Visual Computing

Lab Exercises

Samuel Silva

November 1, 2021

Contents

Vertex Colors, 2D Viewing, and Transformations

2

For today's class, the goal is to implement a simple "Catch the Square" game. While doing it, you should be able to learn about:

- Using color atributes for vertices
- Modifying shaders to consider color data
- Add support for considering transformation matrices in the vertex shader
- Write data to the uniform buffers
- Using the OpenGL Mathematics package (`glm`) to compute transformation matrices
- Understand the purpose of the projection, view, and model matrices
- Have more than one object in one scene and treat them independently
- Learn a bit more about key events and keeping a steady framerate in `pygame`

2.1 Prequel

Open the file `CV_HandsOn_02_Ex1.py` and explore its contents. It is quite similar to the examples in the previous class. Take some time to identify the different relevant parts: the event loop, the initialization of the shaders, and where the instructions for rendering the scene are. Run the script and you should see a big red triangle.

2.2 Color of Duty

You can start by changing the color of the triangle, as you did in our previous class to a value of your choice. Where?

The particularity of placing a static color value inside the shader is that it is applied to all vertices. How can we have a custom color per vertex? We need to pass the colors to the shader during execution. For this, we need to configure input (and output) variables in the shaders.

Receiving Color in Shaders

The first aspect to have in mind is that color **NEEDS** to go through the vertex shader to reach the fragment shader. So, let us start by there. In the vertex shader GLSL code, note that the line `layout (location = 0) in vec3 position;` means that the shader is receiving the vertex position stored in the GPU location 0 (you will see why this is important, later). Add the following line, just below:

```
layout (location = 1) in vec3 color;
```

This means that the color will enter the shader from location 1 (more on this later) and will be stored in variable `color`. Now, as explained the color needs to pass through the vertex shader, so we need to declare an output for color with the line:

```
out vec3 vColor;
```

Then, inside the main() of the vertex shader, we just write:

```
vColor = vec3(color);
```

This just says to take the color in `color` and put it in the output variable `vColor`. So, we are doing nothing to the color, here, just passing it through to the fragment shader.

Now, in the **fragment shader**, the logic is more or less the same. We need to declare the input with:

```
in vec3 vColor;
```

The name `vColor` is important, because it *needs to have the same name* as the output of the vertex shader. This is how the GPU knows it is the same thing.

Finally, the output color of the fragment shader needs to be set to the color arriving in vColor by replacing the current line with:

```
out_color = vec4(vColor,1);
```

This solves the shader part of supporting vertex colors, but now we need to put the data where the shader expects it.

Remembering How to Pass Vertex Data

Go to the rendering code. Remember how we send the vertex data to the GPU with the code already present in `render`:

```
            self.triangle_vbo.bind()
            glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6
* 4 , None)
            glEnableVertexAttribArray(0)
```

This selects the vertex buffer object (with bind), and then tells the GPU to store, in location **`0`** (this is why location, in the shader, is important), an array of data with elements composed of `3` floats (vertex coordinates are 3 floats per vertex), to not transpose the content (`GL_FALSE`), and to jump `6 * 4` bytes to get the next vertex, starting at the beginning (that is what the `None` means).

The 6 * 4 jump, known as stride, comes from how the data is organized in the vertex buffer object. See the comments in the python code.

Finally, we tell the GPU that the attribute placed in location 0 should be enabled (i.e., considered).

Passing Color to Shaders

Now, we need to make something similar to pass the color data to the shader

```
            glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6
* 4, ctypes.c_void_p(3 * 4))
            glEnableVertexAttribArray(1)
```

The code is very similar to the one used for the vertex data. Notice that the location is now 1 and that the beginning is `3 * 4`. Can you understand why? If not, ask for assistance.

Finally, run the script. You should see a colored triangle. Do you understand where you can change the vertex colors? Remember that all the data is set when the vertex buffer object is created.

2.3 Welcome to the Matrix

So, we now have a window with a big colored triangle. But how can we start transforming the way we look at this graphics world and customize where the objects (in this case just our triangle) are positioned? We need to use transformation matrices. These, come in three different flavors: projection, view, and model. Take some time to remember what is the purpose of each.

Yup, shaders, again

All transformations are applied to vertices in the vertex shader. Therefore, the transformation matrices need to be sent to it. First, let us go to the vertex shader and add the matrices as inputs:

```
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
```

One final step to do, in the vertex shader, is to apply the matrices to the vertex position. Basically:

```
gl_Position =  projection * view * model * vec4(position, 1.0);
```

This tells the vertex shader that data for each of these matrices will be put in a buffer of a special type (`uniform`). It basically makes its global to any shader, but we will not get into that. And that is enough of shaders for today.

Let the Triangle Breathe

Look, for instance, for: `def compute_proj_view_matrix(self)`

Now, the code you have already has several methods in place to define the different matrices, but all matrices are the identity matrix. So, let us start making some changes.

By default, OpenGL sets the projection so that the viewport shows data for $x \in [-1, 1]$ and $y \in [-1, 1]$. So, first, we will give some extra room to our triangle by setting a new projection matrix.

In method `compute_proj_view_matrix`, use the following line of code to replace the identity matrix and obtain a matrix for an orthogonal projection.

To work with matrices, we will no define them completely by hand. Instead, we will use some features of the OpenGL Mathematics package (glm).

```
        self.projMatrix = glm.ortho(self.min_x, self.max_x, self.
    min_y, self.max_y, -5.0, 15.0)
```

Note that it uses some variables initialized at the start, in the VCHelper class. Go check it out.

Note that the code already provided includes two extra (very important) lines:

```
        projLoc = glGetUniformLocation(self.shader, "projection")
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, self.projMatrix)
```

These lines sent the matrix into the proper uniform variable, in the shader. The first line locates the uniform named "**projection**" and uses the obtained location to store the matrix. **You will see similar code whenever we need to write a matrix to be accessed by the shader**.

Remember that "**projection**" was the name we gave the uniform for the projection matrix, in the vertex shader

Run the script. Your triangle should have more space.

2.4 Triangle on the Move

Since we have more space, we can move the triangle around. To do this, we can compute a model matrix that translates it to where we want. Check where this is done. At this point, the model matrix for the triangle is the identity matrix. So, nothing is different. Use the following line to translate the triangle one unit up:

```
self.modelMatrix_triangle = glm.translate(self.
    modelMatrix_triangle, 1.0, 0, 0)
```

Note that the first parameter to glm.translate is the matrix, since this function multiplies the current matrix by the translation matrix to translate 1.0 along XX and 0.0 along YY. Can you see the effect in the triangle?

If you notice, at the beginning of the python file, a few variables are defined for the initial translation of the triangle. Replace the parameters of `glm.translate` with these variable names and run the script. Now press the UP key. Where is this being made, in the code? Modify the code to be able to move the triangle in the four directions.

```
self.tx = self.ty = self.tz = 0.0
```

2.5 A Triangle Needs to See Where It is Going

The triangle moves according to your wishes, but it would be nice that it pointed its top tip to the direction it is moving. For this, we need to rotate it properly. To compute a rotation matrix and multiply it by the current matrix, we can use `glm.rotate(...)`:

```
        self.modelMatrix_triangle = glm.rotate(self.
    modelMatrix_triangle, 45, 0, 0, 1.0)
```

This lines just multiplies the current matrix by a rotation matrix rotating the triangle 45 degrees around the ZZ axis (which is pointing towards you, in the OpenGL window). The triangle should now be rotated 45 degrees to the left. Did you add the rotation before or after the translation? What is the difference?

Try to modify the code so that the triangle tip faces the direction it is moving. Use a similar approach as the one for moving it and note that there is already a variable `self.angle` that can be used to store the current angle.

Scale it Down

Finally, we apply some scale factor to the triangle using glm.scale(...). You just need to provide the initial matrix and the three scale factors in X, Y and Z. The syntax is similar to the `translate` and `rotation` methods.

2.6 Give Triangle a Square Friend

At this point, the goal is to add a square to the scene. The procedure is quite similar to the one used to add the triangle. You need to:

- Create the array with the vertex and color data in a vertex buffer object. Note that it needs to be a different buffer than the one used for the triangle
- To render it, you need to move the data onto the GPU, providing the vertex pointers. Do not forget that vertices go in location 0 and colors in location 1
- Enable the vertex attributes (vertex position and color)
- Draw the array

Use the existing code for the triangle as a guide. Do you need to do anything in the shaders?

How can you ensure that the added square stays still while you move the triangle?

2.7 Chaotic Experimentation

Now, a few extra steps to make it more appealing:

- Make the square appear at a random location. Use `random.uniform(a, b)` to generate a float between `a` and `b`
- Animate the square by making it beat like a heart using scale transformations
- Detect when the triangle passes over the square. For instance, use their stored positions and compare the position of the triangle with a small neighbourhood of the square
- Make the square disappear, when hit by the triangle, and a new square appear in a new random position
- Can you think of a nice effect to add when a square is hit?