Michael Ghaussy

Misty Rainforest Scene Documentation

**Controls**

The camera is moved left and right by the A and D keys, front and back with the W and S keys, and rotates around the Y axis with the Q and E keys. Additional functionality, such as movement along the Y-axis or mouse controls were not added, as they are not essential for navigating the scene.

Input functionality was added using the base code for Inputclass from Rastertek. Additional functions that were added to the framework are CameraLeftRight(), CameraFrontBack(), and CameraRotateY(). These functions check if the aforementioned buttons that control camera motion have been pressed. 0.0f will be returned if they are not pressed, and positive or negative float values will be returned if one of the bound keys has been pressed. This information is accessed in the Systemclass during the Run() function. An if statement checks if any of the values that indicate a change in camera position or rotation are non-zero. In that case, the ChangeCamera() function in the Graphicsclass is called, with the float values from each input function passed in as arguments. This function will change the camera object’s position and rotation, using SetRotation() and SetPosition() based off the floats passed into the function. The translation of the camera in the X and Z directions is influenced by the rotation of the camera, such that the axes of motion will be set relative to the camera’s viewpoint, rather than the world matrix.

**Sky Dome**

The sky dome logic was taken from the Rastertek tutorial. However, that tutorial created a separate Skydomeclass and Skydomeshaderclass, whereas this implementation instantiates the sky dome as a single Modelclass object. In the Graphicsclass, the Render() function will first translate the sky dome around the camera’s position. Then, back face culling is turned off so that the texture is rendered on the inside of the sphere, and the ZBuffer is turned off so that the dome is always rendered behind the objects in the scene. The sky dome is rendered using the light shader, instead of a separate sky dome shader, because a separate shader would have virtually all of the same properties as the light shader, without specular. Setting the specular color to (0.0f, 0.0f, 0.0f) in the light shader applies the same effect without having to add new shaders or classes. After the sky dome is rendered, back face culling and the ZBuffer are turned back on so the rest of the scene can render. In order to accomplish this, additional functionality was added to the D3Dclass which allows the ZBuffer and back face culling to be switched on and off.

There is a slight issue in this implementation of the sky dome, which is a visible line running up the dome. This is caused by the texture file not wrapping around at the edges, as the texture was taken from a sky box and is meant to only display on the front face. This problem could be fixed by either selecting a more appropriate texture for the sky dome or changing the UV values for the sky dome model. Also, although the fog shader uses the color from the sky box to render the fog, the overall effect is not a smooth blend to the background, and models rendered with fog stick out a bit too much in front of the sky dome.

**Loading Models**

In rendering this scene, complex models such as trees had to be loaded in. Tree and wall models were taken from free3D.com as .obj files, and the Rastertek converter was used to make .txt files for the model loader to read. Because of the nature of the given framework, only one texture file can be loaded with each model. Thus, in order to apply the leaf texture to the leaves of the tree, and the bark texture to the trunk, the tree model was split into two separate models using 3Ds Max. Then, each part of the tree model was loaded in separately with slightly modified UV mapping to apply the textures in the correct positions. The Modelclass was modified slightly to include SetPosition() and GetPosition() so as to quickly place the models in set positions in the scene. SetPosition() takes place in the Initialize() function in the Graphicsclass, and GetPosition() is used to render the models in the correct location.

**Treeclass**

In order to efficiently render several trees in this scene, a separate Treeclass was created. The tree model has two instances of Modelclass objects: m\_Trunk and m\_Leaves. In the Initialize() function for Treeclass, the trunk and leaves models are initialized in the same way as in Graphicsclass, except that position floats are passed in as parameters such that each tree has a determined location upon initialization. The Shutdown() function acts in the same way as the Graphicsclass. This formatting allows for the two models that make up the tree to be created and dealt with in the same place in Graphicsclass without excessively repeating lines of code.

As many trees need to be rendered in this scene, repeating initializing and rendering code for each tree would be inefficient. Thus, the Graphicsclass contains an instance of a TreeClass pointer list, m\_Trees. In the Initialize() function, m\_Trees is populated with the tree objects, and the Shutdown() function will clear the list. In Render(), the system iterates through each tree in m\_Trees and renders each trunk and leaves of the tree in the same way that models are rendered.

**Fog**

The logic for the fog shader was taken from the Rastertek tutorial, but instead of creating a separate vertex and pixel shader for fog, the code was added to the light shaders. In the vertex shader, a fog buffer with start and end positions is passed in, and the output to the pixel shader includes a fogFactor variable that is determined by the vertex’s distance in the Z axis to the camera (relative to the view matrix). Then, the texture of the model being rendered is combined with a float4 which represents the color of the fog. This leads to the effect that farther away objects will have a greater fog effect, whereas closer objects will have little to no fog effect. The LightShaderClass has also been modified so that the SetShaderParameters() function includes another data pointer to a FogBufferType. This buffer pointer only contains the stand and end positions that are passed into the vertex shader. In this code, these values are set here, whereas in the Rastertek tutorial they are set in the Graphicsclass. That implementation gives the ability to alter the fogStart and fogEnd variables dynamically during the rendering process if necessary, but as this scene does not need that functionality it uses a simpler method to set these variables. In addition, the InitializeShader() function now also includes a fog buffer description to allow the buffer logic to work.

**Sounds**

Sounds were implemented in a separate Soundclass file and utilized in the Systemclass. The original structure was taken from the Rastertek tutorial with small changes. First, in LoadWaveFile() code that checked for stereo format, sample rate and 16 bit format was removed, as it interfered with the audio that was added to the scene. The Soundclass object is initialized in Systemclass’s Initialize() function, and loads the .wav files into two secondary buffers and plays them. The background sound is played on a loop at a set volume. The footstep effect is created by initializing the .wav file at the minimum volume, and checking in the Systemclass whether the camera is being moved in Frame(). In frames where the camera is moving, the SetFootstepVolume() function from the Soundclass is called, setting the volume to maximum, and in frames where the camera is not moving, the function sets the volume to minimum. In this way, the footstep sound is playing constantly, but setting the volume at certain times gives the impression of footsteps when moving.

**Normal Mapping**

Normal maps were added to give a greater degree of detail to the tree and wall models in the scene. The images for the normal maps themselves were taken from <http://cpetry.github.io/NormalMap-Online/>, using the texture image files for each model. The logic for normal mapping was taken from Rastertek, but instead of creating a new shader, as in the fog shader, it was simply added to the light shaders and the Lightshaderclass. In order to accomplish this, tangent and binormal variables are added to the PixelInputType. These values are used to calculate the normals for each model rendered in the vertex shader, which are used to determine the light intensity in the pixel shader. In the Lightshaderclass, the input type layout has changed to allow these two new values. In addition, the Textureclass must be changed in order to store both the texture image itself as well as the normal map. This was done by changing m\_texture to an array of size two, m\_textures, then passing in the texture and normal images in the Intialize() function. This required some slight modifications in the Modelclass, as code dealing with the texture object had to be refined, as well as CalculateModelVectors(), CalculateTangentBinormal(), and CalculateNormal() added as additional functions in order for the shaders to work.

**Conclusion**

If time permitted and I were to continue work on this project, I would work to improve the sky dome such that the visible lines where the texture ends are removed, and the fog effect so that the area around the models but in front of the skybox more closely resembles the fog color that the models are rendered with. In addition, I would try to add shadow mapping, so as to give a more realistic lighting effect with shade beneath each tree. I attempted shadow mapping in this scene but issues with the depth shader prevented its implementation. Having begun this class with no C++ or DirectX experience, I now have a working knowledge of both. In terms of the development pipeline, I am now much more familiar with the various aspects required to graphically render a scene. At the beginning of this class, I was completely unfamiliar with shaders, textures, buffers, but having worked on this scene over the course of the semester, I am able to understand and utilize the relationship between these elements to construct a functional scene in DirectX 11.