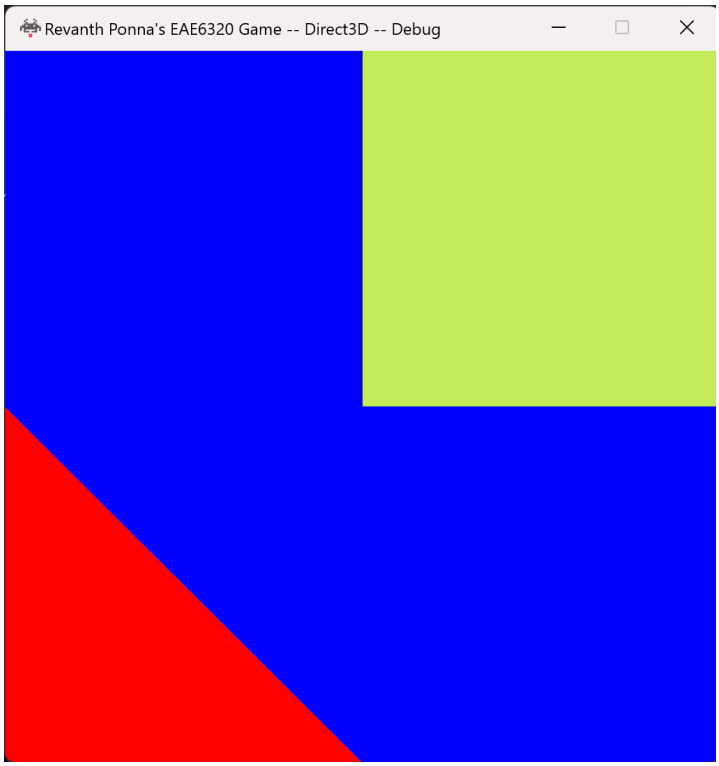
Write Up

Assignment 04 - GAMES6320

[Download Game]

Controls: Space Bar - Hide Mesh, Tab Key - Change Effect

During the completion of Assignment 04, I learned several concepts surrounding threading, the reference counting method, and how Graphics engines communicate with games. I also understood the importance of making a clean and easy-to-use interface for game programmers to pass in game-specific data for rendering.



Default State Of Game

Submitting Background Color

For the background color, I created an interface where the name of a color can be submitted to set the background for the next frame. I implemented this by first finding the Color Hex Codes for various colors online, ranging from

Yellow to Pink. Then, I created a function in the Graphics engine to convert these Hex Codes to RGBA values. This way, a user can simply pass in the name of a color to be set as a background without specifying any RGBA values from the game. Although I could have chosen to submit the background in a much simpler way, I found this part of the assignment to be a fun exercise and also resulted in a neat interface.

```
// Setting background color to blue
Graphics::SetBackgroundColor(Graphics::Blue);
```

Reference Counting

This part of the assignment was probably the one that took the most time, but also where I learned the most new concepts. Before this class, I had no idea how threading worked in a game engine. Although I was vaguely familiar with the concept, the inner workings of threading were unknown to me.

Going through the steps outlined in the assignment helped me understand how to use reference counting as a method to keep track of data, making sure one thread does not delete pointers when another thread is working on it. During this part, I had to considerably revamp my Mesh and Effect classes to accommodate this new behavior. For example, I had to move most of my public functions to now be private, which allowed new instances to be created and destroyed by only incrementing and decrementing the reference count respectively.

One obstacle I had during reference counting was with my Effect class. At first, I wrote the DECLAREREFERENCECOUNT() function right at the end of the class, after all the member variables were defined. However, this led to problems where MyGame was just a black screen with nothing being rendered. After debugging and researching the cause of the issue, I realized that the s_renderState variable should be outside the reference counting, since the render state is responsible for managing the continuous rendering of frames.

Memory Usage

On a 64-bit system, the sizeof() a mesh currently is 32 bytes on Direct3D and 20 bytes on the OpenGL platform.

```
43              // Geometry Data
44              //
45
46      #if defined(EAE6320_PLATFORM_D3D)
47              eae6320::Graphics::cVertexFormat* s_vertexFormat = nullptr;
48              // A vertex buffer holds the data for each vertex
49              ID3D11Buffer* s_vertexBuffer = nullptr;
50              // Index Buffer
51              ID3D11Buffer* s_indexBuffer = nullptr;
52      #elif defined(EAE6320_PLATFORM_GL)
53              // A vertex buffer holds the data for each vertex
54              GLuint s_vertexBufferId = 0;
55              // A vertex array encapsulates the vertex data as well as the vertex input layout
56              GLuint s_vertexArrayId = 0;
57              // A index buffer holds the index to refer to from vertex buffer for rendering
58              GLuint s_IndexBufferId = 0;
59      #endif
60              unsigned int numIndexes = 0;
61              EAE6320_ASSETS_DECLAREREFERENCECOUNT()
62          };
```

The current layout of data is mostly efficient. However, I could make it smaller by reordering the member variables to minimize padding. For example, in Direct3D, I could place numIndexes and the reference count before the 8-byte pointers, which could save about 4 bytes of padding depending on the platform. On the other hand, I could also add padding explicitly to align my data to larger boundaries.

On a 64-bit system, the sizeof() an effect currently is 56 bytes on Direct3D and 16 bytes on the OpenGL platform.

```
34              // Shading Data
35              //-------------
36
37              eae6320::Graphics::cShader* s_vertexShader = nullptr;
38              eae6320::Graphics::cShader* s_fragmentShader = nullptr;
39
40      #if defined(EAE6320_PLATFORM_GL)
41          public:
42                  GLuint s_programId = 0;
43      #endif
44          private:
45                  EAE6320_ASSETS_DECLAREREFERENCECOUNT()
46                  eae6320::Graphics::cRenderState s_renderState;
47          };
```

One optimization that I could do to make it smaller is combining s_vertexShader and s_fragmentShader into a single struct that holds both shaders. Since these two would almost always be paired together, this could be an effective way to reduce memory usage.

The total memory budgeted for the Graphics project, including the two sDataRequiredToRenderAFrame structs, is 944 bytes. When looking at the data inside the struct itself, I believe all the memory should be included in the calculation. I don't really understand why some of the data would be excluded, but this is something I would like to know more about.

Submitting Meshes & Effects

To submit meshes and effects from MyGame to the Graphics engine, I created a simple struct called MeshEffectPairs, which pairs up each mesh with a corresponding effect. In the MyGame project, a game programmer can easily load different meshes and effects using the factory function. And then, in the SubmitDataToBeRendered() function, the MeshEffectPairs struct would have to be populated and initialized according to the functionality that the game wants.

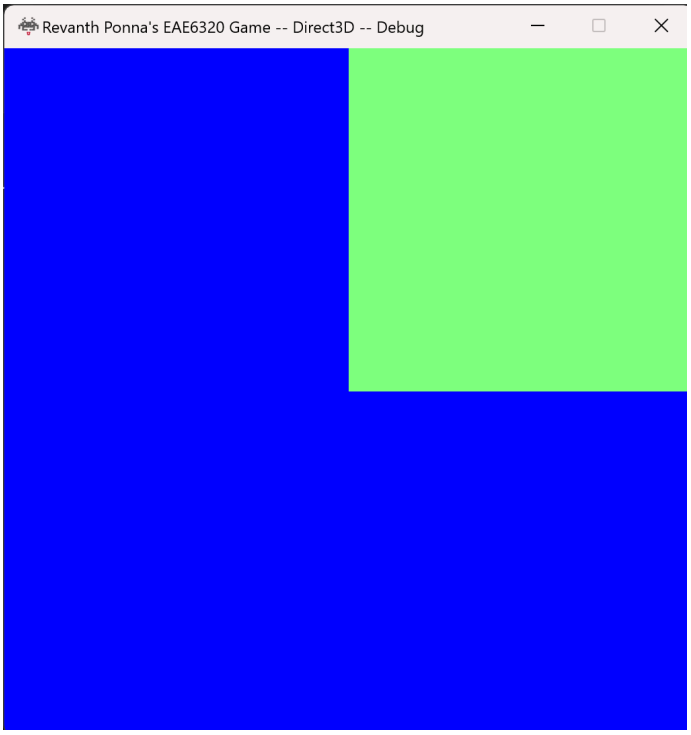
```
159         default:
160             MeshEffectPairs[0].mesh = squareMesh;
161             MeshEffectPairs[0].effect = animatedColorEffect;
162 
163             MeshEffectPairs[1].mesh = triangleMesh;
164             MeshEffectPairs[1].effect = constantColorEffect;
165 
166             Graphics::SetMeshEffectPairs(MeshEffectPairs, numMeshEffectPairs);
```

The reason we are caching all of the mesh and effect data first is to build a multi-threaded game engine. Since we have two copies of the data required (one that is submitted by the application, and the other that is being rendered), both of them can work simultaneously and stay in sync, resulting in much faster graphic performance and smooth results. Of course, this approach comes with a trade-off with memory, since we are duplicating all the data to be rendered. Hence, we are first caching all the data, and then waiting for the other copy to finish rendering the previous frame.

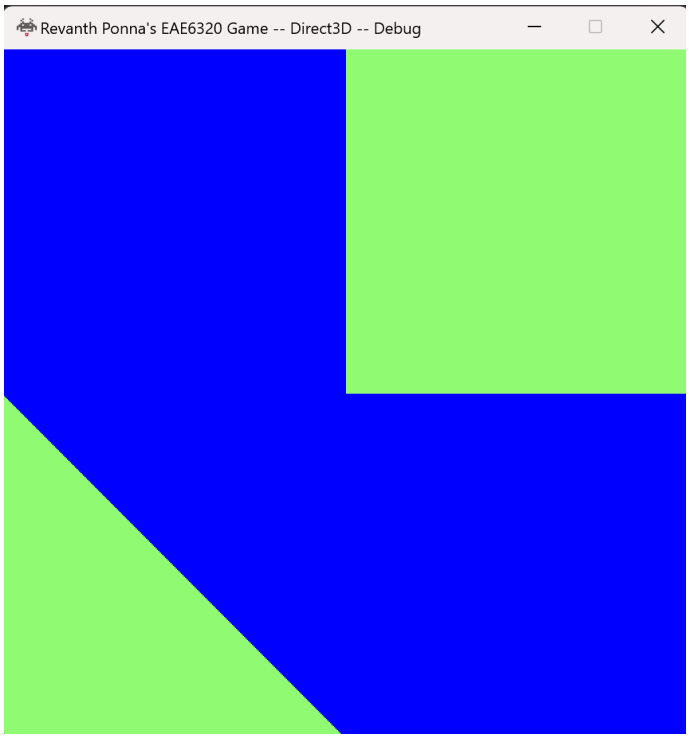
Responding To Input

For the input keys, I chose Space Bar to hide one of the meshes and only render the other. Additionally, the Tab key is used to change the effect on one of the meshes, and animate its color from the default red.

The screenshots of these behaviors can be found below!



Triangle Not Rendered When Pressing Down Space Bar



Triangle With Animated Effect When Tab Key Is Held Down

[Download Game]