Stefan Johansson, stefanj@cs.umu.se

Workshop 1

Introduction to graphics programming and transformations

Introduction

The purpose of this workshop is to introduce graphics programming with OpenGL and to understand the impact of the Model matrix. In the *Resource* folder on Cambro under W*orkshop 1* you find a zip-file with the C++ source code to be used together with Qt. The source code must be compiled with a compiler using the C++11 standard.

Decompress the files and see that you can compile them. The files are configured for the Linux-environment in the CS-department's computer labs.

Files

`vshader.glsl`	The vertex shader. This is were the vertices gets their position in NDC-coordinates. This file will be edited.
`fshader.glsl`	The fragment shader. We will not look much into this one during this workshop.
`openglwindow.cpp`	Contains the main class OpenGLWindow which initialize the window and the OpenGL environment. This class also handles all window events. It should not contain any geometry specific calls.
`openglwindow.h`	Header file for *openglwindow.cpp*.
`geometryrender.cpp`	Contains the GeometryRender class which is a subclass to OpenGLWindow. It initialize and render the geometry (the 3D model).
`geometryrender.h`	Header file for *geometryrender.cpp*.
`main.cpp`	Contains the main-function. You will probably not change anything in this file during the course.
`3dstudio.h`	A generic header file for all classes.
`workshop1.pro`	Qt make file used with *qmake* and *Qt Creator*. Use this to create a Makefile. Open this with Qt Creator.

Look through all files and see if you can understand what is happening in them. Not in detail, but the general outline of the files. In particular, identify the lines with the following function calls (see the comments in the Qt files). Ask if something is unclear.

Note: In Qt, there often exists two choices for the OpenGL commands. Either it is possible to call the native OpenGL functions (which have the prefix `gl`) or the equivalent functions in Qt. Sometimes it can be preferable to use the Qt version and sometimes the OpenGL version (and it is possible to mix them as done in the given code). Both variants are included below.

`program->bind()`
or
`glUseProgram`
	Makes the shader program active and binds it to the active context. The call `program->release()` or `glUseProgram(0)` releases the active shader program.

`vao.create()`
`vao.bind()`
or
`glGenVertexArrays`
`glBindVertexArray`
	These functions create and bind one or several Vertex Array Objects (VAO). There can be several vertex array objects associated to the same context.
	The VAO stores all information about the vertex data and the buffer objects (see below).

`vbuffer.create()`
`vbuffer.bind()`
`vbuffer.allocate`
`vbuffer.write`
or
`glGenBuffers`
`glBindBuffer`
`glBufferData`
	These four functions define the process of getting data to the graphics card and draw it. First we create a buffer on the graphics card with `create`/ `glGenBuffers` and get a buffer object. Using that object we bind that buffer to the `GL_ARRAY_BUFFER` identifier (normally used, but there are others). We can now send our vertex data to the buffer on the graphics card using `write`/`glBufferData`.

`glDrawArrays`	We draw our data using `glDrawArrays`.

`program->attributeLocation`
`program->uniformLocation`
or
`glGetAttribLocation`
`glGetUniformLocation`
`glVertexAttribPointer`
	Compare the arguments to `attributeLocation`/`glGetAttribLocation` with the contents of `vshader.glsl`.
	In `glVertexAttribPointer` notice the arguments '`2, GL_FLOAT`' and the buffer offset position. Why is that? Compare that with the definition of the variable points and what we copy to the graphics card using `write`/`glBufferData`.

`glClearColor` `glClear`	`glClearColor` defines and uses the specified background color and `glClear` clears buffers on the graphics card.

Qt specific calls:

`app.exec()` in main function	Enters the main event loop for the Qt window application.
`context = new QOpenGLContext(this)`	Create a new context for the window.
`initializeOpenGLFunctions()`	Initialize the OpenGL functions for the current context.
`OpenGLWindow::event()`	Callback function for Qt window events.

Normalized Device Coordinates

When the vertex shader is done, all vertices are expressed in *Normalized Device Coordinates*, NDC. Everything inside a specific volume in this coordinate system will then be projected to a 2D viewport in our window. In the next couple of exercises we will investigate NDC and the viewport. Projections will be covered in following lectures.

Exercise 1

Look at the coordinates of the 2D-triangle and how it appears on screen.

> Where is the 2D-coordinate (0, 0) located in NDC? What 2D-coordinate has the lower left corner of the window?

Exercise 2

Open vshader.glsl. The vertex shader is called once per vertex. We can notice that the shader is taking a 2D coordinate as input argument. In the main function, the vertex is given its final position by assigning a 4D (homogeneous) coordinate to the variable `gl_Position`.

> Change the z-value in vshader.glsl between -2.0 and 2.0. For which values do we see a figure on the screen? What do you think happens with the triangle when it is not visible?

Exercise 3

To summarize.

> Which NDC-coordinates are by default projected to the window? What happens with the vertices and lines outside of this cube?

Event callback functions

All events sent to a Qt window are handled by a specific event function or the general event function `event`. The latter function can be overridden (as done in the OpenGLWindow class) to handle any event sent to the window. Read more about how Qt's event system works at

http://doc.qt.io/qt-5/eventsandfilters.html.

Window Coordinates and the Viewport

Try resizing your window. In general, the coordinates are mapped to a viewport that the software decides the size and position of. Qt does not resize the viewport when the size of the window is changed.

Read about the glViewport-command. Since Qt sets the viewport when the window is initialized or resized we have two choices if we want to override this. Either we define the viewport when we redraw the window (currently in the `display` function) or we define a new callback function to handle window resize events. The latter is preferred.

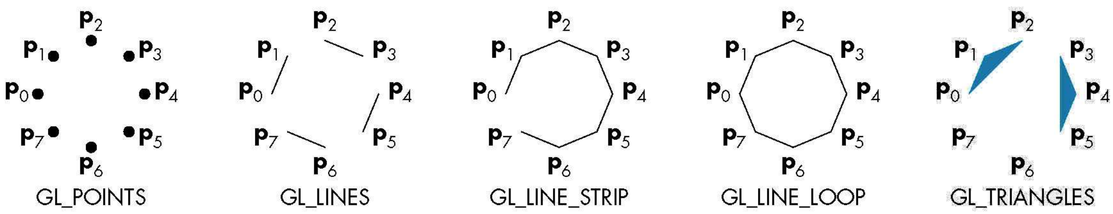
Exercise 4

Define a function `reshape` and use `glViewport` to define a viewport aligned in the lower left corner of the window and equal the window's width and height. There are two possibilities to call reshape, either by overriding the function `QWindow::resizeEvent` or checking for the event `QEvent::Resize` in the `event` function.

What happens now when the window is resized?

Draw properties

Again identify `glDrawArrays`. We only have a triangle but try to change the first argument to `GL_LINE_STRIP`, `GL_LINES`, `GL_POINTS`, `GL_TRIANGLES` and see what happens.



If we would like to draw the same triangle using `GL_LINES` instead, how would that have to effect the contents of `vertices`?

Model matrix

Now, lets play Linear Algebra and have some fun!

So far we have followed the vertices from the model through the vertex shader to NDC and how they are mapped to the viewport (window coordinates) and finally to screen coordinates (their location on the screen). We will now start to fiddle with our coordinates in the vertex shader.

The transformation matrix that takes our object (or model) from its local model coordinates to world coordinates is called the *model matrix*. In the following, we will see how it is integrated into the graphical pipeline.

Exercise 5

To transform the coordinates in the vertex shader we need a 4×4 transformation matrix (model matrix). Lets add the following lines to our program (as a private data member in the GeometryRender class):

```
QMatrix4x4 matModel;
```

Notice that the matrix is an identity matrix.

So, we have a matrix. To send it to the vertex shader we need to do two things. In a similar fashion as with `vPosition` we need to identify the parameter in the shader, and then send it. First we add this parameter to `vshader.glsl`

```
uniform mat4 M;
```

By declaring a variable to be *uniform* tells the shader that the variable is passed from the calling OpenGL application, and is global and read-only. The value of a uniform variable can also not be changed during execution of a draw call. Other common GLSL qualifiers are:

`const`	The declaration of a compile-time constant.
`in, out`	For function parameters passed into and back out of, respectively, a function.
`smooth`	Perspective corrected interpolated parameter.
`flat`	Non-interpolated parameter.

Now we want this matrix to be multiplied with our 4D-vertex in order to transform it. Change the content of the shader to, (this is equivalent to $v_{NDC} = M \cdot v_{model}$):

```
gl_Position =  M*vec4(vPosition, 0.0, 1.0);
```

Now to something tricky, GLSL is column-major ordered and C/C++ is row-major ordered http://en.wikipedia.org/wiki/Row-major_order. This means that if we represent a matrix as an array in C++ and copy that to the graphics card it will be transposed. Luckily this can be handled by OpenGL when transferring the matrix to the buffer. To get it right we must use the OpenGL call

```
glUniformMatrix4fv(location, count, GL_TRUE, v);
```

where the parameter `GL_TRUE` tells OpenGL to transpose the matrix. The corresponding call in Qt is

```
program->setUniformValue(locModel, matModel);
```

which do the transposing for us. We are now ready to send the model matrix to the graphics card. Identify where the following lines of code should be added. It depends on if we think the matrix can change during execution of our program or not. However, we should not put it in `display` since that function should be kept at a minimum. Note that all lines not necessarily should be inserted at the same place.

```
GLuint locModel;
locModel = program->uniformLocation("M");
program->setUniformValue(locModel, matModel);
```

If you run the program, nothing different should happen. The `matModel` matrix is the identity matrix.

Use the model matrix `matModel` so it scales all x-coordinates in the model by 2.0. Next, move all coordinates in positive y-direction by 0.5.