Intermediate Computer Graphics Midterm - 03/09/22

Game: Frogger

In our version of frogger we have decided to make the player character a round and rotund raccoon. You as the player need to roll him along past cars, which drive around and serve as obstacles within the game as they try to impede your progress. To win the game, you as the player must reach the goal before the time limit expires. And you lose if one of the cars hit you before you can reach it.

Patrick Thompson - 100791335
Jake Calvert - 100788373
Jeffrey Li - 100712344

5 + 3 + 4 = 12 => <u>Even number</u>

Intermediate Computer Graphics Midterm - 03/09/22

Explanation of Concepts:

<u>The Graphics Pipeline:</u>

The graphic pipeline represents the process that one must go through in order to take an art asset such as a 3D model and have their game render it correctly within 3D space. One of the most important aspects of the graphics pipeline is when the vertex shader, geometry shader, and fragment shaders are implemented into the code so that they can help to shape and render all 3D objects in the game. Take the main character for example. When we load in the raccoon's model, the first thing we need to do is map out where all of its vertices are. The vertex shader comes in at the '3D Primitives' step in the pipeline. We take the mesh sent over by the art team and render the vertices at the correct point in space using an algorithm. Doing this allows us to better apply any transformations and effects to the model later, such as shading and lighting, since we know where each vertex is and how it'll affect its environment. The geometry shader is the next important part. Now that we've gotten the original position and number of vertices from the vertex shader, the geometry shader is tasked with redefining this. The geometry shader takes the vertices handed to it by the vertex shader and can do a number of things while redefining the primitives. This can be as simple as moving them around a bit, or as advanced as creating more vertices based around the originals handed to it. Since this happens almost immediately after a model is loaded in, the geometry shader is brought in at the 'Model Transformation' stage of the graphics pipeline. The geometry shader is how you go from a set of three points, to a fully drawn triangle for example. In terms of the raccoon; a geometry shader transforms his mesh by connecting each and every point as is required so that his body takes a shape we can more easily recognize as a living creature. And finally there's the fragment shaders. These shaders take the shapes handed to them by the geometry shader, and begin to colour and shade as is needed. For every triangle generated by the geometry shader, the fragment shader uses an algorithm to figure out which colour that specific triangle needs to be in order to look best with the applied texture. Without the fragment shader, our little raccoon would be just an empty wireframe model, or he'd be rendered completely black. This part in the graphics pipeline tends to happen around the 'Lighting' phase: since we need the colours applied to the models to truly tell if the lighting effects are going to work.

<u>Phong Lighting Model:</u>

The Phong lighting model is a diagram used to show how one can implement localised lighting on a surface to give it the appearance of a metallic texture. The model works by applying localised lighting to a metallic surface whenever light would hit it, giving it the appearance that it's reflecting light back at the camera. While modern computer graphics have now developed a way to actually trace the rays of light to the surface they'd most likely hit to cause a more natural reflection (Ray Tracing), the Phong reflection model was amazingly innovative and helped to advance what we could do with computer graphics in order to render more realistic textures in games.

<u>Winter Shader:</u>

The approach to use when attempting to make a shader make a game feel like winter, is to use procedural noise. Procedural noise creates a distortion within 3D space that can cause distortions to any textures that exist in the same place as them. This can have a myriad of helpful effects since it means that anything affected by the procedural noise, even if the same model and texture, can look slightly different from each other (you could make a block of houses look unique, a snowy forest with each tree looking different, etc). The first thing to do when adding procedural noise is to check its size. In order to give it a winter effect, you want the noise sparse enough to see the texture underneath, so you can't have it covering everything. Based on how heavy the snow is meant to be, you'll want to space it out until the model in question appears to be covered in bits of clumped up snow and ice. Once that is done, all you need to do is apply the texture for the model beneath the procedural noise, and the model will look wintery thanks to the procedural noise shader.

Explanation of Implementation:

Dynamic Light:

When implementing dynamic lighting, it is very common to use it to help dictate a day night cycle to the player. The way this works is by changing the tint on the world light so that it goes brom a bright yellow in the day and then as the sun sets, gradient from that yellow to a dark shade of orange. This darker shade of orange gives off the effect of the in-game sun setting, and less light affects people and objects within a game to the player. Then as the sun declines more and more, you want to change the light from that deep orange to now be shaded with a very dark blue. While we know that night time is associated with black and darkness, an easy way to trick the human eye to make the environment still be easily visible is by shading a very dark shade of blue instead of grey or black. This allows vibrant colours to still have their pop without blending into shades of grey, and also looks prettier to most people's eyes. Now in terms of changing how it shades as it moves closer or further from an object, that isn't a big issue with world light since the sun is very far away. However you could code the light to move around the world, thereby creating shadows that stretch and move with the sun rising and setting. This would make the sun feel like a real thing in the world and not just an immovable light hanging in the sky that changes colour as the day progresses.

Implemented Shaders:

The shaders used in this midterm code are ones we made in GDW class. These shaders are straightforward and simple: shading the base colour provided as long as a lighting effect exists within the world. They can create shadows as long as the lighting is hitting an object, and are mostly just a simple vertex and fragment shader.The vertex shader stores some vec4 variables that it can use in order to help it calculate the positions and normals. It then also grabs the base colour data from any textures handed to it to get a better idea of what to hand over to the fragment shader. Once the vertex shader is done compiling all that information, the fragment shader gets the position of the light in the world as well as the camera, so it knows what to shade and how. Then as it runs it's main function, it gets the ambient strength from the light as well as the colour one wishes to use for the light in order to correctly apply the shaded colours to the models and textures. Once that is done, the fragment shader diffuses; using the vectors for the normals, light direction, and light colour to better interpret how to light the entire scene. Attenuation is then used to make sure that the distance for the light to travel isn't causing issues. Finally we use specular lighting to add finishing touches and cause any minor reflections to occur so that the game looks a bit more polished, and then pass through the texture sampler and the UVs so the shading can complete.

Implementation: OpenGL Code