Haider Ali - 100707414 - Artist, Technical Art, Writer
Jayce Lovell - 100775118 - Programmer

Our game - The core gameplay loop of frogger, with a static movement plane (No logs on water) and moving obstacles such as cars. The game is set at night, with a few street lights illuminating the level. The challenge will be to maneuver through the level while the lights are on, as they will flicker on and off. If the player is hit by a car, they lose and are sent back to the beginning. If the player can get to the other side of the screen, they win.

Old Repo Link (NOT Clean, includes all of OTTER):
https://github.com/JayceLovell/Intermediate-Computer-Graphics-Midterm

New GitHub Link (Includes Submission):
https://github.com/JayceLovell/Intermediate-Computer-Graphics-Midterm-Submission

Student Number Last Digit Sum: 18 - Even (This number was based on when we had a third member of the group, however it is still even after his departure part way through the exam)

Playable Character Graphics Pipeline:

Vertex Shader: The vertex shader for our player character takes a variety of input values, including the vertex positions in local space, the normal values, etc., as well as some uniforms for various matrices, current time, our camera position in world space. It then starts by calculating the vertices in clip space, using the Model View Projection Matrix which was calculated on the CPU. This operation is done per vertex, with each MVP Matrix unique to each object. These new vertex positions are set to gl_position, which is then used to assemble our shapes defined by meshes.

We also use the Model Matrix to pass the world space position to our frag shader, and our normal matrix to process and send our transformed normal values to the frag shader. Finally, we pass along the UV Coordinates used for texture data. These are unaltered.

The geometry shader is not being used on this character, and thus does not process our data.

The Fragment shader calculates the color values for our scene per fragment. It will take in vertex data in world space, transformed normal data, and UV data from our vertex shader. We also access a uniform that contains material data, including a 2D Texture to sample for color data and a shininess value. Our normal values are normalized, and lighting values are calculated based on the input position from world space, the normal value, our camera position, and the material's shininess. Then the texture is sampled based on the UV coordinates from our vertex shader. The UV Data is interpolated from our vertices, so we aren't just getting the color value of the closest vertex. Then, the lighting (including ambient, diffuse and specular) and texture’s RGB values (the albedo) are combined to give us the fragment color, with the alpha value of the fragment dependent on the texture’s alpha value.

Metallic Feel: The Phong Lighting Model allows us to include a shininess value, which is used as an exponent when calculating our specular highlights. By increasing this exponent, we get a larger specular highlight, thus giving our object shading that imitates a metallic material.

Dynamic Lighting: To create a set of dynamic lights in OpenGL, we must update the position of the lighting or disable it. For our purposes, it will be more effective to simply move the light far enough away that it will hardly affect our scene due to its distance. We can use this to simulate dynamic lighting for the streetlights of our game, which will flicker on and off.

As lighting is calculated in the fragment shader, we can check there to see lighting attenuation. Lighting strength on a surface perpendicular to the light ray is proportional to the inverse square of the distance between the surface and the source.