COSC3000 Computer Graphics Project - Durak!

Alex Siryj

May 25, 2023

1 Overview of the Project

1.1 Project Aims and Objectives

Every culture has its games. People have a need for fun. It's a simple fact. This project will be my attempt to satisfy the need for fun in a way that pays tribute to my heritage as a Ukranian.

Durak is a Ukrainian card game involving two or more players. The work "durak" means "idiot" in Ukrainian. The loser of the game is the durak. Unfortunately, the rules are very complicated. After starting the project, I soon realized I would not be able to implement the game due to time constraints and technical difficulty. Put differently, I am not Dennis Ritchie. Instead of creating a full fledged game, I chose to render a carousell of cards in PyOpenGL.

2 Project Design, Development, and Difficulties

2.1 Development

This subsection will cover the development of the carousell. It is important to note that much of the code has been adapted from examples found online.

2.1.1 Vertex Shaders

The vertex shader processes geometric data and passes information to other stages in the graphics pipeline. In our vertex shader, we have

1. `vPos` - the local coordinates of fragment we are drawing.
2. `cardTexCoord` - texture coordinates that will be used in the fragment shader.
3. `proj_matrix` - the projection matrix used to pass coordinates from view space to clip space.
4. `time` - the current time in seconds.
5. `model` - a 4x4 matrix used to position and orient the model in 3D space.

6. `fCardTexCoord` - texture coordinates to be used by the fragment shader.
7. `localRotation` - a time dependent local rotation matrix around the y-axis.
8. `globalRotationMatrix` - a time dependent global rotation matrix around the y-axis.

2.1.2 Fragment Shaders

The fragment shader processes color data. This is where we "paint" textures onto the screen.

1. `color` - the color of the fragment.
2. `fCardTexCoord` - texture coordinates.
3. `back` - the texture on the "back" of the card.
4. `front` - the texture on the "front" of the card.
5. `suit` - the texture containing the suit of the card.
6. `value` - the texture containing the value of the card.

To determine whether we are painting the "front" or "back" of our card, we can use the `gl_FrontFacing` boolean. If we are painting the back face, then we call `texture(back,fCardTexCoord)`. Otherwise, we paint the face value and suit "layers" on top of the front texture.

2.1.3 Loading Data Into Our Shaders

Step One: Creating Vertex Array, Vertex Buffer, and Element Buffer Objects

We first need to create vertex array objects, vertex buffer objects, and element buffer objects to store some of the data needed by the vertex and fragment shaders. There are $4 \times 13 = 52$ cards, so we need 52 of each (except the element buffer object since we are really just repeating the same drawing procedure 52 times).

Step Two: Loading in Data to the Vertex Array, Vertex Buffer, and Element Buffer Objects

First, we bind the VAO we are associating data with, then we bind the VBO, send it data using `glBufferData` and `glBufferSubData`, and similarly for the EBO.

Step Three: Enabling Vertex Attributes

We need to tell OpenGL where to find each of the attributes in our vertex array object, so we call `glVertexAttribPointer` followed by `glEnableVertex - ArrayAttrib`.

Step Four: Associating Samplers With Texture Units

Since we have several different textures associated with a single vertex array object, we need to tell our fragment shader which texture is which. First, we have to associate each uniform sampler with a texture unit. We do this by retrieving the uniform value of each sampler using `glGetUniformLocation` and then setting their values using `glUniform1i`.

2.1.4 Loading Images Into PyOpenGL

Step One: Generating Texture IDs

The first step to load images into PyOpenGL is to create a texture ID. To do so, we can use the following command:

```
glGenTextures(n)
```

where `n` is the number of texture IDs we would like to create. If $n = 1$, the command returns an integer. Otherwise if $n > 1$, we get an array of integers.

Step Two: Binding a Texture ID

The next step is to bind our newly created texture. This "activates" our texture as a 1D texture, 2D texture, 3D texture, or whatever target we specify in the first argument of `glBindTexture`. Any commands affecting the target we specify in `glBindTexture` will affect the texture most recently bound to the target.

```
glBindTexture(target, texture)
```

Step Three: Getting Image Data

Our texture data is stored in PNG files. I chose to use the PIL image library to open and read image files. Opening an image is as simple as calling

```
image = Image.open(filePath)
```

where `filePath` is the path to our image.

Step Four: Loading in Texture Data

There are a few different ways we can load texture data into our target. The simplest is to use the following command:

```
glTexImage2D(target, level, internalFormat, width, height,
        border, format, type, data)
```

where `target` is the texture target, `level` is the mipmap level of our texture, `internalFormat` indicates how the texture data will be stored, `width` and `height` are the width and height of the texture, `border` is always 0 for some reason, `format` specifies the format of the pixel data, `type` is the format of each pixel component, and finally `data` is the data to be loaded into the target.

PyOpenGL is a bit fussy about the type of variables begin used in commands. We cannot directly use our image object in the `data` argument. Instead, we must use `image.tobytes()`.

Step Five: Closing the Image File

We no longer need the image object, so we can close it by writing

```
image.close()
```

Step Six: Generating Mipmaps

The final step which I do not completely understand is to generate mipmaps for our texture. We do so by calling

```
glGenerateMipmap(target)
```

What is surprising to me is that this is not optional even though in early stages of development, this command was not present and textures rendered perfectly fine.

2.1.5 Rendering Our Texture

Step One: Set the Time Uniform

Our vertex shader contains a time uniform which we set using `glUniform1f`.

Step Two: Clear the Screen

This step is self explanatory.

Step Three: Bind Textures

For each card, we need to call `glActiveTexture` followed by `glBindTexture` to tell the fragment shader which texture to use.

Step Four: Set the Model Matrix

This step is used to position the cards in view space.

Step Five: Binding the Element Array Buffer

This tells OpenGL the order to draw stuff.

Step Six: Drawing the Data

To draw the data onto the screen, we call `glDrawElements` with the appropriate arguments.

2.2 Difficulties

OpenGL is not the easiest library to use, but it is a lot of fun once it finally works. Most of the code available online is targeted at C/C++. This course uses Python. It was a challenge at times to convert C++ code to Python (particularly arrays, matrices, and pointers).

3 The Finished Product

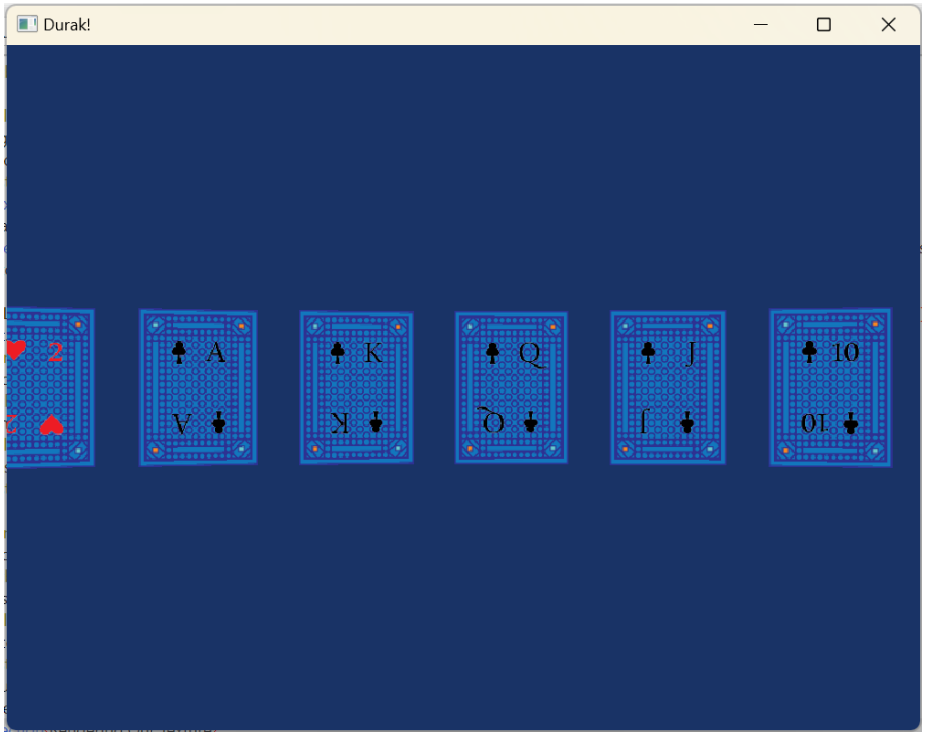

Figure 1: The final product. Cards spin around their local axis and the global axis.

4 Self-Evaluation

4.1 Evaluation Metrics

Character	Geekspeak	Emoji	Plainspeak
A	**A**wesome	😍	The project is exceedingly good.
B	**B**umpin'	💪	The project is pleasing.
C	**C**ool	😎	The project is acceptable.
D	**D**aft	😟	The project is unsatisfactory.
F	**F**ail	🤦‍♂️	The project is worthy of ridicule.

Figure 2: The very objective and serious evaluation metric.

4.2 The Final Score

The project is **B**umpin'. In my opinion it even meets the criteria for being **A**wesome, though I understand that **A**wesomeness is a subjective matter and not everyone agrees on exactly what constitutes **A**wesomeness.

4.3 Future Development

When more time becomes available, I would like to complete a full implementation of durak.

5 References

1. Kessenich, John; Sellers, Graham; Shreiner, Dave. OpenGL Programming Guide. Pearson Education. Kindle Edition.
2. Sellers, Graham; Wright, Richard S, Jr.; Haemel, Nicholas. OpenGL Superbible. Pearson Education. Kindle Edition.
3. https://stackoverflow.com/questions/19993078/looking-for-a-simple-opengl-3-2-python-example-that-uses-glfw
4. https://stackoverflow.com/questions/74408404/sending-a-pointer-to-glvertexattributepointer-in-pyrthon
5. https://learnopengl.com/code_viewer_gh.php?code=src/1.getting_started/4.1.textures/textures.cpp
6. https://github.com/tito/pymt/blob/master/pymt/graphx/shader.py
7. https://learnopengl.com/Getting-started/Coordinate-Systems
8. https://github.com/Zuzu-Typ/PyGLM/issues/1
9. https://community.khronos.org/t/what-is-a-texture-unit/63250