OpenGL: Introduction

February 5, 2019

Computer Graphics

A basic OpenGL application

In this assignment, you will program a very basic OpenGL application that renders a colored triangle in a window. The resulting application will be the basis for subsequent OpenGL assignments.

Download `OpenGL_basics.zip` from Nestor (*Lab Assignments*), unzip/extract the files and open `OpenGL_basics.pro` in QtCreator. Make sure to keep a copy of your finished code around, since assignments build on top of each other.

Framework layout

The downloaded framework serves as a starting point for the exercise. The provided code already creates an OpenGL window and catches input, so you do not have to implement this yourself. You will need to edit or view the following files of the framework during the OpenGL exercises:

- `mainview.h` defines the class for the OpenGL widget we use. You can declare public/private members here in a similar way as in Java.
- `mainview.cpp` will hold all C++ code calling the OpenGL functions:
  - The constructor `MainView()` can be used to initialize variables. Note that you cannot use any OpenGL functions here yet, since OpenGL is not yet initialized at this point.
  - The destructor `~MainView()` is the last function called before the application closes and thus can be used to free pointers and OpenGL buffers you created.
  - The `initializeGL()` function is called when the OpenGL functions are enabled. You should make OpenGL calls here, or after this function has been called.
  - The `paintGL()` function is called when a repaint is executed. Place all your rendering calls inside this function.
- In `user_input.cpp` you will find all functions related to processing user input. Read the comments to understand when each event is generated.

- In QtCreator, expand the Resources folder (located at the bottom of the file tree on the left) until you see the shader files, `vertshader.glsl` and `fragshader.glsl`. These are the vertex shader and fragment shader, respectively.
- Besides shaders, other files can be added to the resources. In the future, textures and 3D models will need to be added too. To add your own files, right click the resources file and click add existing files to select the files you want to add after placing them in the correct folders.
- There is also a directory called "dontpanic", please read the checklist inside for solutions to common mistakes before consulting a TA. This document usually proves useful throughout all OpenGL exercises.
- Feel free to create your own classes and source files when expanding the code. The clearer the code is, the quicker we can help debug it.

Please read the comments and familiarize yourself with the code. If you have any questions, please ask the teaching assistants.

1 OpenGL basics

After downloading the framework, you should have a working OpenGL window. It is currently black, because nothing is rendered yet. To render anything, we need at least three OpenGL objects:

- A shader program, which determines where and how pixels will be rendered on the screen.
- A *vertex buffer object*, or *VBO* to store data in.
- A *vertex array object*, or *VAO* which will remember most of the current OpenGL state.

1.1 The shader program

The shader program consists of two parts; the vertex shader and the fragment shader. Both are small computer programs written in *GLSL*, a language which looks somewhat like C. The shader program operates on data that was sent to the GPU (through a VBO). The vertex shader computes where pixels should be drawn, and the fragment shader determines what color pixels will get.

Creating a shader program requires two text files. Because this is quite tedious, we will use a Qt wrapper to do this: the class `QOpenGLShaderProgram`. After instantiating this class, use the function `addShaderFromSourceFile` twice, once for the vertex shader and one fragment shader. The function takes two arguments:

- The kind of vertex you are adding, either `QOpenGLShader::Vertex` or `QOpenGLShader::Fragment`.
- The source file, `:/shaders/vertshader.glsl` for example. **Note** the `:` in front of the file name.

After adding both the vertex and fragment shader to it, call the function `link()` to compile the shader. After this has been done (and when no errors occured), the shader is ready to be used for rendering. Call the function `bind()` to make this shader active.

In short, shaders are our way to program the GPU. We can send data and commands to the GPU from our C++ application, but the way this data is interpreted and rendered is programmed in the shader programs.

1.2 The VBO

The vertex buffer object is a buffer on the GPU. This is memory usable by the graphics card. When shaders are rendering things, they take their information from such buffers. All points in a 3D model for example are sent to a buffer first, and then rendered from that buffer. But you may also store other forms of data such as normals or colors.

1.3 The VAO

The vertex array object is an object that remembers OpenGL state when bound. It can for example remember how data from a VBO needs to be interpreted. When rendering that VBO, the VAO is bound first.

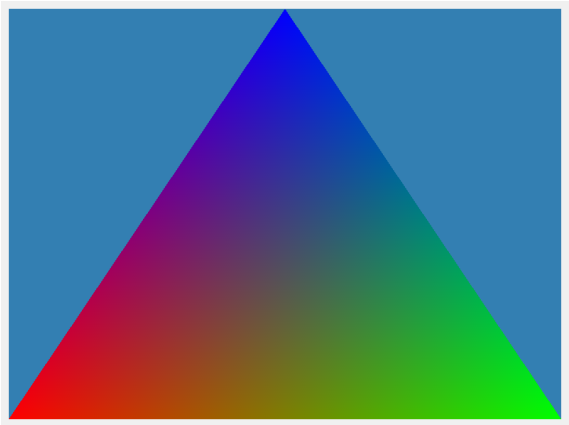

Figure 1: *A colored triangle*

2 Rendering a triangle

Figure 1 shows the goal of this exercise: a triangle with color gradients rendered on some background color. To render a triangle, we must first define a triangle.

2.1 Defining a triangle

A colored triangle consists of three points, and each point contains a color and a location. We must define these points and send them to the GPU before we can instruct the GPU to render them. Every point of this triangle is called a *vertex*. Define this vertex as a struct (or class, if you prefer). I suggest you make a new header file to define it. This struct should contain 2d coordinates and color in three components (red, green and blue). That's five floating point values in total. Make sure the struct *only* contains five floats

and nothing more, because we will send this data to the GPU.

Make an array of three vertices, and set the values of the vertices to form a triangle. Make sure the coordinates lie within *screen space coordinates*, so within the range $[-1, 1]$. The three colors can be any colors, but figure 1 uses red, green and blue. Make sure the rgb values lie in the range $[0, 1]$. The color green for example would be $(0, 1, 0)$.

2.2 Generate the OpenGL objects

The three OpenGL objects need to be created:

1. The shader object can be created from the two shaders in the resources folder. The shaders can have empty main functions for now.
2. The VBO can be created using `glGenBuffers`. Make sure the VBO is a `GLuint`.
3. The VAO can be created using `glGenVertexArrays`. Make sure the VAO is a `GLuint`.

Make sure to store these three objects' variables and pointers in the MainView class (or in your own class if you made one).

2.3 Sending the triangle to the GPU

It is time to send the array of vertices to the VBO. Because we want the VAO to remember the OpenGL state, it must be bound first. This can be done using the `glBindVertexArray` function. Then, the VBO needs to be bound using `glBindBuffer`. This function takes two arguments, the *target* and the buffer. For more advanced OpenGL functionality, multiple types of targets exists. For vertex data, you can use `GL_ARRAY_BUFFER` as target.

The buffer is now bound to `GL_ARRAY_BUFFER`, which means we can work with it. The vertices can be uploaded to the `GL_ARRAY_BUFFER` using the function `glBufferData`.

2.4 Telling the GPU how the data has been layed out

The vertex currently contains two vectors; one 2D vector for the position, and a 3D vector for the color. In OpenGL terms, there are two *attributes*. It is necessary to tell OpenGL (and the VAO we have currently bound) what parts of the data it should look at for what attribute.

1. Call the function `glEnableVertexAttribArray` with the arguments `0` and `1` to enable attributes zero and one.
2. Call the function `glVertexAttribPointer` on both attributes. Make sure to point one of the attributes to the 2d (2 floats) vertex, and the other to the 3d (3 floats) color. Make sure the *stride* parameter is equal to *sizeof(Vertex)*.

2.5 Writing the shader program

The shader program currently has two empty shader files, consisting only of a `void main()` function which does nothing. It is time to implement the shaders. Keep in mind GLSL is much like C, but not the same.

Implement the vertex shader first. On top of the file, after the version definition, the inputs for the shader program can be defined. Assuming your position is attribute 0 and the color is attribute 1, this can be done in the following way:

```
layout (location = 0) in vec2 position;
layout (location = 1) in vec3 color;
```

`vec2` and `vec3` are built in glsl variables, much like `int` or `float` in C++. The variables `position` and `color` can now be accessed throughout the vertex shader.

A color value must be passed to the fragment shader next. There are three vertices, so three color values will be received after another. While the vertex shader is called for each vertex, the fragment shader is called for each pixel that lies within the triangle that the vertices form (called a fragment). When the vertex shader outputs a value to the fragment shader, the output values are interpolated over the triangle it draws. Figure 1 shows the vertex colors are interpolated.

To output information from the vertex shader to the fragment shader, use an *out* variable. After the input values, declare `out vec3 interpolatedColor;`. When the fragment shader then defines `in vec3 interpolatedColor;`, it will receive the interpolated values the vertex shader wrote to its *out* value.

To finish the vertex shader, write the color value to the *out* variable, and write the position to the glsl builtin variable `gl_Position`. Make sure this is a `vec4`; the third and fourth values are the z coordinate (which can be zero in 2d), and the w value, which should be one. To make a `vec4` from a `vec2` and two floats, glsl allows the syntax `vec4(your2dVector, float1, float2)`.

The fragment shader should contain the `in vec3 interpolatedColor` described earlier; it will receive the interpolated color here. It should also define one `out vec4` value, which is the *rgba* color the pixel should be. The last value is transparency (or alpha), this can be one (opaque) for now.

2.6 Rendering the triangle

The final step is rendering the triangle. This should be done in the `MainView::paintGL` function. Before rendering, it is wise to clear the screen from any previously rendered content. This is done using the `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` function. If the background color should be anything else than black, use the `glClearColor` function to set another clear color.

Next, make sure the VAO is bound; the VAO should have been bound when the VBO and the vertex attributes were configured; this state was remembered by the VAO. If the VAO is bound before rendering, it will use that configuration. Bind the VAO using `glBindVertexArray`.

Also make sure the shader program is bound. This can be done by calling its member function `bind`.

Finally, use the OpenGL function `glDrawArrays` to draw the triangle. Its first argument (mode) can be `GL_TRIANGLES` because a triangle should be drawn; there are also other modes for different shapes.

3 Cleaning up

Always make sure to free the resources allocated by OpenGL. This should be done in the destructor function (`MainView::~MainView`). The OpenGL functions `glDeleteBuffers` and `glDeleteVertexArrays` can be used to free VBO's and VAO's respectively. The shaderprogram will be automatically deleted by the Qt wrapper class.