OpenGL Tutorial

Release 1.0

Nokia, Qt Learning

June 01, 2012

CONTENTS

CHAPTER

ONE

INTRODUCTION

This tutorial provides a basic introduction to OpenGL and 3D computer graphics in general. It shows how to make use of Qt and its OpenGL related classes to create 3D graphics.

We will use the core features of OpenGL 3.0/ 2.0 ES and all following versions, which means that we will be utilizing OpenGL's programmable rendering pipeline to write our own shaders with the OpenGL Shading Language (GLSL) / OpenGL ES Shading Language (GLSL / ES).

Chapter one gives an introduction to 3D computer graphics and the OpenGL API including the OpenGL Shading Language (GLSL) / OpenGL ES Shading Language (GLSL / ES). If you are already familiar with this topic and only want to see how to use OpenGL in your Qt programs, you can skip this introductory chapter and continue on to chapter two.

In chapter two, we present examples which utilize the information covered in chapter one and show how to use OpenGL together with Qt's OpenGL related functionality.

At the end of this tutorial you find some references and links which may come in handy especially when working through the examples. Please note that this tutorial is meant to get you started with this topic and can not go into the same depth as a decent OpenGL dedicated book. Also note that Qt's OpenGL-related classes do a lot of work for you by hiding some of the details which you would encounter if you wrote your programs using only OpenGL's API.

In the example part, we will use Qt's high level functionality whenever possible and only briefly name the differences. So if you intend to get a complete understanding of how to use native, you should additionally consult a tutorial or book dedicated to this topic.

1.1 What's OpenGL

OpenGL is the industry's most widely used 2D and 3D graphics API. It is managed by the nonprofit technology consortium, the Khronos Group, Inc. It is a highly portable, scalable, cross-language and cross-platform specification that defines a uniform interface for the computer's graphics accelerator. It guarantees a set of basic capabilities and allows vendors to implement their own extensions.

OpenGL is a low level API which requires the programmer to tell it the exact steps needed to render a scene. You cannot just describe a scene and have it displayed on your monitor. It is up to you to specify geometry primitives in a 3D space, apply coloring and lighting effects and render the objects onto the screen. While this requires some knowledge of computer graphics,

it also gives you a lot of freedom to invent your own algorithms and create a variety of new graphics effects.

The sole purpose of OpenGL is to render computer graphics. It does not provide any functionality for window management or for handling events like user input. This is what we use Qt for.

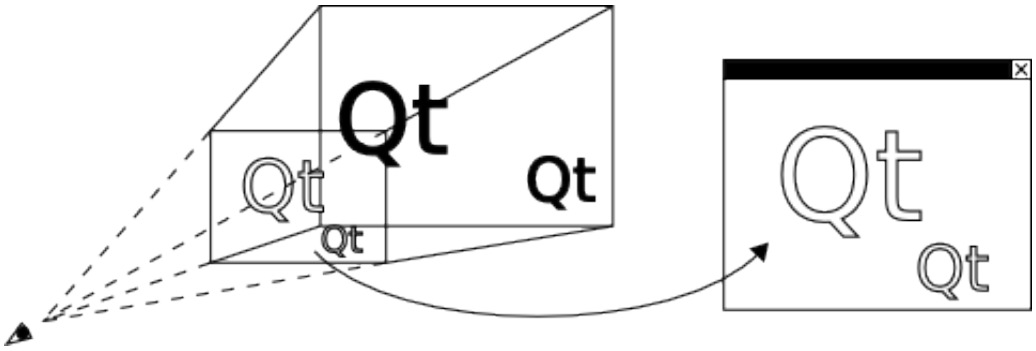
1.2 Drawing in 3D Space

The geometry of three dimensional objects is described by an arrangement of very basic building blocks (primitives) such as single points, lines or triangles. Triangles are the most common ones as they are used to approximate the surface of objects. Each surface can be split up into small planar triangles. While this works well on edged objects, smooth objects like spheres will look jagged. Of course you could use more triangles to improve the approximation, but this comes at the cost of performance as more triangles will need to be processed by your graphics card. Instead of simply increasing the polygon count, you should always consider additional techniques such as improving the lighting algorithm or adapting the level of detail.

To define the spatial properties of your objects, you set up a list of points, lines and/or triangles. Each primitive in turn is specified by the position of its corners (a vertex / vertices). Thus it is necessary to have a basic understanding of how to define points in space and to manipulate them efficiently. But we will brush up our linear algebra knowledge in a moment.

To see those objects, you need to apply coloring to your primitives. Color values are often defined for each primitive (for each vertex to be precise) and used to paint or fill in with color. For more realistic applications, images (called textures) are placed on top of them. The appearance can be further adapted according to material properties or lighting. So how do we actually get our scene displayed on the screen.

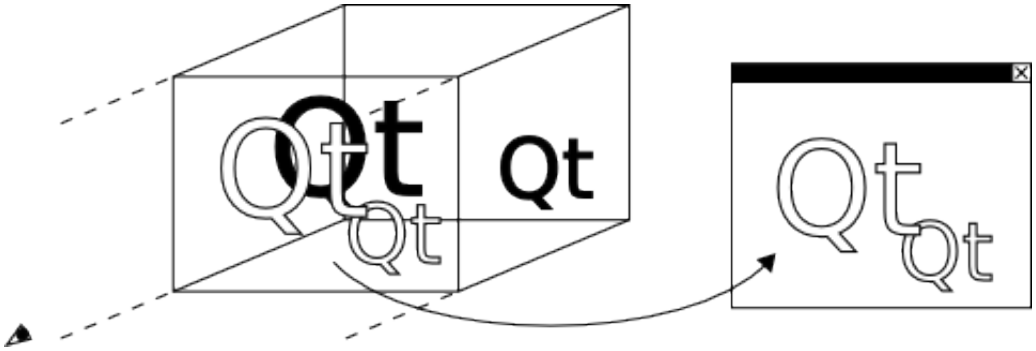
Since the computer screen is a two dimensional device, we need to project our objects onto a plane. This plane is then mapped to a region on our screen called the *viewport*. To understand this process illustratively, imagine that you are standing in front of the window and sketching the outlines of the objects you see outside onto the glass without moving your head. The drawing on the glass then represents a two dimensional projection of your environment. Though the technique is different, this kind of projection also occurs in a camera when taking a picture.



The clipped pyramid is called the *viewing volume*. Everything that is located inside this volume is projected onto the near side towards the imaginary viewer. Everything else is not drawn.

There are two major types of projections: perspective projections and orthographic projections.

What we just introduced is called *perspective projection*. It has a viewing volume in the form of a frustum and adds the illusion that distant objects appear smaller than closer objects of the same size. This greatly contributes to realism, and therefore, is used for simulations, games and VR (virtual reality) applications. The other type is called *orthographic projection*. Orthographic projections are specified by a rectangular viewing volume. Every two objects that have the same size also have the same size in the projection regardless of its distance from the viewer. This is often used in CAD (computer aided design) tools or when dealing with 2D graphics.



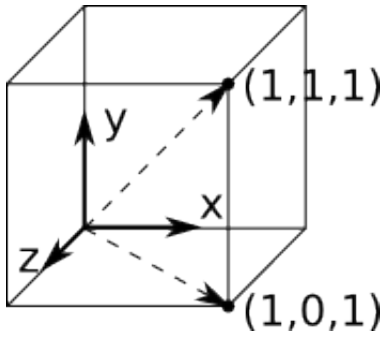
1.3 A Short Recapitulation of Linear Algebra

Since it is essential to have a basic understanding of linear algebra when writing OpenGL programs, this chapter will briefly state the most important concepts involved. Although we will mostly have Qt do the math, it is still good to know, what is going on in the background.

The location of a 3D point in relation to an arbitrary coordinate system is identified by its x-, y- and z-coordinates. This set of values is also called a *vector*. When used to describe primitives, it is called a *vertex*.

$$v = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

An object is then represented by a list of vertices.



One thing you will often want to do is change the position, size or orientation of your object.

Translating an object is as easy as adding a constant vector (here named *d*') that specifies the displacement to all your objects' vertices (here named *v*).

$$v_{new} = v_{old} + d = \begin{pmatrix} v_{old,x} \\ v_{old,y} \\ v_{old,z} \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} = \begin{pmatrix} v_{old,x} + d_x \\ v_{old,y} + d_y \\ v_{old,z} + d_z \end{pmatrix}$$

Scaling means multiplying the vertices by the desired ratio (here named *s*).

$$v_{new} = s \cdot v_{old} = \begin{pmatrix} s \cdot v_{old,x} \\ s \cdot v_{old,y} \\ s \cdot v_{old,z} \end{pmatrix}$$

Rotating, stretching, shearing, or reflecting is more complicated and is achieved by multiplying the vertices by a transformation matrix (here named *T*'). A matrix is basically a table of coefficients that, when multiplied by a vector, yields a new vector with each element that is a linear combination of the multiplied vector's elements.

$$T = \begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix}$$

$$v_{new} = T \cdot v_{old} = \begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} \cdot \begin{pmatrix} v_{old,x} \\ v_{old,y} \\ v_{old,z} \end{pmatrix} = \begin{pmatrix} t_{11} \cdot v_{old,x} + t_{12} \cdot v_{old,y} + t_{13} \cdot v_{old,z} \\ t_{21} \cdot v_{old,x} + t_{22} \cdot v_{old,y} + t_{23} \cdot v_{old,z} \\ t_{31} \cdot v_{old,x} + t_{32} \cdot v_{old,y} + t_{33} \cdot v_{old,z} \end{pmatrix}$$

As an example, these are matrices rotating the vector around the coordinate system's x, y, and z axes. Arbitrary rotations can be composed by a combination of these.

$$T_{rot,x}(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & -sin(\alpha) \\ 0 & sin(\alpha) & cos(\alpha) \end{pmatrix}$$

$$T_{rot,y}(\alpha) = \begin{pmatrix} cos(\alpha) & 0 & sin(\alpha) \\ 0 & 1 & 0 \\ -sin(\alpha) & 0 & cos(\alpha) \end{pmatrix}$$

$$T_{rot,z}(\alpha) = \begin{pmatrix} cos(\alpha) & -sin(\alpha) & 0 \\ sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

There is also one matrix that does not change a vector at all. It is called the *identity matrix* and consists of ones on the main diagonal and zeros elsewhere.

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

If you use a matrix to transform a vector, it is important that the matrix is written on the left side of the multiplication sign and the vector is on the right side. Also, the number of the matrice's colums needs to match the number of the vector's components. Otherwise the multiplication is mathematicaly invalid and math libraries may return unexpected results.

Keep in mind that transformations are not commutative, i.e. the result of a concatenation of transformations depends on their order. For example, it makes a difference whether you first rotate an object and then translate it or if you do it the other way around.

As it is more convenient (and even faster for OpenGL) to express all these operations as a single matrix vector multiplication, we extend our formalism to so called *homogeneous coordinates*. This also enables us to easily apply all kinds of *affine transformations* such as the projections, which we discussed in chapter 1.2. We basically add a fourth dimension, called a *scaling factor*, to our vertices. This might seem to complicate things, but you actually do not really have to pay attention to that factor as it is set to 1 by default and you will rarely change it yourself. All you need to do is declare your vertices with an additional element set to 1 (which is even often implied by default). (In this chapter we denote homogeneous coordinates by a hat on the variable names.)

$$v = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \rightarrow \hat{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ v_w \end{pmatrix}$$

A transformation can then be written as follows:

$$\hat{v}_{new} = \hat{T} \cdot \hat{v}_{old} = \begin{pmatrix} T & d \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} v_{old} \\ 1 \end{pmatrix} = \begin{pmatrix} t_{11} & t_{12} & t_{13} & d_x \\ t_{21} & t_{22} & t_{23} & d_y \\ t_{31} & t_{32} & t_{33} & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} v_{old,x} \\ v_{old,y} \\ v_{old,z} \\ 1 \end{pmatrix} = \begin{pmatrix} v_{new,x} \\ v_{new,y} \\ v_{new,z} \\ 1 \end{pmatrix}$$

A series of transformations can be written as a series of matrix multiplications, and the resulting transformation can be stored in a single matrix.

$$\hat{v}_{new} = \hat{T}_3 \cdot \hat{T}_2 \cdot \hat{T}_1 \cdot \hat{v}_{old} = \hat{T}_{resulting} \cdot \hat{v}_{old}$$

1.4 Coordinate Systems & Frame Concept

How can we use this knowledge of linear algebra to put a three dimensional scene on screen? In this tutorial, we will use the most widely used concept called the *frame concept*. This pattern allows us to easily manage objects and viewers (including their positions and orientations) as well as the projection that we want to apply.

Imagine two coordinate systems: A and B. Coordinate system B originates from coordinate system A via a translation and a rotation that can be described by the matrix

$$T$$

Then for each point defined as

$$p_B$$

in coordinate system B, the corresponding coordinates of point

$$p_A = T \cdot p_B$$

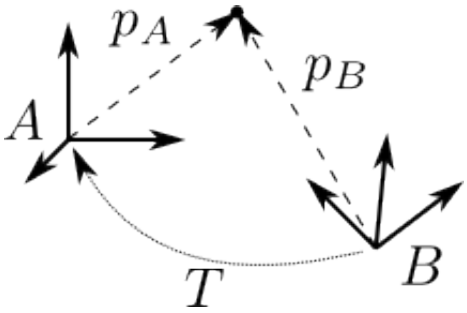
can be calculated.

$$p_A$$

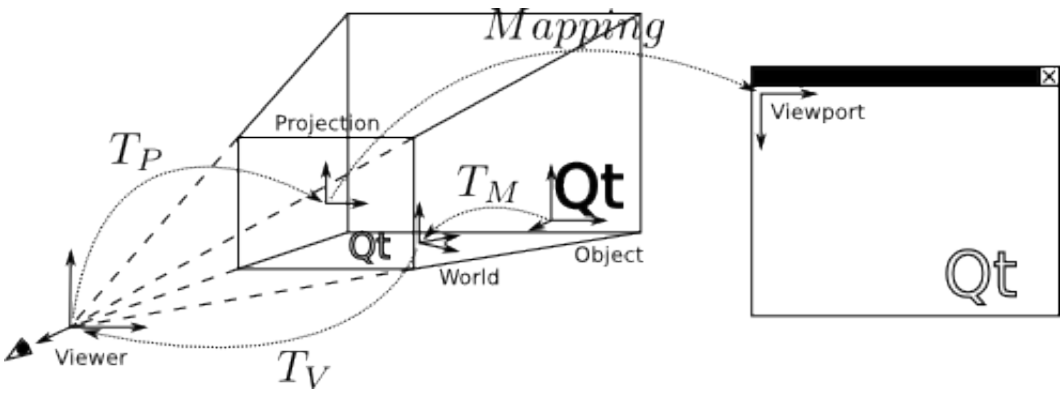
and

$$p_B$$

represent the same point in space but are only noted down differently.



As for the frame concept, every instance of an object is bound to its own coordinate system (also referred to as its *frame*). The position and orientation of each object is then defined by placing the objects' frames inside the world's frame. The same applies to the viewer (or *camera*) with one difference: for simplicity, we actually do not place the viewer's frame inside the world's frame, but instead do it the other way around (i.e. placing the world's frame inside the viewer's frame).

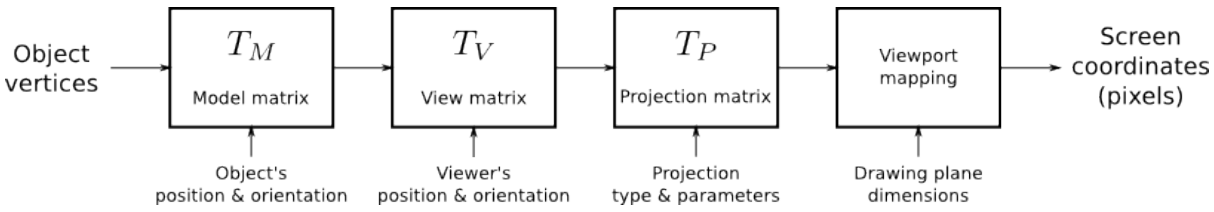


This means we define the position and rotation of every instance of an object in relation to the world's coordinate system. The matrix defined by these parameters, which allows us to calculate an object's vertices inside the world's coordinate system, is commonly called the *model matrix*. Subsequently, we move from world coordinates to viewer coordinates (commonly called *eye coordinates*) using a matrix called the *view matrix* in just the same way. After that, we apply the projection which transforms the object's vertices from viewer coordinates to the projection plane. This is done by a matrix called the *projection matrix*, which yields normalized device coordinates with x, y and z values ranging from -1 to +1 (The -1 and +1 values correspond to positions on the viewing volume's borders). OpenGL then maps all the object's points on this projection plane to the viewport that is shown on the screen.

Another matrix that is often used is the *model-view-projection matrix*. It is the concatenation of the aforementioned three matrices. The *model-view-projection matrix* is generally passed to the *vertex shader*, which multiplies this matrix by the object's vertices in order to calculate the projected form. You will learn about shaders in a later chapter.

The definition of these matrices has various advantages:

- In the design phase, every object's model (i.e. its set of vertices) can be specified in relation to an arbitrary coordinate system (for example its center point)
- The transformation process is divided into small steps, which as such are quite illustrative
- All the used transformation matrices can be calculated, stored and combined efficiently



The figure above illustrates the steps that are required to yield proper screen coordinates from object vertices. Different kinds of transformations are applied in a certain order. You throw in some object vertices and, after some number crunching, you get the appropriate screen coordinates. In this figure, you can also easily see why this part of 3D programming is called the *transformation pipeline*.

1.5 The OpenGL Rendering Pipeline

The OpenGL rendering pipeline is a high level model which describes the basic steps that OpenGL takes to render a picture on the screen. As the word *pipeline* suggests, all operations are applied in a particular order. Very simply put, the rendering pipeline has a state, takes some inputs and returns an image to the screen.

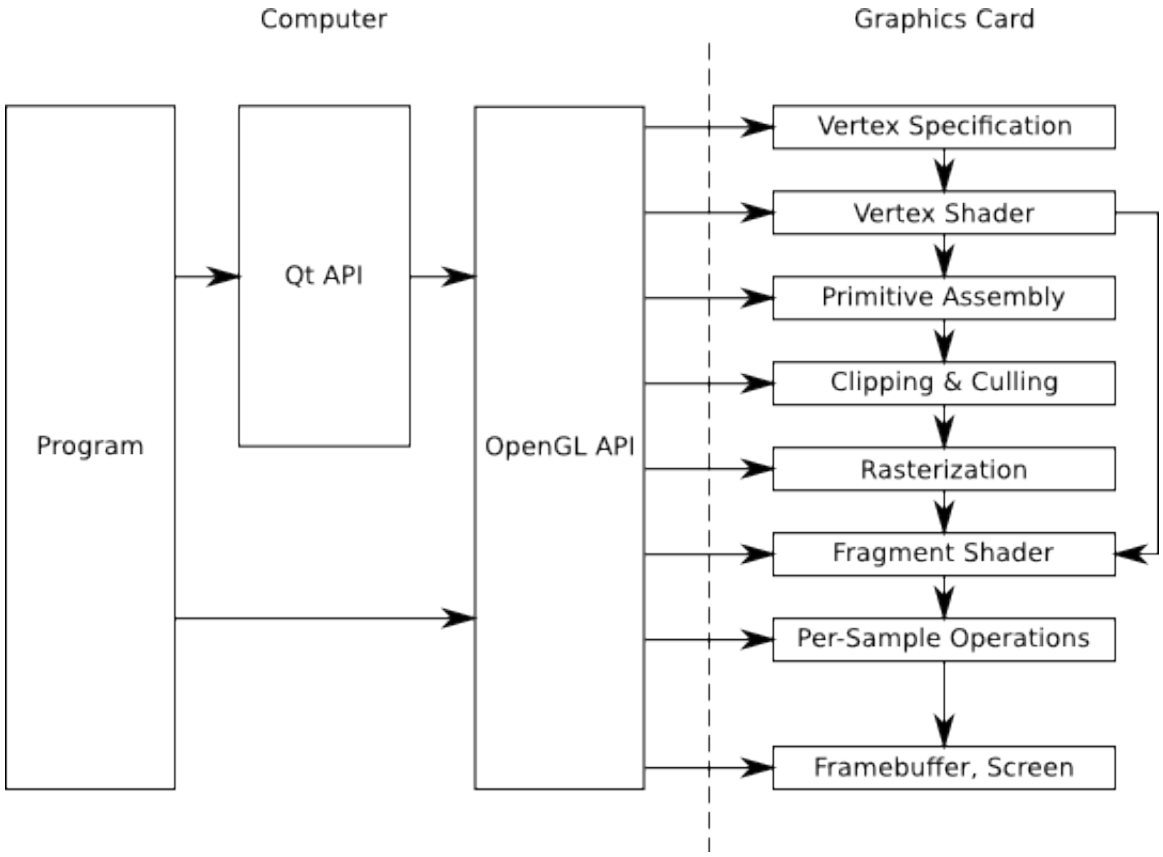
The state of the rendering pipeline affects the behavior of its functions. As it is not practical to set options every time we want to draw something, we can set parameters beforehand. These parameters are then used in all subsequent function calls. For example, once you've defined a background color, that color is used to clear the screen until you change it to something else. You can also turn distinct capabilities like depth testing or multisampling on and off. Therefore, to draw an overlay image on top of your screen, you would first draw the scene with depth testing enabled, then disable depth testing, and after that, draw the overlay elements, which will then always be rendered on top of the screen regardless of their distance from the viewer.

The inputs to the pipeline can be provided as single values or arrays. Most of the time these values will represent vertex positions, surface normals, textures, texture coordinates or color values.

The output of the rendering pipeline is the image that is displayed on the screen or written into memory. Such a memory segment is then called a framebuffer.

The figure below shows a simplified version of the pipeline. The elements that are not relevant to this tutorial were omitted (such as tesselation, geometry shading and transform feedback).

The main program that resides inside the computer's memory, and is executed by the CPU, is displayed in the left column. The steps executed on the graphics card are listed in the column on the right.



The graphics card has its own memory and a GPU just like a small, but powerful computer that is highly specialized in processing 3D data. Programs that run on the GPU are called shaders. Both the host computer and the graphics card can work independently. To take full advantage of hardware acceleration, you should keep both of them busy at the same time.

During *vertex specification*, the ordered list of vertices that gets streamed to the next step is set up. This data can either be sent by the program that is executed on the CPU one vertex after the other or read from GPU memory directly using buffer objects. However, repeatedly getting data via the system bus should be avoided whenever possible since it is faster for the graphics card to access its own memory.

The *vertex shader* processes data on a per vertex basis. It receives this stream of vertices along with additional attributes like associated texture coordinates or color values, and data such as the model-view-projection matrix. Its typical task is to transform vertices and to apply the projection matrix. Besides its interface to the immediately following stage, the vertex shader can also pass data to the fragment shader directly.

During the *primitive assembly* stage, the projected vertices are composed into primitives. These primitives can be triangles, lines, point sprites, or more complex entities like quadrics. The user decides which kind of primitive should be used when calling the draw function. For example, if the user wants to draw triangles, OpenGL takes groups of three vertices and converts them all into triangles.

During the *clipping and culling* stage, primitives that lie beyond the viewing volume, and therefore are not visible anyway, are removed. Also, if face culling is enabled, every primitive that does not show its front side (but its reverse side instead) is removed. This step effectively contributes to performance.

The *rasterisation* stage yields so called *fragments*. These fragments correspond to pixels on the screen. Depending on the user's choice, for each primitive, a set of fragments may be created. You may either fill the whole primitive with (usually colored) fragments, or only generate its outlines (e.g. to render a wireframe model).

Each fragment is then processed by the *fragment shader*. The most important output of the fragment shader is the fragment's color value. Texture mapping and lighting are usually applied during this step. Both the program running on the CPU and the vertex shader can pass data to it. Obviously it also has access to the texture buffer. Because there are usually a lot of fragments in between a few vertices, values sent by the vertex shader are generally interpolated. Whenever possible, computational intensive calculations should be implemented in the vertex instead of in the fragment shader as there are usually many more fragments to compute than vertices.

The final stage, *per-sample operations*, applies several tests to decide which fragments should actually be written to the framebuffer (depth test, masking etc). After this, blending occurs and the final image is stored in the framebuffer.

1.6 OpenGL API

This chapter will explain the conventions used in OpenGL. Although we will try to use Qt's abstraction to the OpenGL API whenever possible, we will still need to call some of its functions directly. The examples will introduce you to the required functions.

The OpenGL API uses its own data types to improve portability and readability. These types are guaranteed to hava a minimum range and precision on every platform.

Type	Description
GLenum	Indicates that one of OpenGL's preprocessor definitions is expected.
GLboolean	Used for boolean values.
GLbitfield	Used for bitfields.
GLvoid	sed to pass pointers.
GLbyte	1-byte signed integer.
GLshort	GLshort 2-byte signed integer.
GLint	4-byte signed integer.
GLubyte	1-byte unsigned integer.
GLushort	2-byte unsigned integer.
GLuint	4-byte unsigned integer.
GLsizei	Used for sizes.
GLfloat	Single precision floating point number.
GLclampf	Single precision floating point number ranging from 0 to 1.
GLdouble	Double precision floating point number.
GLclampd	Double precision floating point number ranging from 0 to 1.

OpenGL's various preprocessor definitions are prefixed with *GL_*. Its functions begin with *gl*.

A function that triggers the rendering process for example is declared as *void glDrawArrays(GLenum mode, GLint first, GLsizei count)*.

1.7 The OpenGL Shading language

As we have already learned, programming shaders is one of the core requirements when using OpenGL. Shader programs are written in a high level language called *The OpenGL Shading Language (GLSL)*, which is a language very similar to C. To install a shader program, the shader source code has to be sent to the graphics card as a string, where the program then needs to be compiled and linked.

The language specifies various types suited to its needs.

Type	Description
void	No *function return* value or *empty parameter* list.
float	Floating point value.
int	Signed integer.
bool	Boolean value.
vec2, vec3, vec4	Floating point vector.
ivec2, ivec3, ivec4	Signed integer vector.
bvec2, bvec3, bvec4	Boolean vector.
mat2, mat3, mat4	2x2, 3x3, 4x4 floating point matrix.
sampler2D	Access a 2D texture.
samplerCube	Access cube mapped texture.

All these types may be combined using a C like structure or array.

To access the elements of a vector or a matrix, square brackets "[]" can be used (e.g. *vector[index] = value* and *matrix[column][row] = value;*). In addition to this, the vector's named components are accessible by using the field selector operator "." (e.g. *vector.x = xValue* and *vector.xy = vec2(xValue, yValue)*). The names *(x, y, z, w)* are used for positions. *(r, g, b, a)* and *(s, t, p, q)* are used to address color values and texture coordinates respectively.

To define the linkage between different shaders as well as between shaders and the application, GLSL provides variables with extra functionality by using storage qualifiers. These storage qualifiers need to be written before the type name during declaration.

Storage Qualifier	Description
none	(default) Normal variable
const	Compile-time constant
attribute	Linkage between a vertex shader and OpenGL for per-vertex data.Since the vertex shader is executed once for every vertex, this read-only value holds a new value every time it runs. It is used to pass vertices to the vertex shader for example.
uniform	Linkage between a shader and OpenGL for per-rendering data. This read-only value does not change across the the whole rendering process. It issued to pass the model-view-projection matrix for example since this parameter does not change for one object.
varying	Linkage between the vertex shader and the fragment shader for interpolated data. This variable is used to pass values calculated in the vertex shader to the fragment shader. For this to work, the variables need to share the same name the in both shaders. Since there are usually a lot of fragments in between a few vertices, the data calculated by the vertex shader is (by default) interpolated. Such variables are often used as texture coordinates or lighting calculations.

To send data from the vertex shader to the fragment shader, the *out* variable of the vertex shader and the *in* variable of the fragment shader need to share the same name. Since there are usually a lot of fragments in between a few vertices, the data calculated by the vertex shader is by default interpolated in a perspective correct manner. To enforce this behavior, the additional qualifier *smooth* can be written before *in*. To use linear interpolation, the *noperspective* qualifier can be set. Interpolation can be completely disabled by using *flat*. Then, for all the fragments in between a primitive, the value output by the first vertex of this primitive is used.

This kind of variables are commonly called *varyings*, due to this interpolation and because in earlier versions of OpenGL this shader-to-shader linkage was achieved using a variable qualifier called *varying* instead of *in* and *out*.

Several built-in variables are used for communication with the pipeline. We will use the following:

Variable Name	Description
vec4 gl_Position	The rasterization step needs to know the position of the transformed vertex. Therefore, the vertex shader needs to set this variable to the calculated value.
vec4 gl_FragColor	This variable defines the fragment's RGBA color that will eventually be written to the frame buffer. This value can be set by the fragment shader.

When using multiple variable qualifiers, the order is <*storage qualifier*> <*precision qualifier*> <*type*> <*name*>.

Just like in C, every GLSL program's entry point is the *main()* function, but you are also allowed to declare your own functions. Functions in GLSL work quite differently than those in C. They do not have a return value. Instead, values are returned using a calling convention called *value-return*. For this purpose, GLSL uses parameter qualifiers, which need to be written before the variable type during function declaration. These qualifiers specify if and when values are exchanged between a function and its caller.

Parameter qualifier	Description
in	(default) On entry, the variable is initialized to the value passed by the caller.
out	On return, the value of this variable is written into the variable passed by the caller. The variable is not initialized.
inout	A combination of in and out. The variable is both initialized and returned.

There are actually many more qualifiers, but listing all of them goes beyond the scope of this tutorial.

The language also offers control structures like *if*, *switch*, *for*, *while*, and *do while*, including *break* and *return*. Additionally, in the fragment shader, you can call *discard* to exit the fragment shader and have that fragment ignored by the rest of the pipeline.

GLSL also uses several preprocessor directives. The most notable one that you should use in all of your programs is *#version* followed by the three digits of the language version you want to use (e.g. *#version 330* for version 3.3). By default, OpenGL assumes version 1.1, which might not always be what you want.

Although GLSL is very similar to C, there are still some restrictions you should be aware of:

- Functions may not be recursive.
- For-loops must have an iteration count that is known at compile time.
- There are no pointers.
- Array indexing is only possible with constant indices.
- Type casting is only possible using constructors (e.g. *myFloat = float(myInt);*).

Note: The scene you want to render may be so complex that it has thousands of vertices and millions of fragments. This is why modern graphics cards are equipped with several stream processing units, each of which executes one vertex shader or fragment shader at a time. Because all vertices and fragments are processed in parallel, there is no way for the shader to query the properties of another vertex or fragment.

CHAPTER

TWO

USING OPENGL IN YOUR QT APPLICATION

Qt provides a widget called *QGLWidget* for rendering OpenGL Graphics, which enables you to easily integrate OpenGL into your Qt application. It is subclassed and used like any other QWidget and is cross platform. You usually reimplement the following three virtual methods:

- *QGLWidget::initializeGL()* - sets up the OpenGL rendering context. It is called once before the first time *QGLWidget::resizeGL()* or *QGLWidget::paintGL()* are called.
- *QGLWidget::resizeGL()* - gets called whenever the *QGLWidget* is resized, and after initialization. This method is generally used for setting up the viewport and the projection.
- *QGLWidget::paintGL()* - renders the OpenGL scene. It is comparable to *QWidget::paint()*.

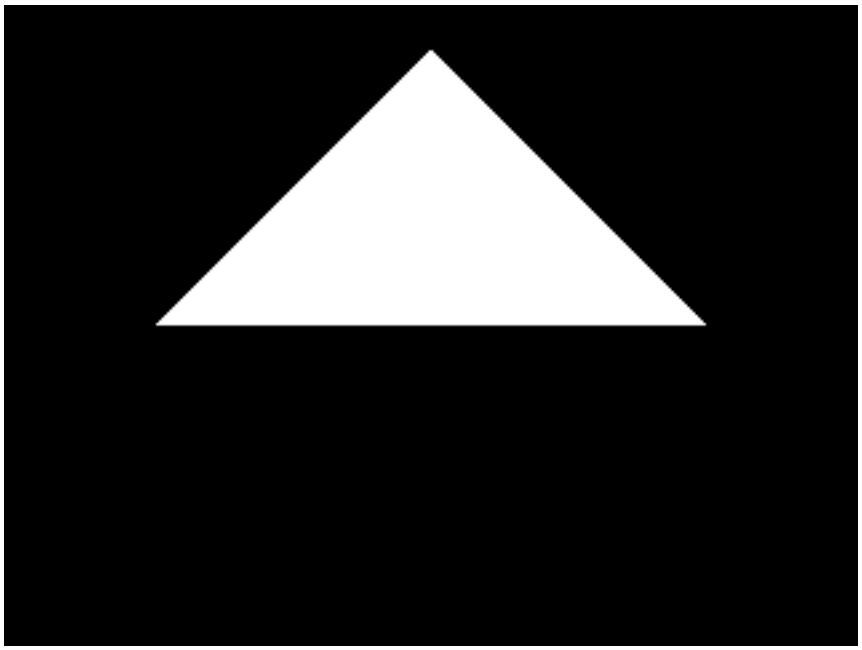
Qt also offers a cross platform abstraction for shader programs called *QGLShaderProgram*. This class facilitates the process of compiling and linking the shader programs as well as switching between different shaders.

Note: You might need to adapt the versions set in the example source codes to those supported by your system.

2.1 Hello OpenGL

We are beginning with a small *Hello World* example that will have our graphics card render a simple triangle. For this purpose we subclass *QGLWidget* in order to obtain an OpenGL rendering context and write a simple vertex and fragment shader.

This example will confirm if we have set up our development environment properly and if OpenGL is running on our target system.



Note: The source code related to this section is located in *examples/hello-opengl/* directory.

First of all, we need to tell qmake to use the *QtOpenGL* module. So we add:

```
QT += opengl
```

The *main()* function only serves the purpose of instantiating and showing our *QGLWidget* subclass.

```
int main(int argc, char **argv)
{
    QApplication a(argc, argv);

    GlWidget w;
    w.show();

    return a.exec();
}
```

Our OpenGL widget class is defined as follows:

We want the widget to be a subclass of *QGLWidget*. Because we might later be using signals and slots, we invoke the *Q_OBJECT* macro. Additionally we reimplement *QWidget::minimumSizeHint()* and *QWidget::sizeHint()* to set reasonable default sizes.

To call the usual OpenGL rendering commands, we reimplement the three virtual functions *GLWidget::initializeGL()*, *QGLWidget::resizeGL()*, and *QGLWidget::paintGL()*.

We also need some member variables. *pMatrix* is a *QMatrix4x4* that keeps the projection part of the transformation pipeline. To manage the shaders, we will use a *QGLShaderProgram* we named *shaderProgram*. *vertices* is a *QVector* made of *QVector3Ds* that stores the triangle's vertices. Although the vertex shader will expect us to send homogeneous coordinates, we can use 3D vectors, because the OpenGL pipeline will automatically set the fourth coordinate to the default value of 1.

```
class GlWidget : public QGLWidget
{
    Q_OBJECT

public:
    GlWidget(QWidget *parent = 0);
    ~GlWidget();
    QSize sizeHint() const;

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

private:
    QMatrix4x4 pMatrix;
    QGLShaderProgram shaderProgram;
    QVector<QVector3D> vertices;
};
```

Now that we have defined our widget, we can finally talk about the implementation.

The constructor's initializer list calls *QGLWidget's* constructor passing a *QGLFormat* object. This can be used to set the capabilities of the OpenGL rendering context such as double buffering or multisampling. We are fine with the default values so we could as well have omitted the *QLFormat*. Qt will try to acquire a rendering context as close as possible to what we want.

Then we reimplement *QWidget::sizeHint()* to set a reasonable default size for the widget.

```
GlWidget::GlWidget(QWidget *parent)
    : QGLWidget(QGLFormat(/* Additional format options */), parent)
{
}

GlWidget::~GlWidget()
{
}

QSize GlWidget::sizeHint() const
{
    return QSize(640, 480);
}
```

The *QGLWidget::initializeGL()* method gets called once when the OpenGL context is created. We use this function to set the behavior of the rendering context and to build the shader programs.

If we want to render 3D images, we need to enable depth testing. This is one of the tests that can be performed during the per-sample-operations stage. It will cause OpenGL to only display the fragments nearest to the camera when primitives overlap. Although we do not need this capability since we only want to show a plane triangle, we will need this setting in our other examples. If you've omitted this statement, you might see objects in the back popping through objects in the front depending on the order the primitives are rendered. Deactivating this capability is useful if you want to draw an overlay image on top of the screen.

As an easy way to significantly improve the performance of a 3D application, we also enable

face culling. This tells OpenGL to only render primitives that show their front side. The front side is defined by the order of the triangle's vertices. You can tell what side of the triangle you are seeing by looking at its corners. If the triangle's corners are specified in a counterclockwise order, this means that the front of the triangle is the side facing you. For all triangles that are not facing the camera, the fragment processing stage can be omited.

Then we set the background color using *QGLWidget::qglClearColor()*. It is a function that calls OpenGL's *glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf apha)* but has the advantage of allowing any color Qt understands to be passed. The specified color will then be used in all subsequent calls to *glClear(GLbitfield mask)*.

In the following section we are setting up the shaders. We pass the source codes of the shaders to the QGLShaderProgram, compile and link them, and bind the program to the current OpenGL rendering context.

Shader programs need to be supplied as source codes. We can use *QGLShaderProgram::addShaderFromSourceFile()* to have Qt handle the compilation. This function compiles the source code as the specified shader type and adds it to the shader program. If an error occurs, the function returns *false*, and we can access the compilation errors and warnings using *QGLShaderProgram::log()*. Errors will be automatically printed to the standard error output if we run the program in debug mode.

After the compilation, we still need to link the programs using *QGLShaderProgram::link()*. We can again check for errors and access the errors and warnings using *QGLShaderProgram::log()*.

The shaders are then ready to be bound to the rendering context using *QGLShaderProgram::bind()*. Binding the program to the context means enabling it in the graphics pipeline. After this is done, every vertex that is passed to the graphics pipeline will be processed by these shaders until we call *QGLShaderProgram::release()* to disable them or a different shader program is bound.

Binding and releasing a program can be done several times during the rendering process, which means several vertex and fragment shaders can be used for different objects in the scene. We will therefore use these functions in the *QGLWidget::paintGL()* function.

Last but not least, we set up the triangles' vertices. Note that we've defined the triangle with the front side pointing to the positive z direction. Having face culling enabled, we can then see this object if we look at it from viewer positions with a z value greater than this object's z value.

```
void GlWidget::initializeGL()
{
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);

    qglClearColor(QColor(Qt::black));

    shaderProgram.addShaderFromSourceFile(QGLShader::Vertex, ":/vertexShader.vsh");
    shaderProgram.addShaderFromSourceFile(QGLShader::Fragment, ":/fragmentShader.fsh");
    shaderProgram.link();

    vertices << QVector3D(1, 0, -2) << QVector3D(0, 1, -2) << QVector3D(-1, 0, -2);
}
```

Now let's take a look at the shaders we will use in this example.

The vertex shader only calculates the final projection of each vertex by multiplying the vertex with the model-view-projection matrix.

It needs to read two input variables. The first input is the model-view-projection matrix. It is a 4x4 matrix, that only changes once per object and is therefore declared as a *uniform mat4*. We've named it *mvpMatrix*. The second variable is the actual vertex that the shader is processing. As the shader reads a new value every time it is executed, the vertex variable needs to be declared as an *attribute vec4*. We've named this variable *vertex*.

In the *main()* function, we simply calculate the resulting position that is sent to the rasterization stage using built in matrix vector multiplication.

```
uniform mat4 mvpMatrix;

in vec4 vertex;

void main(void)
{
    gl_Position = mvpMatrix * vertex;
}
```

The fragment shader simply displays a colored pixel for each fragment it is executed on.

The output of the fragment shader is the value written to the frame buffer. We called this variable *fragColor*. It is an instance of *vec4* with one element for the red, green, and blue color value, and one element for the alpha value.

We want to use the same plain color for each pixel. Therefore we declare an input variable called *color*, which is a *uniform vec4*.

The *main()* function then sets the built in *hl_FragColor* output variable to this value.

```
uniform vec4 color;

out vec4 fragColor;

void main(void)
{
    fragColor = color;
}
```

The reimplemented *QGLWidget::resizeGL()* method is called whenever the widget is resized. This is why we use this function to set up the projection matrix and the viewport.

After we had checked the widget's height to prevent a division by zero, we set it to a matrix that does the perspective projection. Luckily we do not have to calculate it ourselves. We can use one of the many useful methods of *QMatrix4x4*, namely *QMatrix4x4::perspective()*, which does exactly what we need. This method multiplies its *QMatrix4x4* instance with a projection matrix that is specified by the angle of the field of view, its aspect ratio and the clipping regions of the near and far planes. The matrix we get using this function resembles the projection of a camera that is sitting in the origin of the world coordinate system looking towards the world's negative z direction with the world's x axis pointing to the right side and the y axis pointing upwards. The fact that this function alters its instance explains the need to first initialize it to an identity matrix (a matrix that, if it is applied as a transformation, does not change a vector at all).

Next we set up the OpenGL viewport. The viewport defines the region of the widget that the result of the projection is mapped to. This mapping transforms the normalized coordinates on the aforementioned camera's film to pixel coordinates within the *QGLWidget*. To avoid distortion, the aspect ratio of the viewport should match the aspect ratio of the projection.

```
void GlWidget::resizeGL(int width, int height)
{
    if (height == 0) {
        height = 1;
    }

    pMatrix.setToIdentity();
    pMatrix.perspective(60.0, (float) width / (float) height, 0.001, 1000);

    glViewport(0, 0, width, height);
}
```

Finally, we have OpenGL draw the triangle in the *QGLWidget::paintGL()* method.

The first thing we do is clear the screen using *glClear(GLbitfield mask)*. If this OpenGL function is called with the *GL_COLOR_BUFFER_BIT* set, it fills the color buffer with the color set by *glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf aplha)*. Setting the *GL_DEPTH_BUFFER_BIT* tells OpenGL to clear the depth buffer, which is used for the depth test and stores the distance of rendered pixels. We usually need to clear both buffers, and therefore, we set both bits.

As we already know, the model-view-projection matrix that is used by the vertex shader is a concatenation of the model matrix, the view matrix and the projection matrix. Just like for the projection matrix, we also use the *QMatrix4x4* class to handle the other two transformations. Although we do not want to use them in this basic example, we already introduce them here to clarify their use. We use them to calculate the model-view-projection matrix, but leave them initialized to the identity matrix. This means we do not move or rotate the triangle's frame and also leave the camera unchanged, located in the origin of the world coordinate system.

The rendering can now be triggered by calling the OpenGL function *glDrawArrays(GLenum mode, GLint first, GLsizei count)*. But before we can do that, we need to bind the shaders and hand over all the uniforms and attributes they need.

In native OpenGL the programmer would first have to query the id (called *location*) of each input variable using the verbatim variable name as it is typed in the shader source code and then set its value using this id and a type OpenGL understands. *QGLShaderProgram* instead offers a huge set of overloaded functions for this purpose which allow you to address an input variable using either its *location* or its name. These functions can also automatically convert the variable type from Qt types to OpenGL types.

We set the uniform values for both shaders using *QGLShaderProgram::setUniformValue()* by passing its name. The vertex shader's uniform *Matrix* is calculated by multiplying its three components. The color of the triangle is set using a *QColor* instance that will automatically be converted to a *vec4* for us.

To tell OpenGL where to find the stream of vertices, we call QGLShaderProgram::setAttributeArray() and pass the QVector::constData() pointer. Setting attribute arrays works in the same way as setting uniform values, but there's one difference: we additionally

need to explicitly enable the attribute array using QGLShaderProgram::enableAttributeArray(). If we did not do this, OpenGL would assume that we've assigned a single value instead of an array.

Finally we call *glDrawArrays(GLenum mode, GLint first, GLsizei count)* to do the rendering. It is used to start rendering a sequence of geometry primitives using the current configuration. We pass *GL_TRIANGLES* as the first parameter to tell OpenGL that each of the three vertices form a triangle. The second parameter specifies the starting index within the attribute arrays and the third parameter is the number of indices to be rendered.

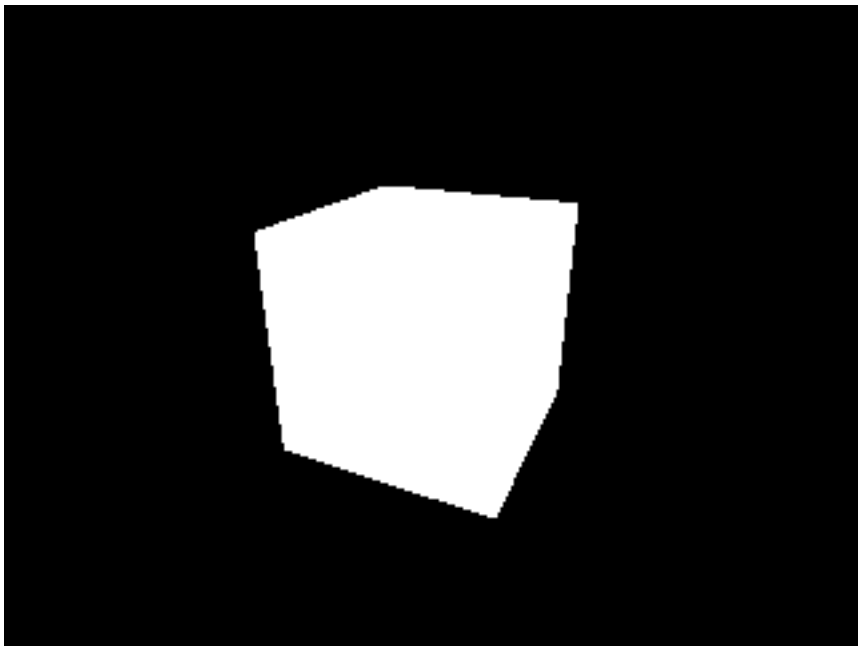
Note that if you later want to draw more than one object, you only need to repeat all of the steps (except for clearing the screen, of course) you took in this method for each new object.

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

    QMatrix4x4 mMatrix;
    QMatrix4x4 vMatrix;

    shaderProgram.bind();

    shaderProgram.setUniformValue("mvpMatrix", pMatrix * vMatrix * mMatrix);

    shaderProgram.setUniformValue("color", QColor(Qt::white));

    shaderProgram.setAttributeArray("vertex", vertices.constData());
    shaderProgram.enableAttributeArray("vertex");

    glDrawArrays(GL_TRIANGLES, 0, vertices.size());

    shaderProgram.disableAttributeArray("vertex");

    shaderProgram.release();
}
```

You should now see a white triangle on black background after compiling and running this program.

2.2 Rendering in 3D

A white triangle on black background is not very interesting and also not 3D, but now that we have a running basis, we can extend it to create a real 3D application. In this example, we will render a more complex object and implement the functionality for interactively exploring our scene.



Note: The source code related to this section is located in *examples/rendering-in-3d/* directory.

Just as with any *QWidget* subclass, we can use Qt's event system to handle user input. We want to be able to view the scene in the same way we would explore a globe. By dragging the mouse across the widget, we want to change the angle that we look from. The distance to the scene shall change if we turn the mouse's scroll wheel.

For this functionality, we reimplement *QWidget::mousePressEvent()*, *QWidget::mouseMoveEvent()*, and *QWidget::wheelEvent()*. The new member variables *alpha*, *beta* and *distance* hold the parameters of the view point, and *lastMousePosition* helps us track mouse movement.

```
class GlWidget : public QGLWidget
{
    ...

protected:
    ...

    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void wheelEvent(QWheelEvent *event);

private:
    ...

    double alpha;
    double beta;
    double distance;
    QPoint lastMousePosition;
};
```

The most important new thing in this example is the employment of the view matrix. Again, we do not calculate this matrix ourselves but use the *QMatrix4x4::lookAt()* function to obtain

this matrix. This function takes the position of the viewer, the point the viewer is looking at and a vector that defines the up direction. We want the viewer to look at the world's origin and start with a position that is located at a certain distance (*distance*) along the z axis with the up direction being the y axis. We then rotate these two vertices using a transformation matrix. First we rotate them (and their coordinate system) by the *alpha* angle around their new rotated x axis, which tilts the camera. Note that you can also illustrate the transformation the other way around: first we rotate the vertices by the *beta* angle around the world's x axis and then we rotate them by the *alpha* angle around the world's y axis.

```
void GlWidget::paintGL()
{
    ...
    QMatrix4x4 cameraTransformation;
    cameraTransformation.rotate(alpha, 0, 1, 0);
    cameraTransformation.rotate(beta, 1, 0, 0);

    QVector3D cameraPosition = cameraTransformation * QVector3D(0, 0, distance);
    QVector3D cameraUpDirection = cameraTransformation * QVector3D(0, 1, 0);

    vMatrix.lookAt(cameraPosition, QVector3D(0, 0, 0), cameraUpDirection);
    ...
}
```

These three parameters need to be initialized in the constructor and, to account for the user's input, we then change them in the corresponding event handlers.

```
GlWidget::GlWidget(QWidget *parent)
    : QGLWidget(QGLFormat(/* Additional format options */), parent)
{
    alpha = 25;
    beta = -25;
    distance = 2.5;
}
```

In the *QWidget::mousePressEvent()*, we store the mouse pointer's initial position to be able to track the movement. In the *QGLWidget::mouseMoveEvent()*, we calculate the pointers change and adapt the angles *alpha* and *beta*. Since the view point's parameters have changed, we then call *QGLWidget::updateGL()* to trigger an update of the rendering context.

```
void GlWidget::mousePressEvent(QMouseEvent *event)
{
    lastMousePosition = event->pos();

    event->accept();
}

void GlWidget::mouseMoveEvent(QMouseEvent *event)
{
    int deltaX = event->x() - lastMousePosition.x();
    int deltaY = event->y() - lastMousePosition.y();

    if (event->buttons() & Qt::LeftButton) {
        alpha -= deltaX;
        while (alpha < 0) {
            alpha += 360;
        }
```

```
        while (alpha >= 360) {
            alpha -= 360;
        }

        beta -= deltaY;
        if (beta < -90) {
            beta = -90;
        }
        if (beta > 90) {
            beta = 90;
        }

        updateGL();
    }

    lastMousePosition = event->pos();

    event->accept();
}
```

In the *QGLWidget::wheelEvent()*, we either increase or decrease the viewers distance by 10% and update the rendering again.

```
void GlWidget::wheelEvent(QWheelEvent *event)
{
    int delta = event->delta();

    if (event->orientation() == Qt::Vertical) {
        if (delta < 0) {
            distance *= 1.1;
        } else if (delta > 0) {
            distance *= 0.9;
        }

        updateGL();
    }

    event->accept();
}
```

In order to finish this example, we only need to change our list of vertices to form a cube.

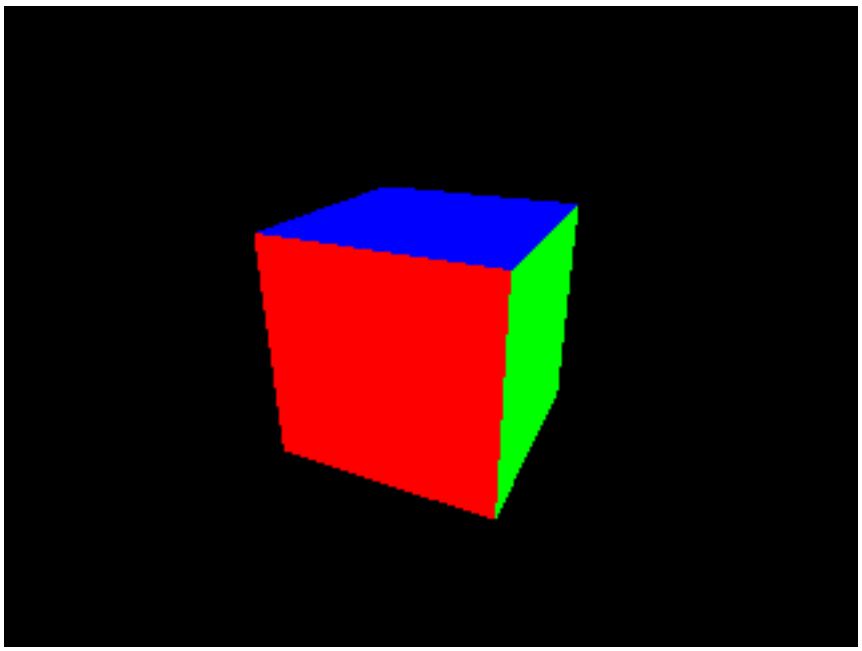
```
void GlWidget::initializeGL()
{
        ...
        vertices << QVector3D(-0.5, -0.5,  0.5) << QVector3D( 0.5, -0.5,  0.5) << QVecto
             << QVector3D( 0.5,  0.5,  0.5) << QVector3D(-0.5,  0.5,  0.5) << QVector3D
             << QVector3D( 0.5, -0.5, -0.5) << QVector3D(-0.5, -0.5, -0.5) << QVector3D
             << QVector3D(-0.5,  0.5, -0.5) << QVector3D( 0.5,  0.5, -0.5) << QVector3D
             << QVector3D(-0.5, -0.5, -0.5) << QVector3D(-0.5, -0.5,  0.5) << QVector3D
             << QVector3D(-0.5,  0.5,  0.5) << QVector3D(-0.5,  0.5, -0.5) << QVector3D
             << QVector3D( 0.5, -0.5,  0.5) << QVector3D( 0.5, -0.5, -0.5) << QVector3D
             << QVector3D( 0.5,  0.5, -0.5) << QVector3D( 0.5,  0.5,  0.5) << QVector3D
             << QVector3D(-0.5,  0.5,  0.5) << QVector3D( 0.5,  0.5,  0.5) << QVector3D
             << QVector3D( 0.5,  0.5, -0.5) << QVector3D(-0.5,  0.5, -0.5) << QVector3D
             << QVector3D(-0.5, -0.5, -0.5) << QVector3D( 0.5, -0.5, -0.5) << QVector3D
             << QVector3D( 0.5, -0.5,  0.5) << QVector3D(-0.5, -0.5,  0.5) << QVector3D
```

```
}
```

If you now compile and run this program, you will see a white cube that can be rotated using the mouse. Since each of its six sides is painted in the same plane color, depth is not very visible. We will work on this in the next example.

2.3 Coloring

In this example, we want to color each side of the cube in different colors to enhance the illusion of three dimensionality. To archive this, we will extend our shaders in a way that will allow us to specify a single color for each vertex and use the interpolation of varyings to generate the fragment's colors. This example will show you how to communicate data from the vertex shader over to the fragment shader.



Note: The source code related to this section is located in *examples/coloring/* directory

To tell the shaders about the colors, we specify a color value for each vertex as an attribute array for the vertex shader. So on each run of the shader, it will read a new value for both the vertex attribute and the color attribute.

As the fragment's color eventually has to be set in the fragment shader and not in the vertex shader, we pass the color value over to it. To do this, we need to declare an equally named varying in both shaders. We called this varying *varyingColor*. If the fragment shader is now run for each fragment between the three vertices of a triangle, the value read by the shader is calculated by an interpolation of the three corners' values. This means that if we specify the same color for the three vertices of a triangle, OpenGL will paint a plane colored triangle. If we specify different colors, OpenGL will smoothly blend between those values.

In the vertex shader's main function, we only need to set the varying to the color value.

```
uniform mat4 mvpMatrix;

in vec4 vertex;
in vec4 color;

out vec4 varyingColor;

void main(void)
{
    varyingColor = color;
    gl_Position = mvpMatrix * vertex;
}
```

In the fragment shader's main function, we set the *gl_FragColor* variable to the color received.

```
in vec4 varyingColor;

out vec4 fragColor;

void main(void)
{
    fragColor = varyingColor;
}
```

Of course we still need to imploy a new structure to store the color values and send them to the shaders in the *QGLWidget::paintGL()* method. But this should be very straightforward as we have already done all of this for the *vertices* attribute array in just the same manner.

```
// glwidget.h

class GlWidget : public QGLWidget
{
        ...

        private:
            ...

            QVector<QVector3D> colors;
};

// glwidget.cpp

void GlWidget::paintGL()
{
    ...

    shaderProgram.bind();

    shaderProgram.setUniformValue("mvpMatrix", pMatrix * vMatrix * mMatrix);

    shaderProgram.setAttributeArray("vertex", vertices.constData());
    shaderProgram.enableAttributeArray("vertex");

    shaderProgram.setAttributeArray("color", colors.constData());
    shaderProgram.enableAttributeArray("color");

    glDrawArrays(GL_TRIANGLES, 0, vertices.size());
```

```
    shaderProgram.disableAttributeArray("vertex");

    shaderProgram.disableAttributeArray("color");

    shaderProgram.release();
}
```

There is only one little inconvenience when switching from a color uniform to a color attribute array. Unfortunately *QGLShaderProgram::setAttributeArray()* does not support the *QColor* type, so we need to store the colors as a *QVector3D* (or a *QVector4D*, if you want to set the alpha value to change the opacity). Valid color values range from 0 to 1. As we want to color each of the cube's faces in a plain color, we set the color value of each face's vertices to the same value.

```
void GlWidget::initializeGL()
{
    ...

    colors << QVector3D(1, 0, 0) << QVector3D(1, 0, 0) << QVector3D(1, 0, 0) // Front
           << QVector3D(1, 0, 0) << QVector3D(1, 0, 0) << QVector3D(1, 0, 0)
           << QVector3D(1, 0, 0) << QVector3D(1, 0, 0) << QVector3D(1, 0, 0) // Back
           << QVector3D(1, 0, 0) << QVector3D(1, 0, 0) << QVector3D(1, 0, 0)
           << QVector3D(0, 1, 0) << QVector3D(0, 1, 0) << QVector3D(0, 1, 0) // Left
           << QVector3D(0, 1, 0) << QVector3D(0, 1, 0) << QVector3D(0, 1, 0)
           << QVector3D(0, 1, 0) << QVector3D(0, 1, 0) << QVector3D(0, 1, 0) // Right
           << QVector3D(0, 1, 0) << QVector3D(0, 1, 0) << QVector3D(0, 1, 0)
           << QVector3D(0, 0, 1) << QVector3D(0, 0, 1) << QVector3D(0, 0, 1) // Top
           << QVector3D(0, 0, 1) << QVector3D(0, 0, 1) << QVector3D(0, 0, 1)
           << QVector3D(0, 0, 1) << QVector3D(0, 0, 1) << QVector3D(0, 0, 1) // Bottom
           << QVector3D(0, 0, 1) << QVector3D(0, 0, 1) << QVector3D(0, 0, 1);
}
```

Our cube now has its six sides colored differently.

2.4 Texture Mapping

Texture mapping is a very important concept in 3D computer graphics. It is the application of images on top of a model's surfaces and is essential for creating a nice 3D scene.

You can do more with textures than just mapping them to a surface. Essentially a texture is a two dimensional array containing color values so not only can you pass colors to your shaders, but an array of any data you want. However, in this example we will use a classic 2D texture to map an image on top of the cube we created in the previous examples.

Note: The source code related to this section is located in *examples/texture-mapping/* directory

In order to map a texture to a primitive, we have to specify so-called *texture coordinates* that tell OpenGL which image coordinate is to be pinned to which vertex. Texture coordinates are instances of *vec2* that are normalized to a range between 0 and 1. The origin of the texture coordinate system is in the lower left of an image, having the first axis pointing to the right side and the second axis pointing upwards (i.e. the lower left corner of an image is at *(0, 0)* and the upper right corner is at *(1, 1)*. Coordinate values higher than 1 are also allowed, causing the texture to wrap around by default.

The textures themselves are OpenGL objects stored in the graphics card's memory. They are created using *glGenTextures(GLsizei n, GLuint *texture)* and deleted again with a call to *glDelereTextures(GLsizei n, const GLuint *texture)*. To identify textures, each texture is assigned a texture ID during its creation. As with shader programs, they need to be bound to *glBindTexture(GLenum target, GLuint texture)* before they can be configured and filled with data. We can use Qt's *QGLWidget::bindTexture()* to create the texture object. Normally we would have to make sure that the image data is in a particular format, according to the configuration of the texture object, but luckily *QGLWidget::bindTexture()* also takes care of that.

OpenGL allows us to have several textures accessible to the shaders at the same time. For this purpose, OpenGL uses so-called *texture units*. So before we can use a texture, we need to bind it to one of the texture units identified by the enum *GL_TEXTUREi* (with i ranging from 0 to *GL_MAX_COMBINED_TEXTURE_UNITS -1*). To do this, we call *glActiveTexture(GLenum texture)* and bind the texture using *glBindTexture(GLenum target, GLuint texture)*. Because we also need to call *glBindTexture(GLenum target, GLuint texture)* if we want to add new textures or modify them, and binding a texture overwrites the the current active texture unit, you should set the active texture unit to an invalid unit after setting it by calling *glActiveTexture(0)*. This way several texture units can be configured at the same time. Note that texture units need to be used in an ascending order beginning with *GL_TEXTURE0*.

To access a texture in a shader to actually render it, we use the *texture2D(sampler2D sampler,*

vec2 coord) function to query the color value at a certain texture coordinate. This function reads two parameters. The first parameter is of the type *sampler2D* and it refers to a texture unit. The second parameter is the texture coordinate that we want to access. To read from the texture unit *i* denoted by the enum *GL_TEXTUREi*, we have to pass the *GLuint i* as the uniform value.

With all of this theory we are now able to make our cube textured.

We replace the *vec4* color attribute and the corresponding varying with a *vec2* variable for the texture coordinates and forward this value to the fragment shader.

```
uniform mat4 mvpMatrix;

in vec4 vertex;
in vec2 textureCoordinate;

out vec2 varyingTextureCoordinate;

void main(void)
{
    varyingTextureCoordinate = textureCoordinate;
    gl_Position = mvpMatrix * vertex;
}
```

In the fragment shader, we use *texture2D(sampler2D sampler, vec2 coord)* to look up the right color value. The uniform *texture* of the type *sampler2D* chooses the texture unit and we use the interpolated values coming from the vertex shader for the texture coordinates.

```
uniform sampler2D texture;

in vec2 varyingTextureCoordinate;

out vec4 fragColor;

void main(void)
{
    fragColor = texture2D(texture, varyingTextureCoordinate);
}
```

In the *GlWidget* class declaration, we replace the previously used *colors* member with a *QVector* made of *QVector2Ds* for the texture coordinates and add a member variable to hold the texture object ID.

```
class GlWidget : public QGLWidget
{
    ...

    private:
        ...

        QVector<QVector2D> textureCoordinates;
        GLuint texture;
        ...

};
```

In the *QGLWidget::initializeGL()* reimplementation, we set up the texture coordinates and also

create the texture object. Each side will be covered with the whole square image that is contained in our resource file.

```
void GlWidget::initializeGL()
{
    ...
    textureCoordinates << QVector2D(0, 0) << QVector2D(1, 0) << QVector2D(1, 1) // Front
                       << QVector2D(1, 1) << QVector2D(0, 1) << QVector2D(0, 0)
                       << QVector2D(0, 0) << QVector2D(1, 0) << QVector2D(1, 1) // Back
                       << QVector2D(1, 1) << QVector2D(0, 1) << QVector2D(0, 0)
                       << QVector2D(0, 0) << QVector2D(1, 0) << QVector2D(1, 1) // Left
                       << QVector2D(1, 1) << QVector2D(0, 1) << QVector2D(0, 0)
                       << QVector2D(0, 0) << QVector2D(1, 0) << QVector2D(1, 1) // Right
                       << QVector2D(1, 1) << QVector2D(0, 1) << QVector2D(0, 0)
                       << QVector2D(0, 0) << QVector2D(1, 0) << QVector2D(1, 1) // Top
                       << QVector2D(1, 1) << QVector2D(0, 1) << QVector2D(0, 0)
                       << QVector2D(0, 0) << QVector2D(1, 0) << QVector2D(1, 1) // Bottom
                       << QVector2D(1, 1) << QVector2D(0, 1) << QVector2D(0, 0);

    texture = bindTexture(QPixmap(":/texture.png"));
}
```

In the *QGLWidget::paintGL()* method, we set the fragment shader's sampler2D uniform to the first texture unit. Then we activate that unit, bind our texture object to it and after that deactivate it again to prevent us from accidentally overwriting this setting. And instead of passing the color attribute array, we pass the array containing the texture coordinates.

```
void GlWidget::paintGL()
{
    ...

    shaderProgram.bind();

    shaderProgram.setUniformValue("mvpMatrix", pMatrix * vMatrix * mMatrix);

    shaderProgram.setUniformValue("texture", 0);

    //glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture);
    //glActiveTexture(0);

    shaderProgram.setAttributeArray("vertex", vertices.constData());
    shaderProgram.enableAttributeArray("vertex");

    shaderProgram.setAttributeArray("textureCoordinate", textureCoordinates.constData());
    shaderProgram.enableAttributeArray("textureCoordinate");

    glDrawArrays(GL_TRIANGLES, 0, vertices.size());

    shaderProgram.disableAttributeArray("vertex");

    shaderProgram.disableAttributeArray("textureCoordinate");

    shaderProgram.release();
}
```

Our cube is now textured.

Note: The Windows OpenGL header file only includes functionality up to OpenGL version 1.1 and assumes that the programmer will obtain additional functionality on his own. This includes OpenGL API function calls as well as enums. The reason is that, because different OpenGL libraries exist, the programmer should request the library's function entry points at runtime.

Qt only defines the functionality required by its own OpenGL-related classes. *glActiveTexture(GLenum texture)* as well as the *GL_TEXTUREi* enums do not belong to this subset.

Several utility libraries exist to ease the definition of these functions (e.g. GLEW, GLEE, etc). We will define *glActiveTexture(GLenum texture)* and *GL_TEXTUREi* manually.

First we include the *glext.h* header file to set the missing enums and a few typedefs which will help us keep the code readable (since the version shipped with your compiler might be outdated, you may need to get the latest version from the OpenGL homepage[1]. Next we declare the function pointer, which we will use to call *glActiveTexture(GLenum texture)* using the included typedefs. To avoid confusing the linker, we use a different name than *glActiveTexture* and define a pre-processor macro which replaces calls to *glActiveTexture(GLenum texture)* with our own function:

```
#ifdef WIN32
    #include <GL/glext.h>
    PFNGLACTIVETEXTUREPROC pGlActiveTexture = NULL;
    #define glActiveTexture pGlActiveTexture
#endif //WIN32
```

In the *GlWidget::initializeGL()* function, we request this pointer using *PROC WINAPI wglGetProcAddress(LPCSTR lpszProc)*. This function reads the OpenGL API function's name and returns a pointer which we need to cast to the right type:

```
void GlWidget::initializeGL()
{
    ...

    #ifdef WIN32
        glActiveTexture = (PFNGLACTIVETEXTUREPROC) wglGetProcAddress((LPCSTR) "glActiveT
    #endif
    ...
}
```

glActiveTexture() and *GL_TEXTUREi* can then be used on Windows.

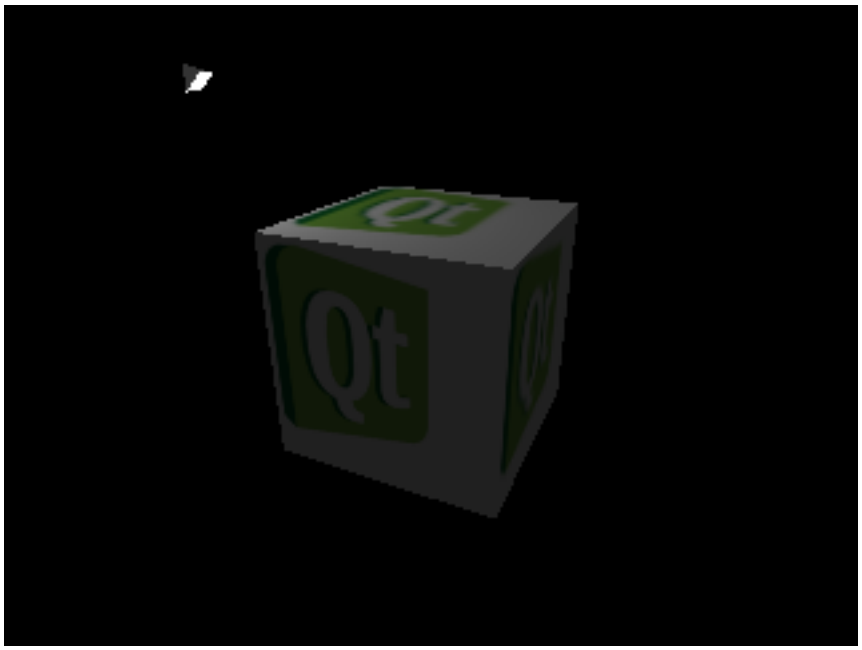
2.5 Lighting

The ability to write your own shader programs gives you the power to set up the kind of lighting effect that best suits your needs. This may range from very basic and time saving approaches to high quality ray tracing algorithms.

[1] http://www.opengl.org

In this chapter, we will implement a technique called *Phong shading*, which is a popular baseline shading method for many rendering applications. For each pixel on the surface of an object, we will calculate the color intensity based on the position and color of the light source as well as the object's texture and its material properties.

To show the results, we will display the cube with a light source circling above it. The light source will be marked by a pyramid which we will render using the per-vertex color shader of one of the previous examples. So in this example, you will also see how to render a scene with multiple objects and different shader programs.



Note: The source code related to this section is located in *examples/lighting/* directory

---

Because we use two different objects and two different shader programs, we added prefixes to the names. The cube is rendered using the *lightingShaderProgram*, for which we need an additional storage that keeps the surface normal of each vertex (i.e. the vector, that is perpendicular to the surface and has the size 1). The spotlight, on the other hand, is rendered using the *coloringShaderProgram*, which consists of a shader we developed earlier in this tutorial.

To track the position of the light source, we introduced a new member variable that holds its rotation. This value is periodically increased in the *timeout()* slot.

```
class GlWidget : public QGLWidget
{
        ...

        private:
            QGLShaderProgram lightingShaderProgram;
            QVector<QVector3D> cubeVertices;
            QVector<QVector3D> cubeNormals;
            QVector<QVector2D> cubeTextureCoordinates;
            GLuint cubeTexture;
            QGLShaderProgram coloringShaderProgram;
            QVector<QVector3D> spotlightVertices;
```

```
            QVector<QVector3D> spotlightColors;
            double lightAngle;
        ...

        private Q_SLOTS:
            void timeout();
};
```

The Phong reflection model assumes that the light reflected off an object (i.e. what you actually see) consists of three components: diffuse reflection of rough surfaces, specular highlights of glossy surfaces and an ambient term that sums up the small amounts of light that get scattered about the entire scene.

For each light source in the scene, we define i_d and i_s as the intensities (RGB values) of the diffuse and the specular components. i_a is defined as the ambient lighting component.

For each kind of surface (whether glossy, flat etc), we define the following parameters: k_d and k_s set the ratio of reflection of the diffuse and specular component, k_a sets the ratio of the reflection of the ambient term respectively and α is a shininess constant that controls the size of the specular highlights.

The equation for computing the illumination of each surface point (fragment) is:

$$I_p = k_a i_a + \sum_{m \in lights} (k_d(\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s(\hat{R}_m \cdot \hat{V})^\alpha i_{m,s})$$

$\hat{L}_m$ is the normalized direction vector pointing from the fragment to the light source, $\hat{N}$ is the surface normal of this fragment, $\hat{R}_m$ is the direction of the light reflected at this point and $\hat{V}$ points from the fragment towards the viewer of the scene.

To obtain the vectors mentioned above, we calculate them for each vertex in the vertex shader and tell OpenGL to pass them as interpolated values to the fragment shader. In the fragment shader, we finally set the illumination of each point and combine it with the color value of the texture.

So in addition to passing vertex positions and the model-view-projection matrix to get the fragment's position, we also need to pass the surface normal of each vertex. To calculate the transformed cube's $\hat{L}_m$ and $\hat{V}$, we need to know the model-view part of the transformation. To calculate the transformed $\hat{N}$, we need to apply a matrix that transforms the surface normals. This extra matrix is needed because we only want the normals to be rotated according to the model-view matrix, but not to be translated.

This is the vertex shader's source code:

```
uniform mat4 mvpMatrix;
uniform mat4 mvMatrix;
uniform mat3 normalMatrix;
uniform vec3 lightPosition;

in vec4 vertex;
in vec3 normal;
in vec2 textureCoordinate;

out vec3 varyingNormal;
out vec3 varyingLightDirection;
```

```
out vec3 varyingViewerDirection;
out vec2 varyingTextureCoordinate;

void main(void)
{
    vec4 eyeVertex = mvMatrix * vertex;
    eyeVertex /= eyeVertex.w;
    varyingNormal = normalMatrix * normal;
    varyingLightDirection = lightPosition - eyeVertex.xyz;
    varyingViewerDirection = -eyeVertex.xyz;
    varyingTextureCoordinate = textureCoordinate;
    gl_Position = mvpMatrix * vertex;
}
```

The fragment shader is supplied with the light source's and the material's properties and the geometry data calculated by the vertex shader. It then sets the fragment's color value according to the above formula.

This is the fragment shaders source code:

```
uniform vec4 ambientColor;
uniform vec4 diffuseColor;
uniform vec4 specularColor;
uniform float ambientReflection;
uniform float diffuseReflection;
uniform float specularReflection;
uniform float shininess;
uniform sampler2D texture;

in vec3 varyingNormal;
in vec3 varyingLightDirection;
in vec3 varyingViewerDirection;
in vec2 varyingTextureCoordinate;

out vec4 fragColor;

void main(void)
{
    vec3 normal = normalize(varyingNormal);
    vec3 lightDirection = normalize(varyingLightDirection);
    vec3 viewerDirection = normalize(varyingViewerDirection);
    vec4 ambientIllumination = ambientReflection * ambientColor;
    vec4 diffuseIllumination = diffuseReflection * max(0.0, dot(lightDirection, normal))
    vec4 specularIllumination = specularReflection * pow(max(0.0,
                                                              dot(-reflect(lightDirectior
                                                              ), shininess) * specularCol
    fragColor = texture2D(texture,
                          varyingTextureCoordinate) * (ambientIllumination + diffuseIllu
                          ) + specularIllumination;
}
```

In the *GLWidget::initiliazeGL()* method, we set up both shaders and prepare the attribute arrays of the cube and the spotlight. The only thing new here is the *QVector* made of *QVector3Ds* that stores the surface normal of each of the cube's vertices.

```
void GlWidget::initializeGL()
{
    ...

    lightingShaderProgram.addShaderFromSourceFile(QGLShader::Vertex, ":/lightingVertexSh
    lightingShaderProgram.addShaderFromSourceFile(QGLShader::Fragment, ":/lightingFragme
    lightingShaderProgram.link();

    cubeVertices << QVector3D(-0.5, -0.5,  0.5) << QVector3D( 0.5, -0.5,  0.5) << QVecto
                 << QVector3D( 0.5,  0.5,  0.5) << QVector3D(-0.5,  0.5,  0.5) << QVecto
                 << QVector3D( 0.5, -0.5, -0.5) << QVector3D(-0.5, -0.5, -0.5) << QVecto
                 << QVector3D(-0.5,  0.5, -0.5) << QVector3D( 0.5,  0.5, -0.5) << QVecto
                 << QVector3D(-0.5, -0.5, -0.5) << QVector3D(-0.5, -0.5,  0.5) << QVecto
                 << QVector3D(-0.5,  0.5,  0.5) << QVector3D(-0.5,  0.5, -0.5) << QVecto
                 << QVector3D( 0.5, -0.5,  0.5) << QVector3D( 0.5, -0.5, -0.5) << QVecto
                 << QVector3D( 0.5,  0.5, -0.5) << QVector3D( 0.5,  0.5,  0.5) << QVecto
                 << QVector3D(-0.5,  0.5,  0.5) << QVector3D( 0.5,  0.5,  0.5) << QVecto
                 << QVector3D( 0.5,  0.5, -0.5) << QVector3D(-0.5,  0.5, -0.5) << QVecto
                 << QVector3D(-0.5, -0.5, -0.5) << QVector3D( 0.5, -0.5, -0.5) << QVecto
                 << QVector3D( 0.5, -0.5,  0.5) << QVector3D(-0.5, -0.5,  0.5) << QVecto
    cubeNormals << QVector3D( 0,  0,  1) << QVector3D( 0,  0,  1) << QVector3D( 0,  0,
                << QVector3D( 0,  0,  1) << QVector3D( 0,  0,  1) << QVector3D( 0,  0,
                << QVector3D( 0,  0, -1) << QVector3D( 0,  0, -1) << QVector3D( 0,  0, -
                << QVector3D( 0,  0, -1) << QVector3D( 0,  0, -1) << QVector3D( 0,  0, -
                << QVector3D(-1,  0,  0) << QVector3D(-1,  0,  0) << QVector3D(-1,  0,
                << QVector3D(-1,  0,  0) << QVector3D(-1,  0,  0) << QVector3D(-1,  0,
                << QVector3D( 1,  0,  0) << QVector3D( 1,  0,  0) << QVector3D( 1,  0,
                << QVector3D( 1,  0,  0) << QVector3D( 1,  0,  0) << QVector3D( 1,  0,
                << QVector3D( 0,  1,  0) << QVector3D( 0,  1,  0) << QVector3D( 0,  1,
                << QVector3D( 0,  1,  0) << QVector3D( 0,  1,  0) << QVector3D( 0,  1,
                << QVector3D( 0, -1,  0) << QVector3D( 0, -1,  0) << QVector3D( 0, -1,
                << QVector3D( 0, -1,  0) << QVector3D( 0, -1,  0) << QVector3D( 0, -1,
    cubeTextureCoordinates << QVector2D(0, 0) << QVector2D(1, 0) << QVector2D(1, 1) // F
                           << QVector2D(1, 1) << QVector2D(0, 1) << QVector2D(0, 0)
                           << QVector2D(0, 0) << QVector2D(1, 0) << QVector2D(1, 1) // F
                           << QVector2D(1, 1) << QVector2D(0, 1) << QVector2D(0, 0)
                           << QVector2D(0, 0) << QVector2D(1, 0) << QVector2D(1, 1) // B
                           << QVector2D(1, 1) << QVector2D(0, 1) << QVector2D(0, 0)
                           << QVector2D(0, 0) << QVector2D(1, 0) << QVector2D(1, 1) // F
                           << QVector2D(1, 1) << QVector2D(0, 1) << QVector2D(0, 0)
                           << QVector2D(0, 0) << QVector2D(1, 0) << QVector2D(1, 1) // T
                           << QVector2D(1, 1) << QVector2D(0, 1) << QVector2D(0, 0)
                           << QVector2D(0, 0) << QVector2D(1, 0) << QVector2D(1, 1) // F
                           << QVector2D(1, 1) << QVector2D(0, 1) << QVector2D(0, 0);

    cubeTexture = bindTexture(QPixmap(":/cubeTexture.png"));

    coloringShaderProgram.addShaderFromSourceFile(QGLShader::Vertex, ":/coloringVertexSh
    coloringShaderProgram.addShaderFromSourceFile(QGLShader::Fragment, ":/coloringFragme
    coloringShaderProgram.link();

    spotlightVertices << QVector3D(   0,    1,    0) << QVector3D(-0.5,    0,  0.5) << Q
                      << QVector3D(   0,    1,    0) << QVector3D( 0.5,    0, -0.5) << Q
                      << QVector3D(   0,    1,    0) << QVector3D(-0.5,    0, -0.5) << Q
                      << QVector3D(   0,    1,    0) << QVector3D( 0.5,    0,  0.5) << Q
                      << QVector3D(-0.5,    0, -0.5) << QVector3D( 0.5,    0, -0.5) << Q
                      << QVector3D( 0.5,    0,  0.5) << QVector3D(-0.5,    0,  0.5) << Q
```

```
    spotlightColors << QVector3D(0.2, 0.2, 0.2) << QVector3D(0.2, 0.2, 0.2) << QVector3D
                    << QVector3D(0.2, 0.2, 0.2) << QVector3D(0.2, 0.2, 0.2) << QVector3D
                    << QVector3D(0.2, 0.2, 0.2) << QVector3D(0.2, 0.2, 0.2) << QVector3D
                    << QVector3D(0.2, 0.2, 0.2) << QVector3D(0.2, 0.2, 0.2) << QVector3D
                    << QVector3D(  1,   1,   1) << QVector3D(  1,   1,   1) << QVector3D
                    << QVector3D(  1,   1,   1) << QVector3D(  1,   1,   1) << QVector3D
}
```

After clearing the screen and calculating the view matrix (which is the same for both objects) in the *GlWidget::painGL()* method, we first render the cube using the lighting shaders and then we render the spotlight using the coloring shader.

Because we want to keep the cube's origin aligned with the world's origin, we leave the model matrix *mMatrix* set to an identity matrix. Then we calculate the model-view matrix, which we also need to send to the lighting vertex shader, and extract the normal matrix with Qt's *QMatrix4x4::normal()* method. As we have already stated, this matrix will transform the surface normals of our cube from model coordinates into viewer coordinates. After that, we calculate the position of the light source in world coordinates according to the angle.

We can now render the cube. We bind the lighting shader program, set the uniforms and texture units, set and enable the attribute arrays, trigger the rendering, and afterwards disable the attribute arrays and release the program. For the light source's and the material's properties, we set values that give us a glossy looking surface.

Next we render the spotlight.

Because we want to move the spotlight to the same place as the light source, we need to modify its model matrix. First we restore the identity matrix (actually we did not modify the model matrix before so it still is set to the identity matrix anyway). Then we move the spotlight to the light sources position. Now we still want to rotate it since it looks nicer if it faces our cube. We therefore apply two rotation matrices on top. Because the pyramid that represents our lightspot is still to big to fit into our scene nicely, we scale it down to a tenth of its original size.

Now we follow the usual rendering procedure again, this time using the *coloringShaderProgram* and the spotlight data. Thanks to depth testing, the new object will be integrated seamlessly into our existing scene.

```
void GlWidget::paintGL()
{
    ...

    mMatrix.setToIdentity();

    QMatrix4x4 mvMatrix;
    mvMatrix = vMatrix * mMatrix;

    QMatrix3x3 normalMatrix;
    normalMatrix = mvMatrix.normalMatrix();

    QMatrix4x4 lightTransformation;
    lightTransformation.rotate(lightAngle, 0, 1, 0);

    QVector3D lightPosition = lightTransformation * QVector3D(0, 1, 1);

    lightingShaderProgram.bind();
```

```
    lightingShaderProgram.setUniformValue("mvpMatrix", pMatrix * mvMatrix);
    lightingShaderProgram.setUniformValue("mvMatrix", mvMatrix);
    lightingShaderProgram.setUniformValue("normalMatrix", normalMatrix);
    lightingShaderProgram.setUniformValue("lightPosition", vMatrix * lightPosition);

    lightingShaderProgram.setUniformValue("ambientColor", QColor(32, 32, 32));
    lightingShaderProgram.setUniformValue("diffuseColor", QColor(128, 128, 128));
    lightingShaderProgram.setUniformValue("specularColor", QColor(255, 255, 255));
    lightingShaderProgram.setUniformValue("ambientReflection", (GLfloat) 1.0);
    lightingShaderProgram.setUniformValue("diffuseReflection", (GLfloat) 1.0);
    lightingShaderProgram.setUniformValue("specularReflection", (GLfloat) 1.0);
    lightingShaderProgram.setUniformValue("shininess", (GLfloat) 100.0);
    lightingShaderProgram.setUniformValue("texture", 0);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, cubeTexture);
    glActiveTexture(0);

    lightingShaderProgram.setAttributeArray("vertex", cubeVertices.constData());
    lightingShaderProgram.enableAttributeArray("vertex");
    lightingShaderProgram.setAttributeArray("normal", cubeNormals.constData());
    lightingShaderProgram.enableAttributeArray("normal");
    lightingShaderProgram.setAttributeArray("textureCoordinate", cubeTextureCoordinates.
    lightingShaderProgram.enableAttributeArray("textureCoordinate");

    glDrawArrays(GL_TRIANGLES, 0, cubeVertices.size());

    lightingShaderProgram.disableAttributeArray("vertex");
    lightingShaderProgram.disableAttributeArray("normal");
    lightingShaderProgram.disableAttributeArray("textureCoordinate");

    lightingShaderProgram.release();

    mMatrix.setToIdentity();
    mMatrix.translate(lightPosition);
    mMatrix.rotate(lightAngle, 0, 1, 0);
    mMatrix.rotate(45, 1, 0, 0);
    mMatrix.scale(0.1);

    coloringShaderProgram.bind();

    coloringShaderProgram.setUniformValue("mvpMatrix", pMatrix * vMatrix * mMatrix);

    coloringShaderProgram.setAttributeArray("vertex", spotlightVertices.constData());
    coloringShaderProgram.enableAttributeArray("vertex");

    coloringShaderProgram.setAttributeArray("color", spotlightColors.constData());
    coloringShaderProgram.enableAttributeArray("color");

    glDrawArrays(GL_TRIANGLES, 0, spotlightVertices.size());

    coloringShaderProgram.disableAttributeArray("vertex");

    coloringShaderProgram.disableAttributeArray("color");

    coloringShaderProgram.release();
}
```

The last thing left to do is to initialize the light source's position and set up the timer. We tell the timer to periodically invoke the *timout()* slot.

```
GlWidget::GlWidget(QWidget *parent)
    : QGLWidget(QGLFormat(/* Additional format options */), parent)
{
    ...

    lightAngle = 0;

    QTimer *timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(timeout()));
    timer->start(20);
}
```

In this slot we update the angle of the light source's circulation and update the screen. We also remove the calls to *QGLWidget::updateGL()* in the event handlers.

```
void GlWidget::timeout()
{
    lightAngle += 1;
    while (lightAngle >= 360) {
        lightAngle -= 360;
    }

    updateGL();
}
```

Now we are finished with the implementation. If you build and run the program, you will see a lit, textured cube.

2.6 Buffer Object

Up till now, we have transferred all the objects' per-vertex data from the computer's RAM via the memory bus and the AGP bus to the graphics card whenever we wanted to re-render the scene. Obviously this is not very efficient and imposes a significant performance penalty - especially when handling large datasets. In this example, we will solve this problem by adding *vertex buffer objects* to the lighting example.

Note: The source code related to this section is located in *examples/buffer-objects/* directory

---

Buffer objects are general purpose arrays of data residing in the graphics card's memory. After we have allocated its space and filled it with data, we can repeatedly use it in different stages of the rendering pipeline. This means reading from and writing to it. We can also move this data around. All of these operations won't require anything from the CPU.

There are different types of buffer objects for different purposes. The most commonly used buffer object is the vertex buffer object, which serves as a source of vertex arrays.

In this example, we intend to use one vertex buffer per object (i.e. one vertex buffer for the cube and one vertex buffer for the spotlight), in which the attributes are densely packed next to

each other in memory. We are not limited to using one single vertex buffer for all the attributes. Alternatively we could also use one vertex buffer for each vertex or a combination of both. Note that we can also mix the usage of vertex arrays and vertex buffers in one rendering.

Instead of using OpenGL API calls, we use the *QGLBuffer* class to manage the vertex buffers of the cube and the spotlight. The type of the buffer object can be set in the constructor. It defaults to being a vertex buffer.

We add a *QGLBuffer* member for each object, remove the vertex arrays we used in the previous version of the lighting example and add variables to hold the number of vertices, which will be necessary to tell OpenGL the number of vertices to render in the *GlWidget::updateGL()* method.

```
class GlWidget : public QGLWidget
{

    private:
    ...

    QGLBuffer cubeBuffer;
    ...

    int numSpotlightVertices;
    QGLBuffer spotlightBuffer;
    ...
};
```

Buffer objects are OpenGL objects just like the shader programs and textures which we have already used in the preceding examples. So the syntax for handling them is quite similar.

In the *GlWidget::initializeGL()* method, we first need to create the buffer object. This is done by calling *QGLBuffer::create()*. It will request a free buffer object id (similar to a variable name) from the graphics card.

Then, as with textures, we need to bind it to the rendering context to make it active using *QGLBuffer::bind()*.

After this we call *QGLBuffer::allocate()* to allocate the amount of memory we will need to store our vertices, normals, and texture coordinates. This function expects the number of bytes to reserve as a parameter. Using this method, we could also directly specify a pointer to the data which we want to be copied, but since we want to arrange several datasets one after the other, we do the copying in the next few lines. Allocating memory also makes us responsible for freeing this space when it's not needed anymore by using *QGLBuffer::destroy()*. Qt will do this for us if the *QGLBuffer* object is destroyed.

Uploading data to the graphics card is done by using *QGLBuffer::write()*. It reads an offset (in bytes) from the beginning of the buffer object, a pointer to the data in the system memory, which is to be read from, and the number of bytes to copy. First we copy the cubes vertices. Then we append its surface normals and the texture coordinates. Note that because OpenGL uses *GLfloats* for its computations, we need to consider the size of the c {GLfloat} type when specifying memory offsets and sizes. Then we unbind the buffer object using *QGLBuffer::release()*.

We do the same for the spotlight object.

```
void GlWidget::initializeGL()
{
    ...

    numCubeVertices = 36;

    cubeBuffer.create();
    cubeBuffer.bind();
    cubeBuffer.allocate(numCubeVertices * (3 + 3 + 2) * sizeof(GLfloat));

    int offset = 0;
    cubeBuffer.write(offset, cubeVertices.constData(), numCubeVertices * 3 * sizeof(GLfl
    offset += numCubeVertices * 3 * sizeof(GLfloat);
    cubeBuffer.write(offset, cubeNormals.constData(), numCubeVertices * 3 * sizeof(GLflo
    offset += numCubeVertices * 3 * sizeof(GLfloat);
    cubeBuffer.write(offset, cubeTextureCoordinates.constData(), numCubeVertices * 2 * s

    cubeBuffer.release();
    ...

    numSpotlightVertices = 18;

    spotlightBuffer.create();
    spotlightBuffer.bind();
    spotlightBuffer.allocate(numSpotlightVertices * (3 + 3) * sizeof(GLfloat));

    offset = 0;
    cubeBuffer.write(offset, spotlightVertices.constData(), numSpotlightVertices * 3 * s
    offset += numSpotlightVertices * 3 * sizeof(GLfloat);
    cubeBuffer.write(offset, spotlightColors.constData(), numSpotlightVertices * 3 * siz

    spotlightBuffer.release();
}
```

Just in case you're interested, this is how the creation of buffer objects would work if we did not use Qt's *QGLBuffer* class for this purpose: We would call *void glGenBuffers(GLsizei n, GLuint *buffers)* to request *n* numbers of buffer objects with their ids stored in *buffers*. Next we would bind the buffer using *void glBindBuffer(enum target, uint bufferName)*, where we would also specify the buffer's type. Then we would use *void glBufferData(enum target, sizeiptr size, const void *data, enum usage)* to upload the data. The enum called *usage* specifies the way the buffer is used by the main program running on the CPU (e.g. write-only, read-only, copy-only) as well as the frequency of the buffer's usage, in order to support optimizations. *void glDeleteBuffers(GLsizei n, const GLuint *buffers)* is the OpenGL API function to delete buffers and free their memory.

To have OpenGL use our vertex buffer objects as the source of its vertex attributes, we need to set them differently in the *GlWidget::updateGL()* method.

Instead of calling *QGLShaderProgram::setAttributeArray()*, we need to call *QGLShaderProgram::setAttributeBuffer()* with the *QGLBuffer* instance bound to the rendering context. The parameters of *QGLShaderProgram::setAttributeBuffer()* are the same as those of *QGLShaderProgram::setAttributeArray()*. We only need to adapt the offset parameter to uniquely identify the location of the data because we now use one big chunk of memory for every attribute instead of one array for each of them.

```
void GlWidget::paintGL()
{

    ...

    cubeBuffer.bind();
    int offset = 0;
    lightingShaderProgram.setAttributeBuffer("vertex", GL_FLOAT, offset, 3, 0);
    lightingShaderProgram.enableAttributeArray("vertex");
    offset += numCubeVertices * 3 * sizeof(GLfloat);
    lightingShaderProgram.setAttributeBuffer("normal", GL_FLOAT, offset, 3, 0);
    lightingShaderProgram.enableAttributeArray("normal");
    offset += numCubeVertices * 3 * sizeof(GLfloat);
    lightingShaderProgram.setAttributeBuffer("textureCoordinate", GL_FLOAT, offset, 2, 0
    lightingShaderProgram.enableAttributeArray("textureCoordinate");
    cubeBuffer.release();
    ...

    glDrawArrays(GL_TRIANGLES, 0, numCubeVertices);

    spotlightBuffer.bind();
    offset = 0;
    coloringShaderProgram.setAttributeBuffer("vertex", GL_FLOAT, offset, 3, 0);
    coloringShaderProgram.enableAttributeArray("vertex");
    offset += numSpotlightVertices * 3 * sizeof(GLfloat);
    coloringShaderProgram.setAttributeBuffer("color", GL_FLOAT, offset, 3, 0);
    coloringShaderProgram.enableAttributeArray("color");
    spotlightBuffer.release();

    glDrawArrays(GL_TRIANGLES, 0, numSpotlightVertices);
    ...

}
```

Rendering the scene now involves less CPU usage and the attribute data is not repeatedly transferred from system memory to the graphics card anymore. Although this might not be visible in this small example, it certainly boosts up the speed of programs where more geometry data is involved.

CHAPTER

THREE

CONCLUSION & FURTHER READING

We hope you liked this tutorial and that we have made you even more curious about OpenGL and 3D programming. If you want to delve deeper into OpenGL, you should definitively think about getting a good book dedicated to this topic. The OpenGL homepage[1] lists quite a few recommendations. If you are looking for an even higher level approach, you may consider taking a look at Qt/3D[2] and/or QtQuick3D[3].

Since OpenGL is able to compute a lot of information really fast, you may have thoughts about using it for more than just computer graphics. A framework based on this approach is called *OpenCL* (which is also managed by the Khronos Group Inc.). There even is a Qt Module for this framework. It is called QtOpenCL[4].

Links:

- *http://www.opengl.org* - The OpenGL homepage
- *http://www.khronos.org/opengl* - The Khronos Group Inc. homepage regarding OpenGL
- *http://www.khronos.org/opengles* - The Khronos Group Inc. homepage regarding OpenGL ES
- *http://doc.qt.nokia.com/qt3d?snapshot* - Qt/3D Reference Documentation
- *http://doc.qt.nokia.com/qt?quick3d?snapshot* - QtQuick3D Reference Documentation

[1] http://www.opengl.org
[2] http://doc.qt.nokia.com/qt3d-snapshot
[3] http://doc.qt.nokia.com/qt-quick3d-snapshot
[4] http://doc.qt.nokia.com/opencl-snapshot