Explanation of concepts; Group 11

The team:

Isaac Taylor - Main Programmer - 100785267

Mithunan Jayaseelan - Main Graphical Programmer - 100787442

Spencer Tester - Main Artist - 100653129

Game Choice:

We chose to recreate the game Space Invaders, originally released in arcades in 1978. The game has a specific player character that moves at the bottom of the screen and has the ability to shoot bullets, a set of walls that act as barriers for the player to hide behind. There is a set of enemies that move down from the top of the screen and shoot bullets back at the player. In order to win, the player must shoot all the enemies one time at minimum, and the player loses if any enemies reach a certain low point on the screen or is shot enough times to lose all their lives.

Student Number Addition:

100785267
+
100787442
+
100653129
= 18, which is even

Note: We are working off of the code base given to us by Sage.

Note: Please play the game in windowed mode, in full screen there is some unintended visual effects

Explanation of Concepts:

Select the playable character and use it to explain the graphics pipeline Associated with the Vertex Shader, Geometry Shader, and Fragment Shader:

Vertex Shader:

The vertex shader is split among five different files.

To start we define the vertex buffer, allowing us to define the input slot, size of the buffer, and the type of data to be passed in. We also define if we should normalize data, where in the element this specific buffer is, and lastly the ability to mark what we will be using this specific buffer for, just to keep note for our own usages.

Once this is completely processed, we can turn the 3D model we have into code that the computer can read and utilize.

Geometry Shader:

While we do not use a geometry shader for our player, the concept is still relatively simple.

We define a set of inputs and outputs for our shader to allow us to take in data, convert it to another shape or form, and then push it into the main program. We can use this for particles, or anything we need to dynamically place or change the shape of.

Fragment Shader:

Our fragment shader allows us to take the data given by our vertex or geometry shader, and display that on the screen. For the objects in our game, we can use this to apply a specific texture to our model and then figure out where to place the fragments of the object on the screen, as the fragment shader converts objects and colors to RGB values on the screen.
Our invader_frag shader(a fragment shader) has a struct called InvaderMaterial that stores multiple shaders. We then create a uniform of the struct and pass all the textures to the uniform struct. Based off an integer value which is also a part of the InvaderMaterial struct a texture is then selected and passed to frag color.

Explain how the Phong Lighting model allows you to create a metallic feel for objects in the game:

The blinn_Phong shader allows us to change how a specific object interacts with lighting. In the instance of creating a metallic object, we can make an object very reflective, by editing the texture of the model we can emulate light reflecting off of an object. We then push the resulting texture to the fragment shader to be processed as a fragment.

Explain what approach allows us to create a winter feel using shaders:

We can use shaders to add a tint to the environment, a simple blue. This could be done by increasing the blue value of a fragment from the fragment shader prior to rendering it on screen. Additionally we can use the phong lighting model to add some reflectiveness to the environment, which in addition to some snow particles and an effective ground texture lets us create an effective “cold” feeling environment.

Explain how to implement a dynamic light that changes the effect of the scene:
How did we implement it:
We first created a new component layer which we called TextureChange. A component layer consists of a .cpp and .h file. The .cpp file has an update loop which we use to make changes to objects in the world and have them change in real time. In order to access lights we must first attach the component to an object in the game world. Accessing lights from the scene is quite simple with the following code:
GetGameObject()->GetScene()->Lights[0]
From there it was a matter of updating the light’s position to continually move across the screen. This was done by using the lerp function. One thing to keep in mind is that in order for changes to lights in the scene to be rendered this function must be called in the update loop:
GetGameObject()->GetScene()->SetupShaderAndLights();
SetupShaderAndLights() essentially rerenders the scene’s lights in order to reflect the changes to the light’s position.

```
if (currentTime <= maxTime)
{
    currentTime += deltaTime;
    float t = currentTime / maxTime;
    glm::vec3 test = lerp(glm::vec3(-40.f, 0.f, 3.f), glm::vec3(40.f, 0.f, 3.0f), t);
    std::cout << test.x << " " << test.y << " " << test.z << "\n";
    GetGameObject()->GetScene()->Lights[6].Position = test;
}
else
{
    currentTime = 0.f;
}
```

The dynamic light movement code.

We have implemented a dynamic light that travels horizontally across the screen. Our intention behind this light is to mimic that of a shooting star traveling across space. The white light traveling across the dark space background helps to make the world appear more alive and thus make the playthrough slightly more immersive.