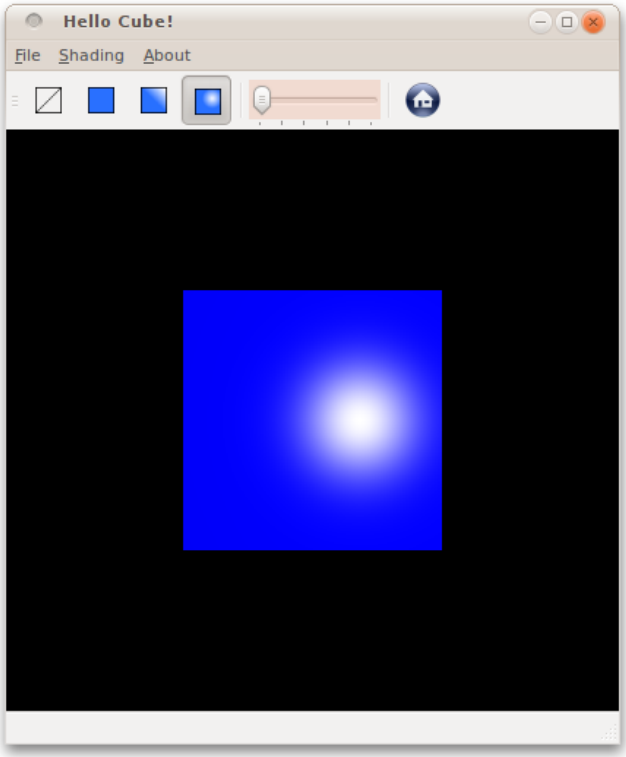
Now, after switching to Phong shading, you should see the silhouette of the cube, filled entirely white. Be sure to pass the correct vertex positions and normals to the fragment shader.

### 5.2 The Fragment Shader

Your last task in this assignment is to complete the fragment shader by implementing Phong shading. Use the same face colors as in the flat and Gouraud shading modes. Moreover, use the same position, diffuse and specular color of the light source. Do not forget to add a nice specular highlight, which is impossible to render using the fixed-function pipeline, unless a very high tessellation is exploited.

Also note that the normal passed to the fragment shader have to be normalized for proper shading computation.

Eventually, your application should produce a result like this:




If this is your first interactive 3D application you developed—*congratulations!*

## References

[1] Digia. Boxes demo. `http://doc.qt.io/qt-5/qtwidgets-graphicsview-boxes-example.html`.

[2] Digia. Hello GL example. `http://doc.qt.io/qt-5/qtopengl-hellogl2-example.html`.

[3] Digia. Qt - A cross-platform application and UI framework. `http://qt.io/`.

[4] OpenGL. GLSL specification. `http://www.opengl.org/documentation/glsl`.

[5] OpenGL wiki. Trackball. `http://www.opengl.org/wiki/Trackball`.

## 6 Grading

You can achieve a total of 20 points in this assignment, based on the following criteria:

| | |
|---|---|
| 1 p | Your “File” menu contains an “Exit” element that works as expected, having a canonical shortcut. |
| 1 p | Your “Shading menu” features all described shading modes. The elements possess individual shortcuts. |
| 1 p | Your tool bar contains the same shading actions. They feature individual icons. |
| 1 p | You added an “About” message box and an (empty) status bar. |
| 1 p | There is a strong consistency between the selected shading mode in the menu and the tool bar, and their shortcuts work as expected. The shading modes are mutually exclusive. The shading mode is correctly reflected in your OpenGL widget. The application starts with flat shading. |
| 1 p | The wireframe mode is implemented correctly and the edges are colored. |
| 1 p | Flat shading is implemented correctly and the cube features distinct colors on its faces. |
| 1 p | Gouraud shading is implemented correctly. |
| 1 p | Your tessellation slider is added to the tool bar. You came up with a meaningful mapping. Its status is correctly reflected in your OpenGL widget, that is, the cube is tessellated accordingly. |
| 1 p | The user can zoom in and out using the mouse wheel. |
| 1 p | The user can translate the cube using the right mouse button. |
| 3 p | You implemented a virtual trackball. The user can rotate the cube using the left mouse button. The mapping between mouse gestures and rotation interpolation is meaningful and comprehensible. You store rotations in a quaternion. |
| 1 p | Your tool bar features a “Camera Reset” button. By pressing it, the camera resets to the same default values for translation, rotation, and zoom that were initialized during application startup. |
| 1 p | Activating Phong shading consistently binds your shader program. Activating all other shading modes consistently releases this shader program. |
| 2 p | Your vertex shader correctly applies the model-view-projection and normal matrices. It passes on processed vertices and normals to the fragment shader. |
| 2 p | You implemented Phong shading in the fragment shader. It features a nice specular highlight. |

You get a *malus* if your code is not nicely written using meaningful variable names in proper English, or significantly lacks proper comments! Moreover, you get a malus if your solution produces errors or way too many warnings!

If we got the impression that you handed in copy-pasted code, and you are unable to explain your solution, we will expel you from the course!