OpenGL Assignment 0: OpenGL Basics

January 20, 2022

Computer Graphics

Introduction

The goal of this assignment is to become familiar with the basics of the OpenGL API by writing an application that renders a colored triangle. The subsequent OpenGL assignments build upon the fundamentals explained here.

Download `OpenGL_0.zip` from Nestor (*Assignments*), extract the files and open `OpenGL_0.pro` in QtCreator.

This framework is to be used for this assignment only. For every future OpenGL assignment, a new framework will be provided. However, it is also possible — and recommended — to repeatedly extend the framework for OpenGL Assignment 1 to complete all subsequent assignments.

Framework layout

The downloaded framework serves as a starting point for the assignment. 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 assignments:

- `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 what causes an event to be sent to that particular event handler function.

- In QtCreator, expand the Resources folder (located at the bottom of the file tree on the left) until you see `vertshader.glsl` and `fragshader.glsl`. These are the source codes for 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 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.

- Feel free to create your 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

In this Section we'll describe the general OpenGL concepts necessary for this assignment. Section 2 contains the instructions to put the concepts into practice.

1.1 OpenGL as a state machine

Before we tell the OpenGL API to render something, we need to tell it *what* to render and *how* to render it. Here, it is useful to visualize OpenGL as a *state machine*, or alternatively, as one big Object-Oriented Programming class with variables that store its state, functions to manipulate its state, and functions that make the class do something depending on its state.
What will be rendered is captured by the concepts of the VBO and VAO, as explained below. *How* this data should be rendered is determined by the shader program, also explained below, as well as the state-changing OpenGL function calls that precede it. For an example of the latter, consider the following code snippet where `glPolygonMode` changes the state and `glDrawArrays` tells OpenGL to render something.

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
// This tells OpenGL to fill the insides of the polygons when rendering.
// This is also the default and can be seen in Figure 1.
// OpenGL's state has now been changed, but the change cannot yet be seen.

glDrawArrays(..);
// OpenGL renders filled polygons as specified by its state.
// We now see filled polygons.

// This tells OpenGL to only draw the outline (wireframe) of the polygons.
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
// OpenGL's state has now been changed, but we still see filled polygons.

glDrawArrays(..);
```

```
// OpenGL renders polygon outlines as specified by its state.
// We now see the outlines of the polygons.
```

1.2 A minimal OpenGL application

To render something, we need (at least) three OpenGL objects:

- A shader program, which determines where pixels will be rendered on the screen and in what color.
- A *vertex buffer object*, or *VBO* to store data in.
- A *vertex array object*, or *VAO* to store which buffers are associated with it and in what format(s) the data in these buffers are stored.

1.3 The shader program

For this assignment, the shader program will consist of two parts: the vertex shader and the fragment shader. Both are small computer programs written in the OpenGL Shading Language, *GLSL*, a language that looks somewhat like C. The shader program operates on data that was sent to the GPU (using a VBO). The vertex shader computes where pixels should be drawn, and the fragment shader determines what color the pixels will get.

Creating a shader program using plain C/C++ and OpenGL requires reading the source code files into strings, and compiling and linking these using OpenGL functions. The code to do this is quite verbose, therefore we use a Qt wrapper for this instead: the class `QOpenGLShaderProgram`. After the class is instantiated, its function `addShaderFromSourceFile` should be used twice, once to add and compile the vertex shader and once to add and compile the fragment shader. The function takes two arguments:

- The kind of shader 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, its function `link()` should be called to link the shader program. After this has been done, the shader program is ready to be used for rendering. If the compilation or linking fails, an error message will appear in QtCreator's Application Output window. Call its function `bind()` to make this shader program the "current shader program". This changes the OpenGL state such that this shader program will be used by the render call (`glDrawArrays`) later in this assignment.

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 shaders.

1.4 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. For example, all vertices in a 3D model are sent to a buffer first, and are then rendered from that buffer. Buffers may also contain other forms of data such as normals or colors.

1.5 The VAO

The vertex array object stores which buffers are associated with it and how the data in these buffers need to be interpreted. Let's say you would like to render a 3D model which consists of vertices and where each vertex has a color. One possible implementation is to create one VAO for this model, which has two corresponding VBO's: one for vertex data and one for color data. To render this model, the VAO is bound first. This informs OpenGL where the data is and how it is laid out. Then a render call (e.g. `glDrawArrays`) is invoked.

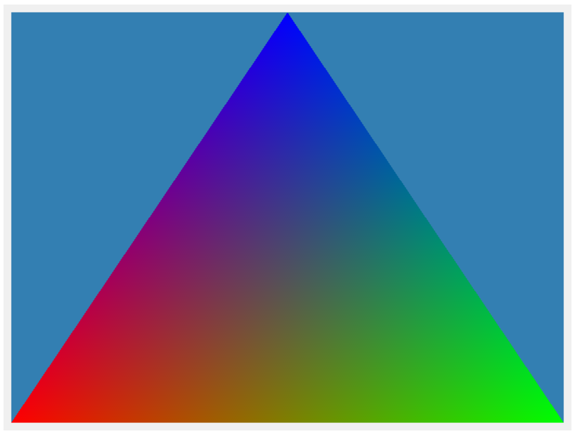

Figure 1: *A colored triangle*

2 Rendering a triangle

After downloading the framework and running the application, you should see a working OpenGL window. It is currently black, because nothing is being rendered yet. Figure 1 shows the goal of this assignment: 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 vertices, and each vertex contains a color and a location. We must define these vertices and send them to the GPU before we can instruct the GPU to render them.
Define a new object (as a struct) for a vertex, preferably in a new header file. This struct should contain a 2D coordinate (x, y) and color in three components (red, green and blue). That is 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.

At the end of the `initializeGL()` function, instantiate three vertices, one for each corner of 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 $(0, 1, 0)$.
Now create an array holding the three vertices. For this assignment, a plain C-array will do, but `std::vector` and `QVector` are also good options.

2.2 Generate the OpenGL objects

Now we will create the VBO and VAO using the OpenGL functions `glGenBuffers` and `glGenVertexArrays`, respectively. These functions take a pointer to a `GLuint` type (an unsigned int) in which they store the 'name' of the object. We can later refer to that object using this name. Refer to the OpenGL documentation to see how the mentioned OpenGL functions should be used exactly.

1. For the VBO, declare one GLuint in the `MainView` class. Then call the function `glGenBuffers` in the `initializeGL` to generate the VBO and store its name in the `GLuint`.
2. For the VAO, also declare one GLuint in the `MainView` class. Then call the function `glGenVertexArrays` in the `initializeGL` function to generate the VAO and store its name in the `GLuint`.
3. For the shader program, instantiate a `QOpenGLShaderProgram` in the `MainView` class. In the `initializeGL` function, add the two shader source files in the resources folder to it. The shader source files can have empty main functions for now. Refer to Section 1.3 for detailed instructions, if necessary.

The variables and objects related to the triangle are now stored in the `MainView` class. Another option would have been to create a Triangle class to store the VAO, VBO and vertices in.

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 store how the data is laid out, as defined by the following function calls, 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 exist. For vertex data, you can use `GL_ARRAY_BUFFER` as a target.

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

2.4 Telling the GPU how the data has been laid 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) position, and the other to the 3D (3 floats) color. The *stride* specifies the number of bytes between different vertices in your vertex array. Since the vertices should be tightly packed, make sure that *stride* is set to `sizeof(Vertex)`.

The last parameter called *pointer* can be a bit confusing. It specifies the offset, in bytes, of the attribute from the start of the vertex structure. For example, for `struct Vertex{ float x, y, r, g, b; }`, the member x is at offset 0 from the start of the structure since it is the first member. The member y is at offset 4, since x is first and takes up 4 bytes. r is at offset 8 since it comes after x and y, etc.

You can make use of the `offsetof` macro (from `<stddef.h>`) to find the offset of any member, and then cast the result to a `void*` like so:

```
glVertexAttribPointer(..., (void *)offsetof(Vertex, x))
```

2.5 Writing the shaders

The shader program currently has two empty shader source files, consisting only of a version definition and 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. At the 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 types, 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 that 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 built-in variable `gl_Position`. Make sure this is a `vec4`; the third and fourth values are the z coordinate (which can be 0 in 2D), and the w value, which should be 1. 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` variable, with a name of your choice, which is the *rgba* color the pixel should be. The last value is transparency (or alpha), this can be 1 (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, at the end of the `paintGL` function, 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 other primitives.

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 shader program will be automatically deleted by the Qt wrapper class.