Stefan Johansson
stefanj@cs.umu.se

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE

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 *Files* folder on Canvas under W*orkshops* you find the zip-file 'Workshop1.zip' with the C++ source code for the workshop, including a Makefile. 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 where the vertices get 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 sub-class 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 much in this file during the course.
3dstudio.h	A generic header file for all classes.
glfwcallbackmanager.h	A class that works as a bridge between the C callback functions in GLFW and our C++ objects.
Makefile	Makefile for Linux and MS Windows.

Stefan Johansson
stefanj@cs.umu.se

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 files). Ask if something is unclear.

`glUseProgram`	Makes the shader program active and binds it to the active context. The call `glUseProgram(0)` releases the active shader program.
`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).

`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 `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 `glBufferData`.

`glDrawElements`	We draw our data using `glDrawElements`.

`glGetAttribLocation`
`glGetUniformLocation`
`glVertexAttribPointer`
	Compare the arguments to `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 `glBufferData`.

Stefan Johansson
stefanj@cs.umu.se

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE

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

GLFW specific calls:

`glfwWindowShouldClose`	Loop as long as the window is not closed.
`glfwMakeContextCurrent`	Make the context of the window the current OpenGL context.
`glfwSetErrorCallback`	Callback function for GLFW errors.
`glfwWaitEvents`	Sleep and wait for a system event to occur. For continuous rendering, use `glfwPollEvents` instead.

Normalized Device Coordinates

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

Note: You do not need to recompile the program when you only do changes in the shader files (however, you need to restart the program). Why?

Stefan Johansson
stefanj@cs.umu.se

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
GLFW has a large set of callback functions, where we below list only some of them. In the next exercise, you will use the callback function `resizeCallback` in the class `OpenGLWindow`. The callback function is set by `glfwSetFramebufferSizeCallback` in the header file glfwcallbackmanager.h. Another callback function defined is `errorCallback`.

More info about the callbacks can be found at

http://www.glfw.org/docs/latest/window.html#window_properties and

http://www.glfw.org/docs/latest/input.html

`glfwSetWindowRefreshCallback`	Set a callback when the window needs to be refreshed.
`glfwSetWindowSizeCallback` or `glfwSetFramebufferSizeCallback`	Set a callback when current window is resized.
`glfwSetKeyCallback`	Set a callback when any key is typed.
`glfwSetCharCallback`	Set a callback when any Unicode character is typed.
`glfwSetCursorPosCallback`	Set a callback for mouse motions.
`glfwSetMouseButtonCallback`	Set a callback for mouse clicks.

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. GLFW does not resize the viewport when the size of the window is changed.

Read about the OpenGL function `glViewport`. Since GLFW set 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
In the class `OpenGLWindow,` create a new 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. Call reshape from the (currently empty) callback function `resizeCallback`. But

Stefan Johansson
stefanj@cs.umu.se

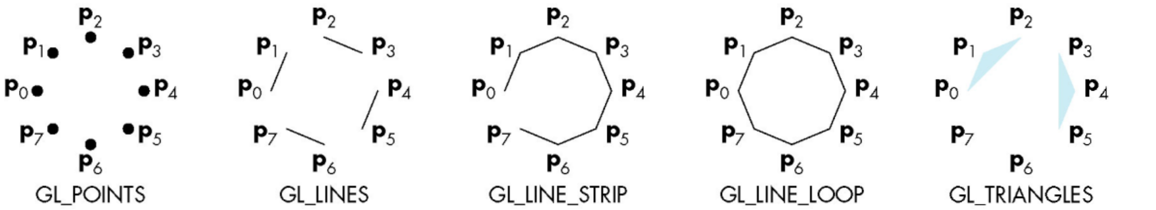
before we call `glViewport` we have to check that we have a valid context. That is done by adding the following lines in the beginning of `reshape`:

```
if (glfwGetCurrentContext() == nullptr)
    return;
```

What happens now when the window is resized?

Draw properties

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



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

Model matrix

Now, let's 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 graphic pipeline.

Exercise 5

To transform the coordinates in the vertex shader we need a 4×4 transformation matrix (model matrix). Let's add the following lines to our program:

Add in the top of geometryrender.h before the class definition

```
typedef float Mat4x4[16];
```

and in the class definition under private

```
Mat4x4 matModel = {1.0, 0.0, 0.0, 0.0,
                   0.0, 1.0, 0.0, 0.0,
```

Stefan Johansson
stefanj@cs.umu.se


```
0.0, 0.0, 1.0, 0.0,
0.0, 0.0, 0.0, 1.0};
```

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.

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 are not necessarily inserted at the same place.

```
GLuint locModel;
locModel = glGetUniformLocation(program,"M");
glUniformMatrix4fv(locModel, 1, GL_TRUE, matModel);
```

Stefan Johansson
stefanj@cs.umu.se

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

> Change 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.

Continue to experiment with the model matrix and the code!

Exercise 6 (if time permits)

In this exercise you will add another triangle (face) to the object and also extend the representation of the vertices with the z-coordinate. We start with adding another triangle.

Go to the method `loadGeometry()` in the `GeometryRender` object. To add another triangle, we need to add *one* vertex to the vector `vertices` and *three* indices to the vector `indices`. The three new indices will be the corresponding index of the vertices that defines the new tringle (two corresponding to already existing vertices).

> Add a new vertex (position of your choice) to the `vertices` vector. Also add the three new indices to the `indices` vector (one should correspond to the new vertex). Did you get a second triangle?

In the project, you will load 3D objects that also include the *z*-coordinate. However, the vertices in the code are now specified only using the *x*- and *y*-coordinates.

> Modify the code such that the vertices also include the *z*-coordinate, i.e. (*x*,*y*,*z*).

Note: You have to change the code in several files and also in the vertex shader `vshader.glsl`. Take extra notice of the vector sizes when writing the vertices to the vertex buffer, and the parameters of `glVertexAttribPointer(…)`.