REPORT – PROGRAMMING FOR GAMES

SEBASTIAN POSKITT-MARSHALL

1. Feature Summary

The following features are showcased in the submitted project;

1. A controllable camera system to navigate the scene
2. Loading in and texturing of a simple model
3. Textured objects affected by lighting, with additional normal mapping and specular mapping to improve the lighting effects,
4. Generation of terrain through the use of height maps
5. Manipulation of textures on an object
6. Implementation of a skybox
7. Rendering the scene to a texture
8. A particle effect system generating geometry outside of the geometry shader
9. An FPS and CPU monitor
10. Sound within the scene

2. Controls

Keyboard - W	Move camera forward
Keyboard – S	Move camera backward
Keyboard – A	Rotate camera right
Keyboard – D	Rotate camera left
Keyboard – I	Rotate camera upwards
Keyboard - K	Rotate camera downwards

3. Features

3.1. Camera Implementation

In order to properly navigate the created 3D scene, a way of controlling the camera had to be implemented. The controls implemented are simple in nature, whilst pressing the A or D keys on the keyboard, the camera is rotated around they axis to either the left or the right respectively. When Pressing the I or K key will orient the camera around the x-axis, either angling it upwards or downwards. Using these 2 sets of keys, the user is able to look around the scene.

The camera can be moved forwards and backwards when the W and S keys are held respectively. The way of achieving this is slightly more complex than the other 2 movements, as the forward and backward direction change based on the cameras orientation. In order to

figure out this direction, the rotation around the y-axis is converted from degrees into radians. Then using this converted value, the camera is moved along the x axis based on the sine of the radian value, and along the z-axis based on the cosine of the value. This allows the camera to move forwards, or backwards, no matter how it is oriented in the world.

3.2. Simple Textured Model

The project makes use of a simple model loading system to create more complex geometry in the scene by loading from a while, instead of defining all geometry in code. As the techniques for achieving this were illustrated in the labs, this report will not go into detail on the processes required to get this object into the scene, however, it is important to note this feature, as it is used as the base of many other features illustrated in the project.

3.3. Textured Object using Normal Map and Specular Map

The scene features a sphere mapped with an array of textures to achieve an effect of realistic lighting. In order to achieve these effects, several techniques were implemented and combined.

3.3.1. Texture Arrays

The first technique that needed to be implemented was the use of an array of textures, instead of a single texture, stored in each instance of the model class. This array can store a user defined amount of textures that the program can then access simply when needed. In order to achieve the lighting effects making use of the normal and specular maps, the model demonstrating the techniques would have to store 3 individual textures, the main colour texture, the normal map to define the objects new normals and the specular map to define the "shininess" of each part of the object. The texture array class provides a simple way to integrate this into the model itself, without the need for creating multiple instances of a texture for each model requiring it.

3.3.2. Normal Maps

Normal mapping is a technique which allows us to create surfaces in the 3D would that appear to interact realistically with the light as if they were made of a certain material. Compared to creating complex high polygon geometry to achieve this effect, normal mapping provides a far cheaper and simpler way to achieve the same effect. The techniques used to implement normal mapping in the project were taken from the Rastertek DirectX11 tutorials.

As seen in figure 1, normal maps are images that are treated the same way as textures by the program. However, instead of mapping the texels of the texture to the object, the red, green and blue values of each texel of a normal map represent the angle of the normal they represent on the object. In order to figure out what texel of the normal map should be used for the normal on the object, the tangent and binormal must first be calculated. The values of these 2 represent the tu and tv coordinates on the normal map that are used for that point on the object. Once these coordinates have been figured out, we can use the values fetched from the normal map as the normals of the object, and calculate our lighting using them to achieve a realistic effect.

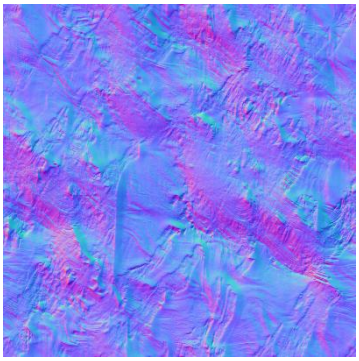

Figure 1 - An example of a normal map

The bulk of the work is done within the normal map shader, which has been merged with the specular map shader in the project. The model class has also been updated with a function to calculate the normal and binormals needed for the shader, depending on the cameras position, and is then passed to the shader along with the other information from the model, in which the final normal are finally calculated using the normal map attatched to the model and the calculated tangents and birnomals.

3.3.3. Specular Maps

Specular maps work in a similar way to the normal maps discussed above. Just as normal maps are saved as an image, so too are specular maps. However, instead of storing the direction of normal as an RGB value in the image, specular maps are gray scale, as they only need to store a single value relating to the specular intensity of each part of the texture.

Figure 2 - An example of a specular map

This can be used to build on top of the simple specular lighting discussed in the labs which was limited due to the fact that we couldn't specify what parts of the object are the most reflective. Through the use of a specular map, it is possible to define how "shiny" parts of the geometry it is applied to are, allowing for more realistic effects when texturing objects. Furthermore, just like the normal mapping technique, it is a computationally

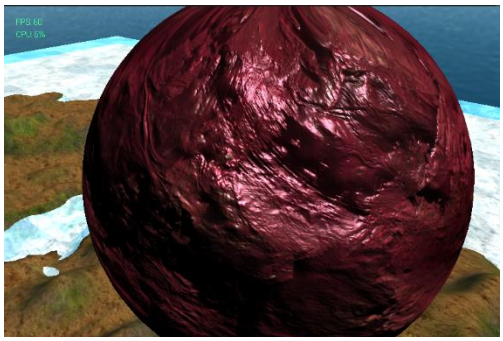

Figure 3 - An example of geometry mapped with both a normal map and a specular map

inexpensive method for creating a realistic looking surface without having to create higher polygon geometry.

3.4. Terrain

The project demonstrates a simple system for creating terrain from a large plane object. Unlike the other geometry within the project, the terrain is generated entirely in code, allowing its size to be specified when the terrain is created. This allows larger terrains to make use of more polygons, which in turn makes the various techniques that can be applied to terrain more effective. The project shows of various methods for adjusting the terrain, which will be discussed below. The techniques discussed have been adapted from the Rastertek tutorials.

3.4.1. Height Maps

Similar to the previously discussed normal maps and specular maps, height maps (also known as a displacement map), height maps are also stored as images where the RGB values of each texel represents something, in this case, how far the related vertex should be extruded from its initial position.

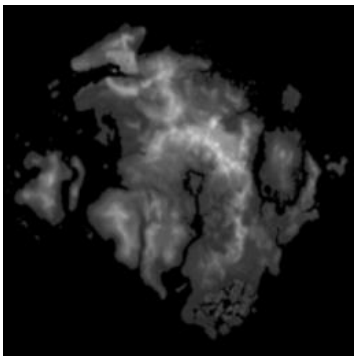

Figure 4 - An example of a height map

This height map is again stored in the texture array structure discussed above. When the terrain is generated, the program also examines the height map image to decide where to place each vertex in world space, based on the RGB values of the image. By adjusting the vertices of the terrain, a realistic landscape can be generated using simple geometry and a grayscale image. Unlike normal maps, the program only needs to read one of the RGB values to figure out the displacement of the vertices. The value obtained from the RGB value directly corresponds to the position of the vertex needed. The results of this process can be seen in figure 5.

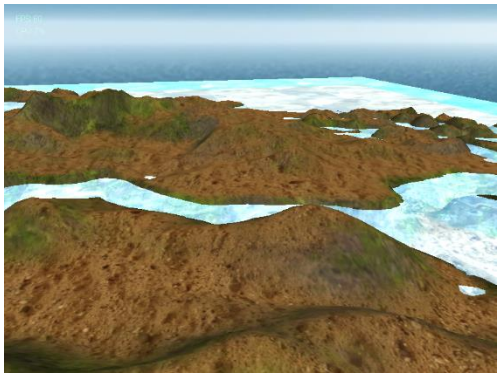

Figure 5 - Terrain generated using height map

This method of generating terrain can be improved however. One such improvement is applying tessellation to the generated terrain. As tessellation increases the amount of polygons in the terrain, the height map would become more accurate and look smoother. This could be improved further upon by tessellating the terrain based on how close it is to the camera, so that terrain that is closer to the camera and needs more detail has it, whilst terrain that is further away has less detail.

The displacement of vertices could also be controlled in the geometry shader, rather than in the main program itself. Whilst this doesn't provide a lot of benefit when compared to the current method, it does work in tandem with mentioned tessellation, as it affords more control over what can be done with generated vertices.

3.4.2. Multiple Textured Terrain

The terrain within the program is textured with multiple different textures to provide a more realistic effect. This again makes use of the texture arrays. The terrain is specifically textured with 3 different textures, a main texture, a slope texture and a peak texture, and the program determines what texture to use for each part of the terrain based on the angle of the slope of the polygon.

The calculations to determine the slope of each polygon are done inside the vertex shader file for the terrain shader. The calculation itself is simple, the normal of the current vertex is taken away from 1.0f, and the resulting value is a value between 0 and 1 that represents the angle of the slope. Once this calculation has been done, the corresponding texture is applied, with values less than 0.2 having the main texture applied, and the values above 0.7 having the peak texture applied. For all values in between these two, the slope texture is blended with either the main texture or the peak texture, giving preference to what texture of the two it is closest to.

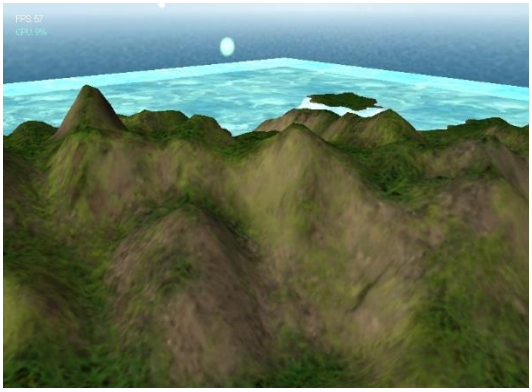

Figure 6 - Example of the slope mapped terrain

3.5. Texture Manipulation

Within the program, texture manipulation using noise is used on the water in order to create a realistic looking flowing water effect. To achieve this effect, a couple of textures are made use of, the main texture of the water, and a noise texture, which is used to distort the water texture to create a flowing effect. The technique to achieve this was adapted from the Rastertek tutorials.

Each frame of the program, the noise texture is translated upwards across the surface of any object it is applied to (in the case of this program, the water plane). The texture itself is not actually visible, as the intention is to use it at the pixel shader stage of the pipeline to distort the main texture of the object. In addition to translating the texture, the texture is assigned a

distortion value as well. This distortion value is a D3DXVECTOR2 which will be used at the pixel shader stage later on to achieve a more realistic effect.

When the translated noise texture is handed off to the pixel shader, the shader samples it in order to attain and tu and tu value needed to sample the water texture later on. This value is then multiplied with the distortion value mentioned earlier. Finally, these values are perturbed and used to sample the coordinates of the main water texture to be used for the current pixel.

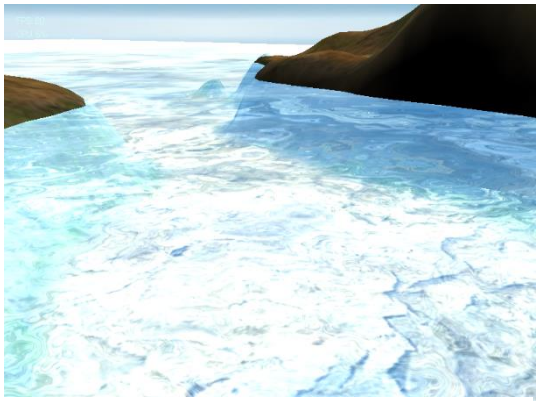

Figure 7 - The water effect in action

Through these steps in the program, a plane demonstrating a realistic flowing water effect can be created. However, these techniques can be applied to achieve several other effects, such as a simple fire effect without the use of a complex particle system.

3.6. Skybox

Sky boxes are often used as a simple and cheap way of giving the illusion of a scene being larger than it actually is. The idea behind them is simple, a small cube (in the case of the project, a 2 unit x 2 unit x 2 unit cube) is drawn at the cameras position at all times. The cube itself is textured using 6 different textures, one for each side of the cube (although these textures tend to be merged into one texture, an example of which can be seen in figure 8). When rendering the skybox, it is rendered first, with the z buffer set to off, so that all geometry rendered afterwards is rendered in front of the box, regardless of its position in world space.

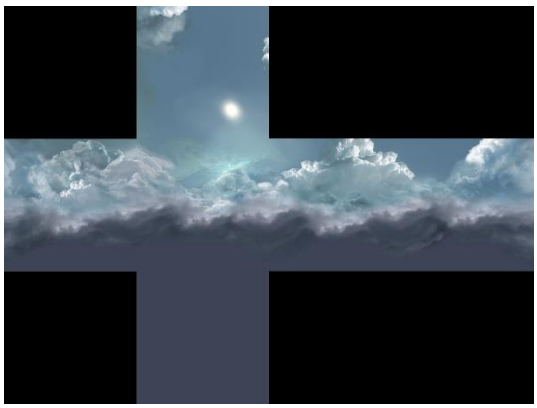

Figure 8 - An example of a merged skybox texture

Using this technique, the scene looks as if it extends far off into the distance, when in actuality, what the user is seeing is a small object built around the camera at all times to give the illusion of an expansive scene. Moving a small box with the camera provides several benefits compared to encapsulating the entire scene in a larger skybox. For one, it is far cheaper to render smaller geometry in this way than larger geometry. Furthermore, as the box moves

with the camera, the user is always cantered within it, and can never reach the “limits” of the skybox, which is entirely possible with the other method.

3.7. Render to Texture

Render to texture is a technique that allows us to render the scene to a texture stored in memory, rather than to the back buffer to be shown on screen. This texture can then be applied to geometry within the world, and can be used for a variety of different techniques, mostly dealing with post processing effects.

Figure 9 - An example of render to texture can be seen in the quad at the upper left hand corner of this image

One of the main shortfalls of this project was the simple use of this feature, as it does not show any of the more complex and interestng effects that can be achieved, due to time constraints. Therefore, some of the imoprovments that could be made to this system within the project will be discussed here.

Possibly the simplest post processing effects that can be done is a simple box blur. This is achieved by rendering the scene to a texture and rendering it to a quad the size of the screen. This textured quad is then passed through a shader 2 separate times, once to blur all of the pixels vertically, then this blurred quad is passed to the second shader to blur them horizontally. The resulting effect is a simple blur that can be used in several different ways to achieve other effects, such as an underwater camera.

The effect that this project aimed to achieve but was unable to due to time constraints was the implementation of an enviroment map. The idea behind the environment map is to create 6 different textures, one facing in each possible direction in the world and mapping them to a cube map texture, similar to the texture discussed earlier with the skybox. This texture can then be mapped to an object based on the cameras position in the world compared to the object using several calculations to determine the correct texture coordinate. After all of this has been done, the resulting object will appear to be reflecting all of the scene around it.

3.8. Particle System

Particle systems generate lots of small pieces of geometry and are used to achieve a variety of different effects in 3d graphics, such as realistic fire and smoke effects. Within the

project, a particle system is used to create the effect of falling lights all around the scene, as a sort of simple weather system.

The particle system contains an array of geometry (in the case of this project, the defined geometry are quads, however this could be any geometry) with a defined maximum size. As the program runs over time, geometry is generated and added to this array, until it hits the maximum amount of particles allowed, at which point the oldest particles in the array are removed to allow the new ones to spawn. These particles are translated downwards over time to simulate falling, and once they hit a low enough point, they too are removed from the array.

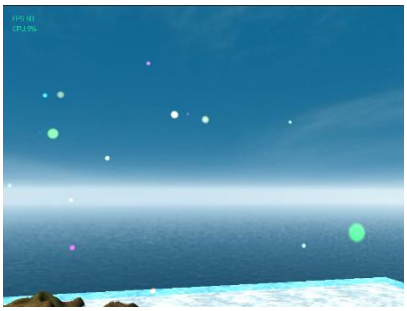

Figure 10 - The particle system in action

The area in which the particles are spawned can also be adjusted to create larger and smaller areas of effect. Within the project, the area of effect is very large, so that it covers the entire scene, however if the aim were to create an effect like a fire or smoke, the defined are would be much smaller. Furthermore, the size of the particles, as well as their lifetime and how many particles can exist at once. This gives a large amount of control over how the particle system works in the scene.

There are some improvements that could be added to the particle system. As it stands now, the particle geometry is generated before the shaders. The array within the particle system could contain points, rather than geometry. When these points are passed along to the shader stage, the geometry shader would be able to generate geometry based on these points, meaning that less information would need to be stored to create the particles.

Another improvement would be to billboard the particles. As it currently stands, the particles all face the same direction in space, meaning that if they are looked at from the side or behind, they are invisible. Bill boarding is a technique which would render the geometry always facing the camera, ensuring that it is visible at all times. With these improvements, the particle system would work from any angle in the scene.

3.9. Sound

The sound implemented in the scene is a simple implementation and makes use of Direct Sound to work. The audio class within the program makes use of a two sound buffers, one that corresponds to the sound buffer on the computers sound card, and the second sound buffer is what the audio file is loaded into in the program.

Although the program only demonstrates one audio file, multiple audio files can be implemented by creating more than one secondary buffer, which will be mixed together within the primary buffer. Furthermore, the audio can be improved by making it 3D. This means that the audio has a position in world space, and based on the listeners (for simplicities sake, the camera will have the listener attached) position compared to the audio's position, the audio is made louder or quieter, and pans accordingly.

4. Shortfalls and Conclusion

The main aim of the project was to implement and experiment with various different techniques that could be used in Direct X11. Whilst several techniques have been demonstrated, there are several that were either scaled back or cut due to time constraints.

Besides the previously mentioned shortfalls, one of the features that was left unimplemented was object selection and manipulation via the mouse. The idea behind the feature was to allow the user to click on an object within the world, showing ray-casting from the mouse to a position in world space, as well as some simple collision detection. This was left unimplemented due to time constraints but would have shown a better understanding of implementing controls into the application, outside of the simple keyboard controls demonstrated.

Another feature that was left unimplemented was a reflective surface on the water, to add more realism to the scene. Through the use of a stencil buffer covering the area of the water plane, it would be possible to render the scene scaled to –1.0f along the y-axis, visible only through the stencil buffer, to achieve a realistic looking reflective surface on the water.

In conclusion, although the project had a few shortfalls, as well as areas in which improvement was possible, the techniques demonstrated within the project illustrate a range of what's possible using Direct X 11, in regards to 3D rendering.

5. References

Rastertek. 2016. Rastertek Tutorial [ONLINE] Available at: http://www.rastertek.com/index.html