Computer Graphics - Lab 1
Getting started with OpenGL

Nov 07, 2016

Introduction

The goal with this lab is to learn some basics about OpenGL, to work with the GLEW and GLFW libraries and to apply linear algebra in 3D graphics calculations. GLFW provides the tools to open a window and receive mouse and keyboard inputs. GLEW is used to load the OpenGL API.

The main subject of the lab is to draw triangles in different ways by using the OpenGL functions. This means sending data from a C++ program to the GPU by using the OpenGL buffer objects and interpret the data in shader programs, written in basic GLSL (OpenGL Shading Language) for vertex and fragment shaders.

The machine that is used for this lab is a laptop with the operating system Ubuntu 16.04.

1.1 Hello GPU

The first task is to get some information about the graphics hardware and the version of OpenGL that is used.

Renderer and OpenGL version

Using the OpenGL function *glGetString*, it's possible to receive information about the graphics hardware in the machine.

Renderer	GeForce GT 740M / PCIe / SSE2
OpenGL version	4.5.0 NVIDIA 361.42

Hardware specifications

Video RAM	2048 MB
CPU Count	1
Clock Frequency	810 MHz
Memory Bandwidth	28.83 GB / s
Pixel Rate	6.48 GPixel / s
Texture Rate	25.92 GTexel / s
Shading Units	384

The supported OpenGL version 4.5 is the latest release, so no features are missing.

1.2 Your first triangle

To draw a triangle, some OpenGL functions were used. The data that is needed to draw a triangle is an array of vertex positions. These vertices are points in a space, expressed as vectors. The x and y -components of the vertices are in the range [-1, 1]. This is to describe where on the screen they appear. The size and position of the triangle is defined by the position of these vertices (corners).

Creating the Vertex Objects

To draw the triangle on the window (640 x 480), there are a few steps that needs to be completed first. To tell the GPU that these vertices represent a triangle a Vertex Buffer Object (VBO) is created, using the function *glGenBuffers* and the macro **GL_ARRAY_BUFFER**. After that the function *glBindBuffer* is called to tell that the VBO is the one currently used so that it can be initialized and filled by the triangle vertices data by using the function *glBufferData*.

After that a Vertex Array Object (VAO) is created using the function *glGenVertexArrays*. Calling the function *glBindVertexArray* binds the VAO so that it can be used to store connections between VBOs and the vertex attributes. When the VAO is bound as the current object it's time to indicate that the currently bound VBO should be used with the vertex attributes with index 0, using the function *glEnableVertexAttribArray*. The next function *glVertexAttribPointer* tells the vertex attribute how it should interpret the data in the VBO. So now the vertices have been stored into the buffer which the GPU will read from.

Next step would be to program the shaders, to process the vertices which are compiled during runtime, but since they've already been provided, it will be covered more in the next task. Instead, it's time to tell the CPU to draw the data that has been provided. However, it's important to bind the VAO before drawing to prevent complications and to tell which data is to be drawn. The remaining function to be called now is the draw function by using the *glDrawArray*, placed in a continuous loop. If everything went well a triangle should be rendered on the window.

Dimensions and z-coordinate limits

The vertices that was used:

```
static const GLfloat triangle_vertices[] = {
    -1.0f, -1.0f, 0.0f,
     1.0f, -1.0f, 0.0f,
     0.0f,  1.0f, 0.0f,
};
```

Since the created window has the dimensions 640 x 480 and the origin point (0, 0) for the coordinates start in the middle of the screen (default for OpenGL), the vertices would equate to the following dimension positions:

- First vertex would be in the lowest left-corner, which means $(x,y) = (0,\ 640)$
- Second vertex would be in the lowest right-corner, which means $(x,y) = (480,\ 640)$
- Third vertex would be in the middle top, which means $(x,y) = (480/2,\ 0)$

The z-coordinates of the vertices were always 0, so that it would stay flat on the screen. However changing these values would give an interesting result. Since this is a flat triangle with only 3 vertices, if changing the z-value for example the first vertex to above 1.0f, one would observe that a part of the lowest left-corner has disappeared a bit. Actually however it has not and is just not being drawn in the visible space (the screen). The same goes for the opposite direction, negative z-coordinate goes through the screen towards the reader. When the limits are exceeded it will disappear from the screen. The z-coordinates of the vertices “pulls” the triangle towards the camera or the away depending on the value. When the limits are beyond what can be displayed, it will disappear from the visible space.

1.3 Introduction to shaders

The goal of this task is to draw a multi colored triangle with help of a vertex shader and a fragment shader. A shader is a GPU program and a stage in the rendering pipeline that is used to draw primitives. The vertex shader draws the primitives with vertices as input to set the size and position. The fragment shader sets the colour of the primitive. The OpenGL shaders are written in the programming language GLSL, which is a simple C-like language.

Shader Program

To be able to make the project cleaner and easier to handle, a wrapper class for the shader programs was implemented with appropriate functions for updating the input variables in the shaders.

To create a shader program first the function *glCreateProgram* is called to create an empty program object. Thereafter a file is read with the shader program code (GLSL), for either Vertex shader or Fragment shader. This is then made into the appropriate shader program through the function *glCreateShader* and linked to the empty program object through attaching and linking the file code using the functions *glAttachShader* and *glLinkProgram*. When all the linking is done, the program is installed to be used in the rendering by calling the function *glUseProgram*.

Passing variables

An “out vector4” variable was added to the vertex shader and an “in vector4” to the fragment shader to be able to pass the vertices to the fragment shader for coloring the triangle. 0.5 was added to each vertex before passing the vector to the fragment shader to get the colours as close as possible to red, green and blue.

The reason for the different colour value of the vertices in the triangle, is that the colour that is set, depends on the position of the vertex. More clearly, output of color from the fragment shader depends on what vector the vertex shader outputs.

Rasterization and interpolation

Rasterization causes different values for the fragment shader from the vertex shader. Interpolation qualifiers controls this behaviour by specifying a way the output values are interpolated across a primitive.

There are three types of interpolation: **flat**, **noperspective** and **smooth**. Flat means no interpolation will take place the value from vertex shader will be passed on to the fragment shader. Noperspective will perform a linear interpolation. Smooth will perform a perspective correct interpolation on the values from the vertex shader and is also the default in OpenGL.

1.4 Passing parameters to shader programs

Making the triangle respond to key inputs to that it expresses some kind of motion, such as changing position or rotating, could be achieved by sending data to the vertex shader, which controls the position of the triangle. This will be implemented by sending a uniform variable to the shaders, with the new updated values, instead of changing the coordinates and recalculating.

The way this task was done was by utilizing the Shader class from the previous task and making a method (*SetVertexPosition*) for setting a uniform variable in the vertex shader with a different variable which was modified by the inputs of the keys. To get the uniform variable from the vertex shader to be modified, the function *glGetUniformLocation* was used with the shader program. The uniform vector2 variable in this case was named "*pos_offset*". Upon changing the uniform variable it needed to be sent back to the vertex shader, by using the function *glUniform2fv*, which sends a vector2 to the vertex shader, to the specified uniform variable.

The same procedure was done for the color changing by the scroll wheel. A uniform float variable named "*modifier*" was created in the fragment shader. The method in the Shader class is named (*SetFragmentModifier*) and does the same procedure as described before. This time it gets the uniform variable from the fragment shader and sends it back with the new value applied from the scroll wheel.

The result is a triangle which can move vertically and horizontally with the arrow keys and change the color gradiently by scrolling the mouse wheel.

1.5 3D geometry

Instead of a triangle, in this task a 3D icosahedron was to be displayed. An icosahedron is a geometric figure with 20 flat sides. An array of 12 3D vectors (of type float) was given to describe the vertices. Same vertex could be used for more than one triangle in the icosahedron, so only totally 12 vectors were needed. To be able to know which 3 vertices were to be used for which triangle, an index array of 60 elements (of type short) was also given.

The reason for doing it this way instead of just use an array of 20 3D vectors (of type float), is because of better performance and lower memory cost.

60 * 4-bytes **float**	240 bytes
12 * 4-bytes **float** + 60 * 2-bytes **short**	168 bytes

To be able to draw this, besides creating the Vertex Array Object and Vertex Buffer Object, an Element Buffer Object (EBO) had to be created to be able to handle the indices together with the float array. This buffer will specify what the indices should draw (called indexed drawing). Creating the EBO was done by using the same functions as described previously for VBO: *glGenBuffers* to create the EBO with the **GL_ELEMENT_ARRAY**, *glBindBuffer* to bind it as current buffer and *glBufferData* to load the indices data into the buffer.

To color the icosahedron a function *hsv2rgb* (hue saturation value to red green blue) was provided in the fragment shader. It was used to set the color of the icosahedron with respect to the z-value of the vertices. The z-value is in this case always negative, because z starts at 0 of a visible vertex and decreases the further ahead it is of the observer. Because of this, the z-value had to be negated to get a positive value to represent the depth.

Finally to draw the object a different draw function was used instead of the earlier used one *glDrawArrays*. Since EBO is used this time, it's indexed drawing. This requires the function *glDrawRangeElements* to draw the triangles for the sides.

1.6 Model, view and projection transformations

In this task different types of matrices will be created to be able accommodate the need for a camera and its projective properties and the position of the model in the world. The model and view transformations are rigid-bodies with the differences that the view transformations are inverted and the model transformations are applied directly to the geometry. The projection matrix is scaling objects in the world to look bigger or smaller. The matrices that were created had to be calculated in a certain way and some multiplication was going to be done so a function called *MUL_4x4* was created to multiply two 4x4 matrices together. Starting with the model matrix, since it was going to be able to rotate, the x and y values for the rotation matrices was going to be determined by the keyboard inputs. So the model matrix was made up of two rotation matrices, x and y, which was multiplied after each other $(M = X * Y)$.

The view matrix was going to represent the camera, so it had to be transformed 2 units along the z-axis, and then inverted to be able to multiply it with the model matrix to create a model-view matrix. The projection matrix was a simple 4x4 matrix with the near (1.0f) and far (100.0f) values. By multiplying this projection matrix with the model-view matrix, the finished 4x4 matrix was produced, model-view-projection matrix. This matrix is the one which will be applied to the uniform variable *mvp* in vertex shader, which will set the position values by multiplying the matrix with a vertex and the color of the icosahedron in the fragment shader, using the *hsv2rgb* function. Since a 4x4 matrix is sent in to a uniform variable, this time a different function is used to set it. The function is called *glUniformMatrix4fv.* The resulting matrix multiplication ends up like this:
$MVP = Projection * View * Model$

The order of the matrix multiplication is important since it is not commutative. It is not like the regular multiplication where $z = x * y \Leftrightarrow z = y * x$. The produced result is different and gives different results depending on from which side you multiply the matrices. $A = BC \Rightarrow A \neq CB$. If we were to multiply the matrices in the reverse order, such as this:

$MVP = Model * View * Projection$, it would mean that the model is expressed as the projection matrix and the projection as the model matrix, which doesn't make sense.

1.7 Linear algebra isn't fun

GLM (OpenGL Mathematics) is an open-source header for C++ which implements classes and functions based on the ones in the GLSL shaders for calculating linear algebra and constructing vectors and matrices. This makes writing the code easier and more understandable. The task here was again to define a model view projection matrix to use in the shaders, but this time with help from GLM.

To do that, the matrix calculation functions *glm::rotate*, *glm::translate*, *glm::inverse* and *glm::value_ptr* were used together with the GLSL-like classes *glm::mat4* and *glm::vec3*. See *lab1-7.cpp* in the project for coding details.

glDepthFunc

The *glDepthFunc* function is used for specifying the depth buffer comparison value. The parameter *func* specifies the depth comparison function and is initially set to **GL_LESS** which means that the pixel that currently is tested for depth is to be drawn if it's depth value is less than the current stored depth value.

GL_NEVER - Never passes.
GL_LESS - Passes if the incoming depth value is less than the stored depth value.
GL_EQUAL - Passes if the incoming depth value is equal to the stored depth value.
GL_LEQUAL - Passes if the incoming depth value is less than or equal to the stored depth value.
GL_GREATER - Passes if the incoming depth value is greater than the stored depth value.
GL_NOTEQUAL - Passes if the incoming depth value is not equal to the stored depth value.
GL_GEQUAL - Passes if the incoming depth value is greater than or equal to the stored depth value.
GL_ALWAYS - Always passes.

The **GL_LESS** parameter is appropriate in this case because the representation of the icosahedron would not make sense if for instance the closest triangle would be overdrawn by the triangle furthest away. That could happen if **GL_ALWAYS** is used and the triangle furthest away is last in the draw order.