**CMP301 – Graphics Programming with Shaders – Application Documentation**

**User Guide:**

Movement within the application is controlled using the framework. Forward, left, back and right is W, A, S, D. Upwards and downwards movement is controlled by E and Q respectively.

There are also several user interface elements that can be controlled through ImGui. These include a wireframe mode toggle, tessellation factor, vertex height scale, world matrix scale, specular power, post process toggle and a geometry manipulation toggle.

If the application is to demonstrate the shaders as expected, textures for the terrain, height map and normal map must be placed in the res folder.

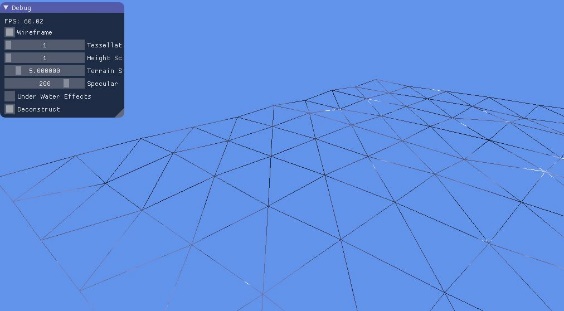
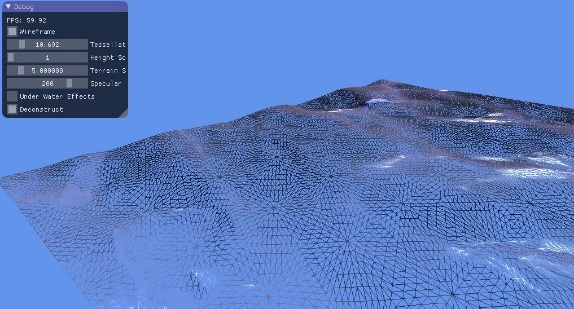
**Overview of Shaders:**

There are three sets of shaders within the application. The first is the scene shader, which handles vertex manipulation, tessellation, geometry manipulation and lighting. The other two shaders are used for post processing. They produce blur and under water effects.

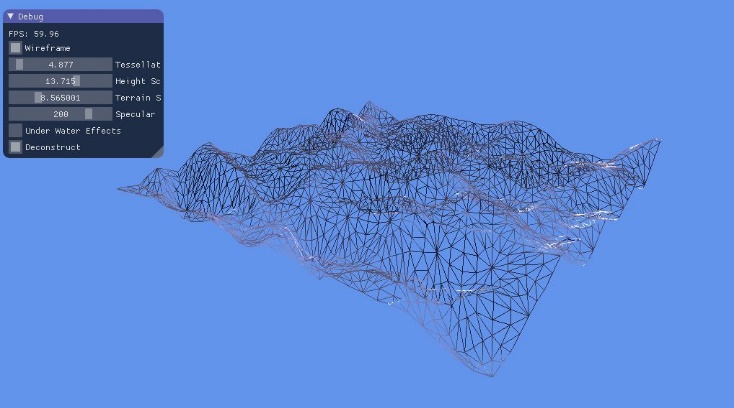
**Scene Shader:**

The scene vertex shader receives a plane mesh position, texture coordinate and normal. The plane mesh is the same as the mesh in the framework, with two changes. The first change is that the resolution has been set to 10 so the plane has less vertices, to demonstrate tessellation. The second change is that sendData() has been overridden to change the primitive topology type to a control point patch list for triangles. The output type for the vertex shader is the same as the input type with two additions, a tangent and bitangent for the position of the vertex, which will be used in the pixel shader to calculate the world space normal of each pixel. As the mesh being used in this shader is a plane, the direction vectors for the tangents and bitangents are in the same direction for each vertex. The tangent vector goes positively down the x-axis. The bitangent vector goes negatively down the z-axis.

The scene hull shader receives a tessellation factor using a buffer in addition to the vertex shader’s output. The tessellation factor for the edges and inside of each triangle is then set to the value from the buffer. From this point onwards, the hull shader just passes on the relevant values to the next stage to be used after tessellation has occurred.



The domain shader obtains the new position, texture coordinate, tangent and bitangent of each vertex, using uv coordinates and patch data. Then, using a height map, the y-component of each vertex is offset by the height data multiplied by the height scale (which is received through a buffer and set by the user). The height data is obtained by sampling the heightmap texture, at the texture coordinate of the current vertex. The height map is only sampled on one colour channel, since it is a grayscale image. The output position is then set to be the offset vertex position.

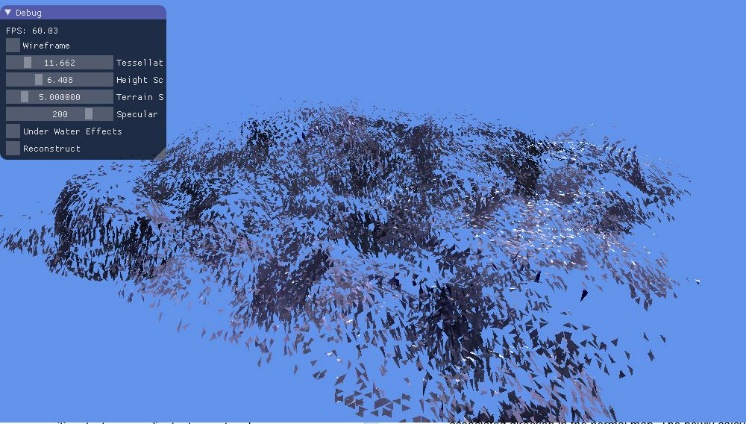


The geometry shader is used in the scene to manipulate the triangle primitives of the terrain based on user input through the GUI. It has two buffers, one containing the world, view and projection matrices and one containing the camera position and a timer. The output type is the same as the input type with one addition, a vector in the direction of the geometry from the camera. This will be used in the pixel shader for specular lighting calculations.

The shader loops through the vertices of the triangle and determines the output values for use in the pixel shader. To calculate the view direction vector, the vertex position is calculated against the world matrix. This is then subtracted from the camera position to give a vector in the direction of the vertex. This vector’s magnitude is then set to one to ensure accurate calculations in the pixel shader. The output texture coordinate is set to that of the input coordinate. The normal, tangent and bitangent vectors are then calculated against the world matrix and normalised. Since these are direction vectors, 3x3 matrices rather than 4x4 are used as there is no translation being applied.

To manipulate the geometry, the output position of the vertex is set to be the input position plus the face normal multiplied by the timer. The output position is then transformed to world space using the world, view and projection matrices. The face normal is used instead of the vertex normal to ensure that the vertices of each primitive move in the same direction as one another. The face normal is calculated by taking the fact that each vertex in the triangle primitive lie in the same plane, so taking the cross product of two direction vectors (calculated using the triangle vertices), results in a normal that is perpendicular to both direction vectors.

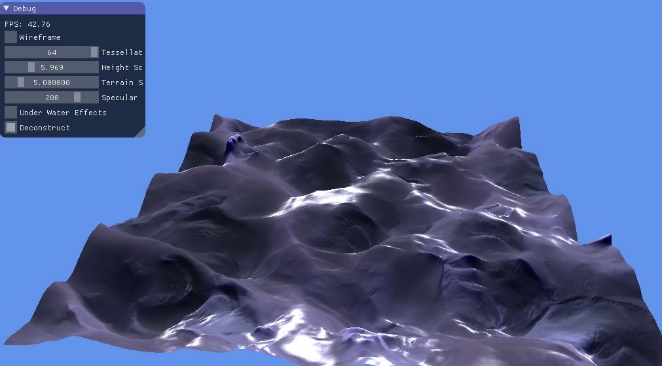
The timer is set externally by user input. If the user selects the deconstruction check box, this will begin incrementing the timer and each primitive will move in the direction of its face normal. If the user selects the reconstruction check box then the timer will start decrementing until it reaches zero. This results in the geometry gradually moving back into its original place. The timer is calculated by adding on delta time each frame in the application.



The pixel shader calculates per pixel lighting using a normal map. As well as the geometry shader output type, it receives two textures (colour and normal map), a sampler and a buffer. Contained in the buffer is the properties of the directional light in the scene. They include diffuse, ambient and specular colour values, the light’s direction and the specular power.

First the normal map texture is sampled to get a direction vector represented by RGB colour values. This vector has normalised components which lie between zero and one. To get these components in the range negative one to one, each component must be multiplied by two, one is then subtracted from that value. After the component range has been corrected, the resultant vector is in tangent space and needs to be converted to world space to be used in the lighting calculations. This is done by multiplying each input vector (normal, tangent and bitangent) by its associated direction in the normal map. The newly calculated world space normal is then normalised for the lighting calculations.

The lighting calculations are like that of the examples provided in the lectures. The colour of the pixel is set to the ambient colour value by default. The diffuse lighting is calculated by taking the dot product of the world normal and the light direction, then saturating this value to calculate the amount of diffuse light on the pixel (the diffuse intensity). It is then checked if there is diffuse light on the pixel and if so the diffuse intensity is multiplied by the diffuse colour and added to the colour value. If the pixel is lit by diffuse light then the specular component is calculated, by multiplying the specular intensity by the specular colour value. The specular intensity is calculated using the Blinn-Phong reflection model. The colour value is then multiplied by the sampled pixel colour from the colour texture and afterwards, the specular component of the lighting is added to the final colour to provide specular reflection whenever the light rays are reflected off the terrain and in the viewers direction.



**Blur Effect Shader:**

The blur shader is a basic box blur similar to the example provided in the lecture except instead of dealing with the nine surrounding pixels, 24 surrounding pixels are used to increase the effect of the blur. If post processing is enabled in the application then the scene will be rendered to a texture called sceneTex.

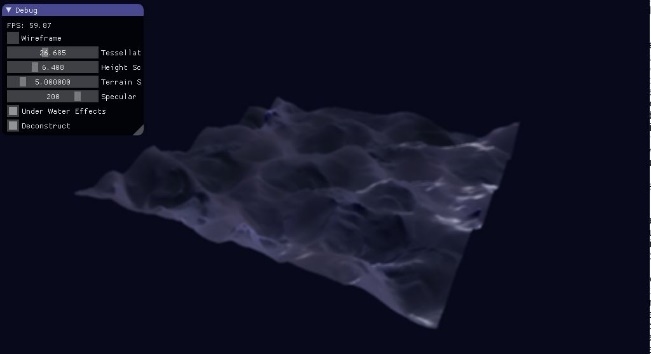
In the vertex shader, the width and height of each texel is calculated using the screens resolution (passed in from a buffer), and the coordinates for the pixel’s surrounding neighbours are obtained and then passed to the pixel shader. The scene texture is then sampled at each coordinate in the pixel shader and each pixel colour is added to the output colour then divided by the number of surrounding pixels. This shader mainly servers to enhance the underwater effect.

**Underwater Effects Shader:**

The underwater shader is a post processing effect that samples the texture resulting from the blur shader. The vertex shader calculates the world space coordinates of the input OrthoMesh vertex then passes this to the pixel shader along with the texture coordinate.

In the pixel shader, each texture coordinate that gets sampled is offset by different amounts in the x and y directions to produce an underwater effect. The offset amount follows the pattern of a sinewave in each direction. Offsetting the x and y texture coordinates is achieved by multiplying the input y and x texture coordinates by a frequency variable, then adding the timer value to that result. The timer is set from the waveTime variable in the application that starts at zero and increments by delta time each frame. The sine of the offset texture coordinates is multiplied by the height variable to reduce the amplitude of the waves so that the scene is still visible to the viewer. Then the offset values are added to the input texture coordinates to give a repeating wave motion in the x and y directions when sampled. Finally, the pixel is darkened then tinted blue by halving the RGB colour values and adding a different small amount to each colour channel.

The post processed scene texture is then rendered to the screen using an OrthoMesh from the framework.



**Application Shortfalls:**

There are many aspects where development is needed to improve the application. These include but are not limited to:

* The scene shader is not flexible with different meshes, it can only handle plane meshes as the tangents and bitangents used in the lighting calculation are fixed in the code in the vertex shader.
  + This could be improved by calculating the tangents and bitangents further down the pipeline to allow for more complicated geometry.
* When lighting the geometry, the calculated normal seems to be unaffected by the height scale applied to the geometry as the specular lighting stays the same when the height scale is increased.
  + This could possibly be due to the way that the tangent and bitangent values are handled within the application
  + To confirm this, the application could be extended by using the geometry shader to render the calculated vectors to make sure they are in sensible directions
* The lighting calculations in the pixel shader could be improved by including multiple lights and shadows, and perhaps real time reflections
* After the tessellation process, the entire plane is tessellated the same amount. This is unsuitable due to the size of the plane, the tessellation factor for one side of the plane will be the exact same as the opposite side.
  + This could be improved by rendering the plane with a dynamic level of detail controlled by the distance between the camera and geometry, by getting a vector in the direction of the geometry from the camera and then tessellating each edge differently based on the distance value.
* The underwater effects produced from post processing the scene could be altered to provide more realistic effects.
  + This could be achieved by implementing volumetric lighting and replacing the box blur with something similar to depth based fog where geometry far away from the camera is less visible based on depth calculations.
  + These could be applied by increasing the number of rendering passes and applying different effect on each pass.