1 Assignment: Hello Cube³!

Daniel Kauker       Alexandros Panagiotidis       Michael Krone

Institut für Visualisierung und Interaktive Systeme

1 Topics

- Basics of a C++-Application with GUI
- Using the Qt Widget library
- 3D Graphics with OpenGL
- Coordinate systems
- Simple geometry
- Transformations
- Introduction into the GLSL Shading Language

2 Introduction

The first assignment of the Fachpraktikum Graphics-Programming of the institute Visualisierung und Interaktive Systeme (short: FaPra) will teach you how to program a 3D graphics application. The X11 window system is used for the presentation of the graphical user interface. The application features common elements like a menu bar, a tool bar, a drawing area, and a status line.

In the drawing area our application displays a simple colored cube. The user is able to modify the scene by rotating, scaling, or translating the cube with the mouse. Additionally, menu items allow to switch between different interaction modes.

If you want to find out how the final version of your program should look like change into the directory /proj/fapra/examples and execute ./HelloCube3 ( ./HelloCube3-x64 on 64 bit machines).

3 First Steps

This document guides you through the tasks step by step resulting in the described application at the end. We start with the program skeleton. This means, you get a running program missing some functionality that has to be implemented by yourself.

We use Subversion (SVN) to manage the source code of this course. Every student gets an own working copy. You can find some slides that explain how to obtain the source skeleton of the first assignment on the course web page. The grading is based on the latest submitted version that is stored on the server before the deadline expired.

Before getting started with QT you have to set up the environment variable for the QT directory. You can do this with either setenv QTDIR /usr/lib/qt3 in tcshell or export QTDIR=/usr/lib/qt3 in sh/bash. Afterward you can compile the application with the command make in the source directory. If everything worked out fine, you should find the executable HelloCube3 in the source directory. Start it!

A new application window opens. It is composed of a menu bar, a tool bar, and a graphics area. A coordinate cross is drawn at the center of the graphics window. The cube and the interaction methods are yet missing. The menu bar contains only a single entry. It will be your task to implement the missing functionality of the user interface.

Now we look at the implementation. The program is designed in a strict object oriented way. All source code files contain classes. Each class has to be defined by two files <class>.C and <class>.H. The declaration of the class is contained in the .H-File. The other file contains its implementation. You have to keep to this structure.

The files and classes of the program skeleton are presented in the following table:

<table>
<thead>
<tr>
<th>File</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.C</td>
<td>-</td>
<td>Program Start</td>
</tr>
<tr>
<td>AppWin.qt.C</td>
<td>AppWin</td>
<td>Application Window</td>
</tr>
<tr>
<td>OGLFrame.qt.C</td>
<td>OGLFrame</td>
<td>Drawing Area</td>
</tr>
<tr>
<td>OptionsDlg.qt.C</td>
<td>OptionsDlg</td>
<td>Options Dialog</td>
</tr>
</tbody>
</table>

4 GUI Programming with Qt

The execution of the program starts in main.C. There, the main object of the GUI-application, which is an object of the class QApplication, is created. Names of Qt classes always start with a Q. The file extension of source code files containing a class derived from a Qt-class has to be .qt.H or .qt.C. Otherwise, the provided Makefile will not work.

4.1 The Application Object

The object QApplication encapsulates the difficult communication with the underlying window system. It also manages the communication with input devices like the keyboard or the mouse. Another advantage of Qt is its platform independence. The same code can be compiled and run on a UNIX machine as well as on a MS-Windows system.

On creation the QApplication object gets all command line parameters stored in the parameters argv and argc. Qt-specific items are filtered and an array containing the parameters not known to Qt is returned. The array may be used for your own program parameters.

The only information needed by the application object is the main window to be used. A main window owns a menu bar and a main widget for displaying the application data. Optionally it features a tool bar and a status line. As most GUI-Applications have a similar basic appearance, Qt provides the QMainWindow class, which offers all standard elements. The programmer adds the desired functionality. This is done by deriving a class from QMainWindow and implementing the functionality in your own methods and data elements. Our derived class is named AppWin. Later, we will examine this class in more detail.

In main.C an instance of the class AppWin is created and the application is told to use it as its main window. After that the window size is specified and the window is displayed. The last command of the main-function calls the Event Loop of the application. The application never returns from this function.
4.2 Event Processing

To understand the concept of event-driven programming, it is necessary to understand that the control flow of GUI-application differs completely from sequential code. A „usual“ program without a graphical user interface gets its input by means of the command line at the start of the program. Then, data is processed by a number of algorithms and finally the output is presented to the user. After the program ends.

The behavior of interactive application completely differs. The program starts with an initial state, which is displayed by the graphical interface. The interface allows the user to activate certain commands and actions by producing input events with the mouse or the keyboard. The application responds to the input, calculates a result, displays the result, and then waits for further input events. For this purpose the Event Loop waits for user input and initiates further processing. This basic mechanism is implemented in the application object of Qt. We just call the correct method and terminate the program after the method returns.

4.3 Widgets

All visible components of Qt’s user interface are derived from the common base class QWidget. A widget (short for window and gadget) is a visible element of the GUI. The main window is also called a top level widget. It contains other widgets like the menu bar or the status line. Qt provides a large number of widgets. Each button, menu element or text field is a widget. By deriving a class from a widget class, its functionality can be extended. Therefore, the desired properties of the new widget are implemented in the derived class.

The event processing of a Qt-application is based on the widget concept. User input activates certain events. This may be a movement of the mouse, a key press, or the click on a button. Such low-level events are registered by the window system and handed over to the Qt event loop. The Qt library interprets those events and produces so called signals for the widgets that are affected by the event. Signals are high-level events, that can be handled in a simpler way: imagine, the user presses a mouse button when the cursor resides over a button widget. The low-level event would be a button press event at a certain coordinate (x, y) of the window system. Such information is not very useful. This is why the Qt event handler analyzes the low-level event, determines the according widget at the current mouse position, and produces a high-level button pressed signal for this widget.

4.4 Signals and Slots

How to react? Therefore, Qt knows the concept of Slots. A slot is a ordinary C++-method of a class. The only difference is, that it is connected with a signal. Qt ensures that the slot-method is called each time, the connected signal is emitted. An source code example of the QButton definition will make this more clear:

```cpp
class QButton : public QWidget
{
    Q_OBJECT
    // ...

    signals:
    void pressed();
};
```

A signal `pressed` is defined like an ordinary method. The difference is, that there must no be an implementation of the method. The implementation is generated automatically by the Meta Object Compiler (moc). QButton uses the following command to emit the signal and initiate the event handling:

```cpp
emit pressed();
```

This call sends the signal to all objects connected to the signal. Assume, the following class wants to receive the signal:

```cpp
class AClass {
    Q_OBJECT
    // ...

public slots:
    void doStuff();
};
```

We define a slot (with public access). This slot can be implemented like an ordinary method. Though, we have to ensure, that the class which defines a slots or signals, also includes the macro Q_OBJECT in its definition. This is necessary for the implementation of the signal-slot-mechanism.

Finally Qt needs to know that the signal pressed calls the slot doStuff. This is done by the method connect:

```cpp
QButton * myButton = new QButton();
AClass * myAClass = new AClass();
connect(myButton,SIGNAL(pressed())), myAClass,SLOT(doStuff()));
```

In our special case the method uses signals and slots without any arguments. However, like with C++-methods, signals and slots may feature arbitrary lists of arguments, but they must not be connected if their arguments differ. This mechanism is a powerful technique to propagate events between objects. It is used extensively by the Qt-library. We also suggest, that you define your own signals and slots and use them for handling user interaction.

4.5 The Application Window

Now it is time to proceed with the overview over the application skeleton. The class AppWin follows. It is derived from QMain-Window and implements the main window of our application. The constructor of the class defines the elements of the main window. This includes the widgets for the menu bar. We integrated a mechanism for generating the graphical menu to ease your implementation effort. You just have to define a corresponding menu structure for each main item of the menu. Calling buildGroup with these structures as argument then generates the menu. The skeleton already contains the source code for the file menu. Have a closer look at it before you proceed with the next task.

1st Task: Adding the Edit Menu

Your first task is to create a edit menu with the following structure:

<table>
<thead>
<tr>
<th>Menu Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjust Camera</td>
</tr>
<tr>
<td>Translate Cube</td>
</tr>
<tr>
<td>Rotate Cube</td>
</tr>
<tr>
<td>Scale Cube</td>
</tr>
<tr>
<td>Reset Cube</td>
</tr>
<tr>
<td>Reset Camera</td>
</tr>
<tr>
<td>Reset All</td>
</tr>
</tbody>
</table>
Include the separation lines at the correct positions. Add an approriate keyboard shortcut for each menu item. Also define slots for each menu entry and connect them to signals of the menu. The body of the slots may stay empty at the moment.

Additionally, icons have to be added to the menu items of the edit menu. The icons will be displayed in the tool bar. The corresponding files can be found in the folder icon.

### 4.6 Options Dialog

Now we look at the class OptionsDialog. It implements the dialog, that appears, when the first menu entry of the file menu is clicked. In the first part of the constructor the structure of the dialog is described. Therefore two types of widgets are used. Group or container widgets can contain other widgets and arrange them in a way that is defined by a layout object.

The second widget type are input widgets. Such widgets allow the user to specify input values in various ways (e.g., by typing a text into an input field or by activating a check box). If their internal value changes, input widgets emit a specific signal. To react on such changes, the signals have to be connected to appropriate slots. Qt provides a huge collection of widget classes. You can get an overview at Qt’s online documentation (http://doc.trolltech.com).

#### 2nd Task: Extension of the Options Dialog

For the next task the options dialog has to be extended. Add a new group Show. This group will contain two QCheckBox buttons, with the labels Model and Coord Axis. The buttons have to be connected with keyboard shortcuts and their positions are determined by a Gridlayout. Try to understand how the input values are obtained and handed on when the Apply button is pressed. Extend the mechanism and add further variables of the type bool. Implement the slot for the reset button, which resets the internal state. For the check boxes the default value is checked. Wireframe is the default for the pair of radio buttons.

### 4.7 Drawing Area

The class QGLFrame handles the drawing of the 3D scene at the center of the application window. It is derived from the class QGLWidget that allows to use the 3D graphics library OpenGL within a Qt application. The main concept behind OpenGL is a state machine. All properties of the scene, like the lighting model, light sources, drawing color, line width, fill mode for polygons, or vertex transformations form the current state of the OpenGL state machine. After establishing the OpenGL context the state machine already has a defined initial state. Changes to the state, which are triggered when the user calls certain OpenGL functions, are valid until the same parameter of the state machine is reset.

Three protected methods are of particular importance within a QGLWidget. The initial state of the OpenGL state machine is defined in initializeGL(). Performance reasons suggest to set all OpenGL parameters that will not change during the execution of the program in this method. This could be the background color, the lighting model, etc. The resizeGL(int w, int h) method is called if the size of the widget changes. This happens e.g. when the application window is resized. The most important method is paintGL(). Its task is to draw the scene. Therefore, it is called each time the window has to be refreshed. This is the case when other windows occlude the application window or when the window is moved. 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! The actual OpenGL commands that are required for our implementation are discussed in the next section.

The interaction with the drawing area is based on the mouse events defined in the class QWidgets. There is a method mousePressEvent and mouseMoveEvent. They are called each time when the cursor is located over the drawing area and a mouse button is pressed, or when a mouse button is pressed and the cursor is moved over the OGLWidget, respectively. Insert the following lines into your code to get more insight into the mechanism:

```cpp
qDebug("X: %d Y: %d State: %d",
 m->x(), m->y(), m->state());
```

Observe the relation between action and the triggered events. This knowledge will help you in the following.

#### 3rd Task: Input Handling

Use both methods to implement a mouse input handler with the following behavior: each time the mouse cursor is moved a mouse event is sent to one of four methods. The names of the methods should be translateCube, rotateCube, scaleCube, and adjustCam. All methods have the same arguments: ButtonState button, float dx, and float dy. Thereby, ButtonState contains the information about the pressed mouse buttons; dx and dy the relative movement of the mouse cursor to the position of the last mouse event. The parameters dx and dy have to be normalized to the size of the widget. Use a variable of type NavMode to select the appropriate interaction method. It is helpful to include a small text output for each of the methods for debugging purposes.

To allow the user to choose the interaction mode through the menu items implemented in the edit menu, the appropriate connections have to be established (signal/slot). To also assist the user in remembering the current interaction mode, a corresponding text message has to be printed in the status line. Therefore, get a reference to the status line in AppWin. The status line is defined in every QMainWindow by default. The message has to be updated on each change of the interaction mode. Also at the start of the application the message has to be correct and it should match the state of QGLFrame at every instance. The status line appears as soon as a message is sent to it.

### 5 3D Graphics with OpenGL

#### 5.1 Introduction

The part of our program which is responsible for the 3D graphics is based on the graphics library OpenGL. This standard library provides various commands for displaying different three-dimensional geometric primitives with distinct rendering techniques. For the programming with OpenGL two aspects are important. OpenGL is a so called „immediate mode“ library and uses a „state machine“ for storing parameters.

A primitive gets drawn in the „immediate mode“ when the corresponding call takes places. In contrast, in „retained mode“ only a description of a scene—e.g., in terms of a scene graph—is created. Based on this description a „retained mode“ library can render the scene every time it is necessary. Both approaches have pros and cons. The essential advantage of „immediate mode“ libraries lies in the drawing speed. No overhead is produced by management of scene data. Hence, the performance only depends on the number of graphic primitives and does not depend on the complexity of the scene, which is assembled by these primitives. On the other hand redrawing the scene has to be done explicitly.

The concept of the OpenGL „state machine“ was already introduced in section 4.7. There are a lot of parameters which could be
set when drawing single graphic primitives. It would be quite cumbersome if one had to set all these parameters for each primitive explicitly. That is why OpenGL uses a „state machine“. This means, that for each possible parameter the current value is stored within the library. The drawing is carried out with the parameter values of the current state. OpenGL provides methods to set and query its state to the programmer.

A simple example with different colored polygons should illustrate this concept. For same-colored polygons the desired color has to be set only once. All subsequent polygons will be drawn with this color until the programmer changes the state of the drawing color. Again, all polygons drawn afterwards will be rendered in the new color.

You can find more details of the graphics library either in the OpenGL Programming Guide or in the OpenGL Reference Manual. On the web page of the graphics programming lab you will find links to older pdf versions of those books.

5.2 Graphic Primitives

The most important fact during rendering is the drawing of geometric primitives. OpenGL only supports simple forms like triangles, quadrilaterals, or lines. More complex objects have to be assembled out of those basic primitives.

Drawing in OpenGL happens by the following code sequence: First of all there has to be a glBegin. Afterwards all vertices of the geometry are defined. Finally, glEnd finalizes the primitive. The following example draws a simple quadrilateral.

```c
    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();
```

4th Task: Drawing with OpenGL

Your next challenge is to construct the cube. Create a new method in the class OGLFrame which draws a wireframe cube within the dimensions $[-1,1]^3$.

Now it is getting colorful. All edges along the x-axis should be red colored, all edges along the y-axis should be green, and all edges along the z-axis should be blue (hint: glColor3f). As the wireframe display of the cube is rather boring create an additional method which draws a solid cube with filled colored faces (glPolygonMode). Each of the faces should have a distinct color.

Remember the check boxes in the options dialog which have not been used so far. This should be fixed now. Implement signal-slot pairs to direct the current state of the dialog to the drawing area when the „Apply“ button is pressed. The drawing area should store this state locally. The „Reset“ button should reset the dialog to its state when showing up. „Draw Mode“ determines whether the cube is shown with solid faces or in wireframe. The two „show“ check boxes define the visibility of the cube and the coordinate axis respectively.

5.3 Coordinate Systems and Transformations

So far, all objects are drawn in the origin of the world coordinate system. Now they should be moved. Therefore we define a local model coordinate system for each object which is drawn. The transformation from model to world coordinates is described in OpenGL by a number of affine transformations. They are defined one after another whereby their corresponding matrices are multiplied by OpenGL. Here are a few typical examples:

```c
    glTranslatef(3,-2,1); // movement by (3, -2, 1)
    glScalef(1,2,3);   // object scaling varying per direction
    glRotatef(45,1,0,0); // 45 degree rotation about the axis (1, 0, 0)
    drawCube();
```

Experiment with the transformations by adding them to your program just before the call of the drawing method of the cube. You will recognize that the order of the transformations plays an important part.

All objects are transformed from their local model coordinate system into the world coordinate system before drawing. Have a look at the transformation in paintGL. It defines the coordinate system of the camera.

5th Task: Model-to-World Transformation

The next task is the construction of an appropriate model-to-world transformation. This transformation should allow rotation, scaling and translation of the cube in the following manner: it should be possible to rotate the cube around its x-, y-, and z-axis (at first x, then y, and finally z). The scaling should be applied in the local model coordinate system so that width, height, and depth of the cube are scaled directly. The cube position should be defined relative to the origin of the world coordinate system.

First, start with storing the state of the cube using appropriate variables for each transformation. Then generate the corresponding OpenGL function calls in the correct order. Finally, you can test the various transformations by setting different constant values.

Now it is time for interaction! Having the event handling already implemented, all that remains is to implement the corresponding methods in OGLFrame.

The mouse movements should correspond to the program behavior in the following way:

### Rotate Cube Mode Mapping

<table>
<thead>
<tr>
<th>Button</th>
<th>X Movement</th>
<th>Y Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Y</td>
<td>X</td>
</tr>
<tr>
<td>Middle</td>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>Right</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

### Scale/Translate Cube Mode Mapping

<table>
<thead>
<tr>
<th>Button</th>
<th>X Movement</th>
<th>Y Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>Middle</td>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>Right</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

The tables above show how you should map the mouse movements onto the different transformations. It is upon you to find an appropriate mapping of the relative mouse motion $dx$ and $dy$. Maybe a scaling factor or special handling is necessary. Please bear in mind that the user must not specify illegal values, especially when scaling the cube! Test your implementation and compare the behavior with our sample application. The last step consists of implementing the reset functions which are already defined in the „Edit“ menu.

6th Task: World-to-View Transformation

Now you have to implement the movement of the camera. First, define a variable for the current camera position as well as variables for the rotation around the x-axis (yaw) and the y-axis (pitch). Then you have to realize the following interactions:
5.4 GLSL Shading Language

In the previous tasks all vertices were transformed with the help of the Fixed Function Pipeline of OpenGL. Recent graphics hardware allows to exchange this Fixed Function Pipeline by small self-written programs—so called shaders. There are usually two shaders involved, namely a vertex shader and a fragment shader. Only the latest hardware supports a third shader, the geometry shader, which will not be covered here. Figure 1 shows the OpenGL pipeline of current graphics hardware. As you can see in this figure, all transformations of the vertices are bypassed when a vertex shader is used.

7th Task: Simulation of the Fixed Function Pipeline

In this task a vertex shader and fragment shader has to be developed in GLSL to emulate the Fixed Function Pipeline. GLSL stands for Graphics Library Shading Language and has a syntax almost like C. You can find more information about GLSL in the OpenGL Shading Language Specification on [http://www.opengl.org](http://www.opengl.org). First of all, you have to look if the hardware, i.e., the graphics processing unit (GPU), supports this functionality. In OpenGL this functionality is provided through various extensions. In the framework this check is already done in the function OGLFrame::initializeGLExt. An error flag is set if the extensions are not fully supported. The function glh_init_extensions is used to query whether the functionality is supported and to initialize the corresponding function pointers. The glh_* methods come with a package provided by NVIDA to simplify the handling of OpenGL extensions.

In this task the following functions are necessary to handle GLSL programs which can consist of vertex and fragment shaders.

- **glCreateShader(GLenum type)**
  Creates either a vertex or a fragment shader. type has to be either GL_VERTEX_SHADER or GL_FRAGMENT_SHADER. If 0 is returned, a new shader could not be created.

- **glDeleteShader(GLuint shader)**
  Marks shader shader for deletion. The shader is deleted as soon it is no longer attached to a GLSL program. It is wise to call this function right after the shader has been attached.

- **glShaderSource(GLuint shader, GLsizei count, const char **string, const int *length)**
  Loads the source code from the array string into the shader. count defines the number of character strings in string (in our case this should be 1). If length is NULL then the character strings in string have to be null-terminated.

- **glCompileShader(GLuint shader)**
  Compiles the shader shader. The status of the compilation can be queried with glGetShaderiv and the parameter GL_COMPILE_STATUS.

- **glGetShaderiv(GLuint shader, GLenum pname, GLint *params)**
  Queries different OpenGL states for the shader shader and stores the result in params. In case of GL_COMPILE_STATUS as pname the result is of type GLint and contains a value of either GL_TRUE or GL_FALSE.

- **glCreateProgram(void)**
  Creates a GLSL program to which a vertex shader and/or a fragment shader can be attached. The creation of a new GLSL program failed if 0 is returned.

- **glDeleteProgram(GLuint program)**
  Deletes the GLSL program program. All attached shaders marked for deletion (see glDeleteShader) are also deallocated.

- **glAttachShader(GLuint program, GLuint shader)**
  Attaches the shader shader to the GLSL program program. If both, vertex and fragment shader, should be used, this function has to be called for both. It is possible to detach a shader with glDetachShader (same parameters).
• `glLinkProgram(GLuint program)`
  
  A GLSL program has to be linked with this function before it can
  be used. The status of the compilation can be queried with
  `glGetProgramiv` and the parameter `GL_LINK_STATUS`. Shaders have to be compiled before this call.

• `glGetProgramiv(GLuint program, GLenum pname, GLint *params)`
  
  Queries different OpenGL states for the GLSL program
  and stores the result in `params`.

• `glUniform1i[2][3][4]{f|i}v(GLuint location, {float|int} value, ...)`
  
  Variables which are constant for one or more primitives are called `Uniforms`. These uniforms can be accessed in both, 
  vertex and fragment shader. The function `glUniform1iv` sets a uniform at `location` of the type `float` to `value`. The
  location has to be determined previously.

• `glGetUniformLocation(GLuint program, const char *name)`
  
  The location of a uniform variable with the name `name` can be
determined with this function. The type of a location is an
unsigned integer (GLuint).

You should now extend the options dialog to allow switching
between the Fixed Function Pipeline and the GLSL program. Similar
to task 2, you should add two radio buttons, one for the Fixed Function Pipeline and one for the shaders. If the GPU does not sup-
port GLSL programs both radio buttons should be disabled and the
button for the Fixed Function Pipeline should be preselected.

Now you can start implementing the setup stage and the utili-
zation of the GLSL program. You will need two shaders (a ver-
tex and a fragment shader) and a GLSL program. You can find
source code of both shaders in `VertexShader.glsl` and `FragmentSha-
der.glsl`. The source code is already read and assigned to two char-
acter strings via the function `loadFile(char *fileName)` in `initializeGLExt()`. You have to assign the source code to the corresponding shader with `glShaderSource`. After that, the shader can be compiled. Possible errors during compilation can be
shown with the function `printInfoLog` (the boolean flag `programObject` has to be false for shader logs). You can check whether the compilation was successful with a call of `glGetShaderiv` and the flag `GL_COMPILE_STATUS` as described above.

Having both shaders compiled successfully you can attach them to a previously created GLSL program object `glCreateProgram`. Afterwards the program has to be linked. Again, possible errors and the link state can be checked with `printInfoLog` and `glGetProgramiv` respectively. The linked program can now be enabled by the call of `glUseProgram(GLuint program)`. To disable the GLSL program simply call `glUseProgram(0)`. Don’t forget to free both shaders and the program when terminating HelloCube!

If the GLSL program has been successfully linked and enabled you should see a horizontal white line, which divides the viewport
horizontally, instead of the cube. However, the line is only visi-
ble when the cube is shown in the Fixed Function Pipeline mode! Now all transformations of the Fixed Function Pipeline ha-
\v{v}e to be implemented in the vertex shader. Therefore you should enhance the file `VertexShader.glsl`. More information on this topic can be found in the OpenGL Shading Language Specification on `http://www.opengl.org/`. You should see no difference between the
Fixed Function Pipeline and the GLSL program if the shaders are implemented correctly.

8th Task: Procedural Shader

In this task you will experience some of the possibilities pro-
gammable fragment shaders offer. The goal of this task is a proce-
dural shader which changes the appearance of the cube. The appe-
ance can be modified in various ways. One of them is the surface
texture. With a procedural shader it is possible to create different
looking textures. In comparison to textures created from 2D images,
the pixels of procedural textures are calculated by an algorithm on
the fly. One major advantage of such an approach is, that it produces
resolution independent textures. This is possible, as the algorithm
is evaluated for each drawn pixel.

But before you can start you have to extend the options dialog
by a slider (default value 1). This slider should act as a scaling fac-
tor for the procedural shader. Setup a signal/slot pair to propagate
the slider value to the class `OGLFrame`. To transfer the factor from
`OGLFrame` to the GLSL program, you have to use an uniform va-
riable. These `Uniforms` are global variables of GLSL shaders and
are constant per primitive.

Now we need a parametrization of the cube surface. As the cube
vertices lie on the main axes of the local coordinate system, you
can directly use the 3D coordinates. Hence, each face of the cube
is parametrized along two axes. You have to scale and to bias the
coordinates to receive a range of [0, 1]³. This should all be done in
the vertex shader because you have direct access to the 3D coor-
dinates there. To pass this parametrization from the vertex shader
to the fragment shader you have to use `Varying` variables. They are
like `Uniforms` global variables of a shader. But as the name sug-
gests, they can vary for each single vertex. The usual procedure is
to define a varying variable of the same name and type both in the
vertex and the fragment shader. In the vertex shader the value of
the varying output variable is set for each vertex. In the raterizer-
ization step the graphics hardware automatically interpolates linearly
between the values defined at the vertex locations and assigns the
interpolated values to the generated fragments. In the fragment sha-
der the value of a fragment can be accessed through the varying
input variable.

If implemented correctly, you can show the result if you map the
parametrization to the color in the fragment shader. This should lead
to smooth color transitions on each face of the cube. If a scaling
factor of one is chosen, the cube should look like as if the Fixed Function Pipeline is used. Higher values should lead to a repeated
pattern.

Now you should rewrite the fragment shader to produce a 3D
checker-board texture. First, think how to realize such a pattern and
then try to implement this function in the fragment shader. As the
shader is loaded from a file when the application is started, you
only have to restart your program instead of recompiling it to see
the effects of the shader.
6 Criteria for Grading

You can achieve a total of 20 points in this assignment. Therefore the following criteria have to be fulfilled:

<table>
<thead>
<tr>
<th>Points</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>The ,,Edit“ menu is constructed as specified (1 point). All slots are implemented correctly (1 point).</td>
</tr>
<tr>
<td>2</td>
<td>The options dialog consists of the described groups (1 point). The storage of the internal state is implemented correctly (1 point).</td>
</tr>
<tr>
<td>3</td>
<td>The handling of the mouse events takes place as described. The signal-slot mechanism was used and realized properly (2 points). The message in the status bar is correct at all times (1 point).</td>
</tr>
<tr>
<td>3</td>
<td>The cube can be rendered in both modes—wireframe and solid colored faces (1 point each). The mode is controlled by the option menu (1 point).</td>
</tr>
<tr>
<td>3</td>
<td>All transformations work as described above (2 points). The ,,Reset“ functions are implemented sound (1 point).</td>
</tr>
<tr>
<td>2</td>
<td>The movements of the camera are correct (1 point). The field-of-view is also adjustable as stated (1 point).</td>
</tr>
<tr>
<td>5</td>
<td>The vertex program is functionally correct (1 point). The uniform variable and the varying variable are set and used properly (1 point). The procedural fragment shader produces a 3D checker texture on the cube (2 points). The options dialog is adapted to hardware functionality and the slider for scaling is implemented correctly (1 point).</td>
</tr>
</tbody>
</table>

Precondition for grading is the error-free compilation of the program on the computers of the VISGS pool. There will be deduction of points if the program compilation leads to warnings (Warnings of QT in the style of “...has virtual functions but non-virtual destructor” are excluded therefrom)! The annotation of the source code should be that extensive to provide enough information to the author to explain the functionality of the code to the supervisors.