Visual Computing

Lab Exercises

Samuel Silva

October 16, 2024

Contents

2D/3D Visualizations and Projections

3

This hands on will allow you to:

- Understand how to add some movement to an object
- Interactively manipulate the parameters defining an orthographic projection
- Understand the effects of an orthographic projection in how the scene is presented
- Apply a perspective projection and interactively change some of its parameters
- Add multiple objects to a scene by reusing existing vertex buffers

3.1 Back to the Past

The file `CV_HandsOn_03_Ex1.py` implements a version of the past hands on outcome. Load it and run it. You can move the triangle (cof... spaceship... cof) and when you hit the red square it will appear in another random position.

3.2 Square in Focus

Now, let us just forget about the "game" and focus our attention on the square. To ease this, just modify the projection parameters to narrow the view. Note that the first square that appears is centred at (0,0,0).

Explore the code and check where this is being performed. What are the current projection parameters? Modify the code so that you can change the projection parameters using the '+' and '-' keys. For simplicity just make it symmetrical, i.e., vary the maximum and minimum values simultaneously and to symmetrical values.

The scene is being applied an orthogonal projection through `glm.ortho`. Where?

If you changed them independently, what would happen?

Shaking that Square

Our square is a bit dull. So, now is time to animate it. **What is the matrix to be used for these transformations**? Please note that there is a method `animate()` that is already available for modifying the parameters for your animation ideas.

- Start by applying some variable scaling to the square. Consider the same scale in all dimensions. A simple idea is to make the scale vary, at each frame, by a small amount until it reaches a limit (e.g., 0.25), then invert the variation until it reaches a small value (e.g., 0.15). And repeat...

- Add a continuous rotation around its center. Note that positive Z is pointing towards you, so, this means a rotation around ZZ'
- Now, add a rotation around each of the other two axes.. Ahah!

So, you can see that this square has more to it than it seemed, at start, but it is actually a very awful object, full of holes. What is happening here?

Deep Thoughts

One important aspect when going 3D concerns depth. By default, OpenGL just piles pixels as they come, just looking at the order the different shapes have been rasterized: a triangle that was ordered to be rendered last appears on top of all others. This has the inconvenient that it does not make any sense of depth. If a triangle is drawn last, but is farer than all, it should be rendered behind them, i.e., be partially or totally occluded. For this to happen, we need to activate the depth test. To ths effect, find the method were the OpenGL context is initialized

Look for `def initializeGLWindow`

Adding some Neighbours

The next task is to add a couple more cubes to the scene. You can reuse some of the code that already exists to render them. However, please remember that you need to apply individual transformations to each, if you want them to appear in different positions. This means a different **[you name it]** matrix for each. Place the two new cubes relatively to the existing cube: a) the first, 0.8 to the left and 4.0 back; and b) the second, 0.8 to the right, 0.5 up, and 7.0 to the back.

Now visualize the result and adjust the view with the '+' and '-' keys, as implemented earlier, in a way that lets you see the triangle again. Can you bump into the cubes and do they all change their position synchronously? Good. But, do you notice something odd about **how you see the cubes relative to each other**? Why does this happen?

3.3 Putting Things Into Perspective

It is now time to move into a projection that provides a more natural look to the rendered scenes by mimicking the effect of distance in how we perceive objects. To this end, you need to replace the current projection matrix (which considers an orthographic projection – `glOrtho`) with a perspective projection. Fortunately, the `glm` module already provides a method for that:

```
self.projMatrix = glm.perspective(self.fovy, 1,  self.
nearPlane, self.farPlane)
```

The parameters used for the projection are defined at the beginning of the Python script. One very important aspect to take into account is to ensure that the objects we want to see are inside the view frustum defined by the perspective projection. If you kept to the recommendations of where to place the cubes and use the default values for the different variable, you should be OK.

Interactive field-of-view

One easy parameter to manipulate, without any major issue, is the field-of-view (stored in `self.fovy`). Before experimenting with the value, just picture what you expect to see for different values. Then, modify the Python script to enable changing it up and down with the 'A' and 'Z' keys.

3.4 Fifty Shades of Colour and Blending

Now let us experiment a little bit more with the shaders. There is a shader where the colour is set. Which one is it? How many components has the defined colour? What does each of them mean? And the last one stands for what? What do you expect if you change it? Give it a try! What happened?

Alpha Centauri

So, nothing happened? The last parameter in the color definition does not represent what you thought, or does it? In fact, the 4th parameter corresponds to alpha (where you right?), i.e., the opacity of the defined colour ranging from 1.0 (opaque) to 0.0 (completely transparent). However, considering the opacity is something that is not enabled by default in OpenGL, since it turns rendering a little bit more costly (why?). You need to enable what is called blending: if something is semi-transparent, its color blends with those colors seen through it. To do it, you need to add `glEnable(GL_BLEND))`. However, nothing happens, still. This is because we need to specify how the blending is performed. To implement transparency, we need to add this line:

Look for `def initializeGLWindow`

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

There are many options for blending, but we will not explore them. Some good explanation of what this line means is available in https://learnopengl.com/Advanced-OpenGL/Blending. Ask if you did not understand the logic.

Alpha Party

Now, let us add an animation effect that randomly changes the transparency of the square. How can it be done? Some hints:

- in each frame, generate a random alpha value
- add a variable to the proper shader to receive a value from your script
- obtain the shader location of that variable, in your script, and fill it with the generated alpha
- change the shader code to consider the passed alpha value

Moving Colours

Can you colour the objects with the position of its vertices? For that, you need to work on the shaders. First hint: you need to have the vertex positions available on the fragment shader.

3.5 Chaotic Experimentation

- Compute the framerate. Remember what the dt = mygl.clock.tick(60) line is doing
- Fill the scene with 50 animated cubes randomly placed in the world