Coursework Documentation Alexander Levick

User Controls

Along with the standard controls for the camera, there are added keyboard controls to demonstrate aspects of the application. I will refer to the keyboard letters as ‘A’, etc. ‘M’ and ‘N’ increase and decrease the tessellation factor of the main geometry respectively. ‘B’ toggles wireframe mode for the scene. ‘I’ and ‘K’ increase and decrease the point lights Z position, ‘J’ and ‘L’ the X position, and ‘U’ and ‘O’ for the Y value. ‘G’ toggles the post processing greyscale.

Tessellated Plane

Overview

The main focus of the scene is the large plane. This is a mesh built up of triangles with a control point topology allowing it to be tessellated using the corresponding shaders. The tessellation factor can be altered by the user to manipulate the level of detail on the mesh as it is displaced by a height map. The mesh is coloured according to vertices height and lit by a single point light.

Vertex Shader

The Vertex shader, which is always required, just passes the control points’ position, texture coordinates and normal to the Hull shader. No manipulation is done in the Vertex shader as we do not know if we have all of the vertices.

Hull Shader

The Hull shader has multiple attributes to set for what data it receives and outputs. As the plane is made up of triangles, we set the domain as “tri”, the output topology as “triangle_cw” and to output 3 control points. The Hull shader sets the tessellation factor for each edge and the inside of the patch. This factor is the same for all and is set each iteration by a value that can be altered by the user to increase and decrease the level of detail of the mesh.

Domain Shader

Once a patch has been tessellated it is passed into the Domain shader, where you can manipulate the vertices in any way possible. The vertex position and the texture coordinates are just interpolated from the control points, which only give a mesh more vertices without more visible detail unless using wireframe mode. I then use a height map that is loaded into the Domain shader to displace all of the vertices in the mesh through the Y-axis. This is done by sampling the height map texture at the texture coordinates and storing this pixel colour. As a height map is black and white, each pixels RGB values will all be equal to each other, so using the pixel’s R value, the vertex’s Y position is set as the R value multiplied by an arbitrary multiplier so that the max height of the mesh looks good according to the X and Z dimensions of the mesh.

To calculate the normals to correctly light the mesh in the Pixel shader, a Sobel Filter is used. The Sobel Filter can be used to calculate edges and thus the normals for vertices by using a height map. I followed a good example online by Catalin Zima. To calculate the normal vector you take the colour intensity (black and white so RGB are all the same) of the height map one texel in each direction. These values are then put into the Sobel Filter matrix to calculate the X and Z values of the normal vector. Y will always be the same positive value.

The Domain shader passes the position before and after being multiplied by the world, view and projection matrix. This is so that the Pixel shader has access to the unaltered vertex position for calculating the gradient colour. The normals and vertex position in 3D space are also passed to the Pixel shader for lighting.

Pixel Shader

The Pixel shader colours each individual pixel. The starting colour for each pixel is calculated by taking its Y position, normalising it, and then using it as the U value of a texture coordinate and sampling a gradient texture. This means that the lowest parts of the mesh are red and the tallest parts blue. The lighting is calculated using a point light whose attributes have been placed in one of the Pixel shader’s constant buffers. The amount of light the individual pixel is receiving is found by calculating the direction of light and its intensity. This amount of light is then added onto the pixel colour. Each pixel also receives a small amount of ambient light to represent light reflected from other sources.

Greyscale Post Processing

Overview

A feature of the application is the ability to view the scene in greyscale rather than the normal full colour scene. This is done through post processing. The first step is to render the scene as normal, however instead of rendering the scene to the frame buffer we can render it to a texture since the frame buffer is in essence just an array of pixels, like a texture. This texture is then rendered onto an orthographic mesh which is the exact size of the screen. The orthographic mesh is then put through the greyscale shaders. Again this is rendered to a texture instead of the frame buffer. Finally, the greyscale texture is rendered onto the orthographic mesh used previously and ran through a standard texture shader which just colours each pixel to the texture to be used. This is then rendered to the frame buffer to be displayed on the screen. The resultant image is the same scene however everything has been greyscaled.

Shaders

To create the greyscale effect only the Vertex and Pixel shader stages are required. The greyscale shader’s purpose is to render a texture onto an object with the texture having its colours changed to varying intensity of grey. To achieve this effect only the position and texture coordinates are required to be passed to the shaders. The Vertex shader is only used to pass the position and texture coordinates to the Pixel shader as no vertex manipulation is used in the greyscale shader.

In the Pixel shader the loaded in texture is sampled at the pixel's texture coordinates. This colour is stored. When an image is greyscaled, the more intense a pixel's colour, the closer to white that pixel's greyscale colour will be. The same applies the opposite way where lack of intensity is equal to being near black. To calculate each pixel's greyscale value in the Pixel shader the following formula is used, (R+B+G)/3 = intensity; greyscale = (intensity, intensity, intensity). This creates the correct greyscale colour according to the pixel's previous colour.

Shortfalls

Overview

This section will detail any of the issues with the application, reasons behind said issues and potential resolutions which could be implemented.

Normal Calculation

When calculating normals for the mesh after tessellation and displacement, a Sobel Filter is used to determine the normal vector by sampling the height map at various locations. However in the application there seems to be a slight lighting issue. When altering the position of the light, the pixels of the mesh react accordingly as would be expected. However when moving the light along the Z axis in the positive direction, the amount of light that each pixel is receiving does not seem to decrease despite moving a very large distance. In every other direction including the Z axis in the negative direction the scene darkens appropriately. To solve this issue a stronger grasp and understanding of how the Sobel Filter works would most likely be required as I believe the issue lies within the normal calculation.

Geometry Shader

A final shortfall of this application is the lack of use of the geometry shader. This shader allows the creation and manipulation of vertices, allowing whole models to be built from a single point. The reason for the geometry shader not featuring in the application is simply a lack of time to implement it. However it there was, the intention would have been to pass a point mesh to the geometry shader whose position was equal to that of the light in the scene. 4 triangles or 12 vertices would have then been created in the geometry shader to create a cross shape centred around the input point which would represent the light in the scene.

References

Normals Calculation

As previously mentioned, the code for normals calculation was altered from the original copy by Catalin Zima posted on the website

http://www.catalinzima.com/2008/01/converting-displacement-maps-into-normal-maps/