Jamie Haddow
0705082

# CMP301 Report

## Contents:

## Overview

### Outline of coursework contents

The scene contains two models. Those being a box to surround the scene,and a model containing 3  picture frames, which you can see on the left side of the picture below. The models were created inside Maya using simple rectangles. Within the confines of the box, the scene contains a tessellated water plane with wave functionality and a “shallow water” effect. Next, there is a tessellated terrain plane that will be integrated into multiple other scene functionality such as  lighting,shadows and grass. Along with 3 tessellated pictures sitting on picture frames containing a Paul Robertson Easter Egg, the noise map used in a number of calculations, and two students photo-shopped onto the bodies of 'Jump Street 21' actors. The scene contains two directional Lights and two point lights, the latter having a figure of eight movement pattern to display the working lighting and shadow effects. Finally the scene contains grass geometry that can be rendered depending on the height map used for the scene*(figure 1)*.

### Response to coursework brief

Covering the Vertex Manipulation aspect of the brief, multiple objects in the scene have had displacement maps added to them, along with calculated normals and texture coordinates. There are also examples of non trivial tessellation as the increased tessellation factor directly impacts the look of both the water and terrain, as well as increasing the effect of shadows and lighting*(figure 2,3).*

For Post processing, a Depth of field technique was created using the Gaussian blur shown to the students during the blur lab and then extended upon. Two examples of the Depth of field close to the camera and then in the middle of the scene can be seen in the Appendix*(figure 4,5)*.

Two directional Lights and Two point lights were created for this scene. All tessellated planes were correctly calculated to display shadows and lightning correctly*(figure 3)*. Along with the terrain, Geometry was created to represent grass. Each blade consists of 5 triangles*(figure 6).*

## UI elements and Controls

The lecturer created controls for camera movement have not been changed or adjusted. For every object in the scheme, a window has been created using the ImGui framework. A tick box can be found for each category.  These have been summarized in the below table.

| | |
|---|---|
| **Landscape** | |
| Tessellation | Adjust the tessellation value of the terrain |
| Displacement | Adjust the Height of the terrain |
| Height Maps | Choose between 7 Height maps |
| Texture | Choose between 5 textures |
| Grass Objects | Render Grass dependent on Height |
| **Grass** | |
| Speed | Adjust the speed of the grass wind effect |
| Offset | Adjust the offset of the grass  wind effect |
| Height | Adjust the Height of the grass blade |
| Width | Adjust the Width of the grass blade |
| Colour 1 & 2 | Colour picker to choose the desired colours |
| **Water** | |
| Tessellation | Adjust the tessellation value of the water |
| Speed | Adjust the speed of the water waves |
| Amplitude | Increase the height of the water waves |
| Frequency | Increase the number of full waves in a 0-1 range |
| Colour 1 & 2 | Colour picker to choose the desired colours |
| **Picture Frames** | |
| Tessellation 1,2,3 | Adjust the tessellation value of the frames |
| Displacement 1,2,3 | Adjust the Height of the frames |
| **Point Lights & Directional Lights** | |
| Position & Direction | Adjust the position and direction of each light. |
| Attenuation | Point lights can have their attenuation values changed |
| Ambient Colour | Colour picker to choose the desired colour |
| Diffuse Colour | Colour picker to choose the desired colour |
| **Blur** | |
| Range | Adjust the range of the near and far plane blurring |
| Offset | Moves the centre point towards or away from camera |
| Horizontal Strength | Adjusting the weighting on the horizontal blur |
| Vertical Strength | Adjusting the weighting on the vertical blur |

## Techniques demonstrated

### Vertex Manipulation

Both the water and terrain make use of vertex manipulation.

First, the terrain makes use of a height map texture passed in during the SetParameter stage of the pipeline. This is then fed into the shader's hlsl stages through the SetShaderResource function and is assigned a sampler. This process is used for all proceeding textures to avoid repetition in this report. Both the water and terrain planes are tessellated, as such, these pipeline steps will be discussed here rather than later in the report.

After the Vertex shader stage, the hull shader will handle the tessellation of the terrain plane. Using quads, 4 outside edges and 2 inside edges are correctly assigned a value that is passed in during the SetParameter mentioned previously. A dynamic tessellation was implemented which then proceeded to pass this information to the domain stage of the pipeline. With provided code by the lecturer, new Vertex's, and texture coordinates were generated. With the height map plane now being tessellated, the provided code for normal calculations by the lecturer would no longer work.

Before the new normal values could be calculated the vertex position.y needed to be changed. The position.y element is offset taking a colour sample from the passed in height map, and multiplied by the displacement value*(figure 7)*, a variable passed into the shader to be adjusted using Imgui.

During week 5's height map lab work, knowing the size of the plane, offsetting the texture coordinates was a simple task of doing 1/size of plane. For example “float2(1.0f / 100.0f, 0.0f).

For a tessellated plane, the newly generated edges would not follow this pattern, even more so when the tessellation is dynamic. To solve this problem. During the hull shaders output stage, the tessellation calculation is passed into the domain shader. This can be seen in figure 11. In the domain shader, using this tessellated value, an average tessellation value was created using the four patch points.

Once again, the texelscale of the plane was calculated(in this case 1.0/50.0), then finding the tagent and bitangent values of the current position, and finally using a cross product, a new normal value could be calculated(*figure 8*).

The water planes process is very similar until the domain stage. Unlike the terrain where just a single value was passed in, multiple variables including math functions were introduced as indicated by appendix *Figure 9.* Building on the manipulation lab work, by using time along with speed,frequency and amplitude a single direction wave was created. After random inputs were tested, the final wave calculation was decided on.

## Lighting

There are four lights in the scene, these being two directional and two point lights. The directional lights are set above the scene and can be seen in the top left and top right ortho meshes. All four lights contain Ambient and Diffuse lighting, however, both point lights also allow additional attenuation. All the functionalities were transferred from the week 3 and 4 lab work. All Lights also cast shadows which will be discussed next

## Shadows

For calculating shadows, multiple render textures have been used. Each light has its own shadow pass function inside app1.cpp. For each pass, the lights view matrix is generated, this information is then assigned to the view,projection and world Matrices. The scene objects are passed in using a depthShader pass, retrieving a full black and white image of the scene, orthographically for the directional light and projectively for the point lights. For the point lights, 6 directional render textures were required due to the need to calculate each direction from the centre point of the light. These render textures are then passed into the shadowShader inside the RenderScene() function.

Before the shadow textures can be used, they must be transformed by Transposing the current view and orthographic/projection matrices of each light. Once done, they can be passed into the vertex shader to then be transformed into screenspace coordinates and then passed into the pixel shader stage.

Inside the shadow shader's pixel stage, all of the depth maps have been passed in as Texture2D's along with the required light information stored inside the light buffer (ambient,diffuse,attenuation etc.). These can now be sampled, the current texels be read with the aid of the lecturers provided functions and the correct colour values can be outputted to display the shadows of objects where the current lights view position depth value is less than the sampled textures depth value and then light calculations be done.

## Tessellation

For the scenes tessellation, quad meshes were used to allow 4 outside edges and 2 inside edges. A custom dynamic tessellation was attempted having the closer you are to the scene, the more tessellated a plane becomes*(figure 10 & 11)*. This will be mentioned in the critical reflection section of the report.

To calculate the dynamic tessellation, the average position of the input patch was calculated, then using the cameras position and the average position, a distance variables was obtained. This was then used in Clamp function for all the inside and edges to keep the tessellation between 1 and 10(any more than 10 was found to be unnecessary)*(figure 12)*

## Geometry

For fulfilling the geometry requirements, a blade of grass was created using the geometry stage. Using a tessellated plane's vertex position, a field of grass was obtained depending on the tessellation factor of said plane.

Following the previous stages mentioned above for the tessellation, the information of the new vertices were passed into the geometry shader stage. From here, a number of data packets were also sent in from the shader.cpp. Three Texture2D's, a matrixBuffer,a cameraBuffer and a grassBuffer. The matrix buffer being used to convert the geometry to screen space, the camera buffer being used if the user wished to use billboarding on the grass and then the grass buffer containing a number of variables used to transform the blade of grass through Imgui sliders.

A Position buffer array was created and manually assigned texture coordinate locations to allow each triangle strip to be textured/coloured correctly. For the actual blade of grass, five triangles were created*(figure 13)*, each vertices location being linked to input[0].position.xyz and having to be positioned manually*(figure 14 & 15)*

To create the grass, along with the wind affect visible in the scene, a number of different texture were required. The HeightMap passed in, is the same Heightmap used for the terrain displacement. Having it linked to the terrains displacement, allowed grass to be rendered at a specific height by taking the r value of the heightmap and then checking it against a float variable passed into the grass buffer accessible in the landscape Imgui window. If the passed in variable is great than the current heightmap.r value, then the blade of grass is rendered.

Next, a noise map was used to offset the grass blades original position. Without a noise map, visually, the blades of grass would render in a square chunks with a gap between each 'chunk'*(figure 16)*. With the noise map, this issue was overcome by sampling the noiseMaps.r value and then creating a float 3 using the r value as the x and z component to then be passed in during the vertices creation*(figure 17 & 18)*

## Post processing

For post processing, a Depth of Field, ' Bokeh' blur effect was created. To create this, the following steps were used.

First, a render texture of the unblurred scene was required. Inside the RenderScene Function, the texture “renderSceneRenderTexture” was set as the render target and the scenes object were created. Next, this render texture was passed into horizontal blur and vertical blur passes. This was done twice to double the blurring effect. The now 'verticalBlurRenderTexture,' along with the unblurred scene's render texture are passed into the DoFPass function. A camera depth pass, carried out before the shadow passes for the lights  was also passed into the Depth of Field function. These three render textures were fed into the “depthOfFieldShader”, along with other variables, DoF distance, Offset and the screens near and far plane data to then be used in that shaders Pixel stage.

For the depth of field to work properly, the centre of the scene needed to be calculated using the cameras depth render texture. This was achieved by instead of passing in the input.tex to the Sample, a float2(0.5,0.5) was used instead. This idea was taken by reading the details on Circle of Confusion and focal points from the GDC 2004 – 'Advance Depth of Field” pdf

Next, The center texel and depthvalue from the depth Texture were multiplied by the near plane minus the far plane. This gave two resulting value that could be used in the next calculation, where those values, plus the offset were then dividied by the range and passed through a math ABS function to return only a positive value, then a math saturate function to clamp that value between zero and one.. Finally a lerp was carried out between the unblurred scene, the blurred scene to achieve the desired effect.

The post processing effects can be seen in figures 4 & 5 in the Appendix section of the report

## Critical reflection

For this module, the difficulty was a step up compared to second years openGL module, it is now clear why the lecturer choose to use the older OpenGL's fixed pipeline. Many new concepts were covered in this module which were both interesting and challenging. After speaking to a number of industry programmers, it is clear that shadows, tessellation, shaders are all used on a daily basis and are a core part of a programmers knowledge.

Debugging practice and knowledge has increased due to this module and some limitations on how to interact and retrieve the data sent to the GPU along with some strange issues that cropped up during development such as after creating a new shader class, the application would not continue after the first frame. The importance of packing in packets of 16 bits is again reinforced along with last terms discussion with Chris Trewartha on low level Cache hits and misses related to correctly packed structs.

Inheritance is used heavily in this project using the framework provided and creating new classes which again is another important technique that needs to constantly be built upon.

Knowledge of external graphics debugging tools such as RenderDocs and Visual studios graphics debugging became necessary to debug any issues with incorrectly passed in data/textures/samplers. This knowledge will potentially be useful in the industry depending on what software the company uses. Use of the RenderDocs resource Inspector would have been interesting to develop further, this is something that could be researched into next year.

Dynamic Tessellation was a specific area of the project that was not implemented correctly. After  researching both Frank Lunas, “3D Games programming” method(19.2.3) and pages 424-426 of 'Practical Rendering and Computation with DX11” the correct implementation was still not reproducible. This is something that will be researched more due to being a very important correction before entering the industry.

Correctly shadowed geometry was also another area that was not creatable within the time frame of this submission. On the first attempt, the problem stemmed from the domain shader receiving the shadows view and projection matrices and then passing those into the geometry shader. However, with the number of inputs sent in, the geometry input would only allow 16 inputs as seen below(figure 18).

All shadow and light data was then passed directly to the geometry pixel shader to avoid this problem, however again, the correct solution was not solvable within the time frame allowed.

Additions to the application at a further date to improve this project as a portfolio piece will include more impactful water generation, specular lighting on the water, corrected dynamic tessellation, a more complex Depth of field using the “Nvidias Depth of field GPU gems chapter”

## References

Paul Robertson – usage of TextureShader, shadow functions and base vertical and horizontal blur shaders

Digitalerr0r. 2009. XNA Shader Programming. [ONLINE] Available at: https://digitalerr0r.wordpress.com/2009/05/16/xna-shader-programming-tutorial-20-depth-of-field/ [Last Accessed 04/12/2020]

Scheuerman. GDC. 2004. Advanced Depth of Field. [ONLINE] Available at: https://developer.amd.com/wordpress/media/2012/10/Scheuermann_DepthOfField.pdf [Last Accessed 04/12/2020]

Frank D. Luna. Introduction to 3D Game Programming with DirectX 11.0. Chapter 19 – Terrain Rendering. [BOOK]

Jason. Zink. Practical Rendering And Computation with Direct 3D 11. Chapter 9 Dynamic Tesselation. [BOOK]

http://markusrapp.de/wordpress/wp-content/uploads/papers/Adaptive%20Terrain%20Rendering%20using%20DirectX%2011%20Shader%20Pipeline.pdf

## Appendix: Figures

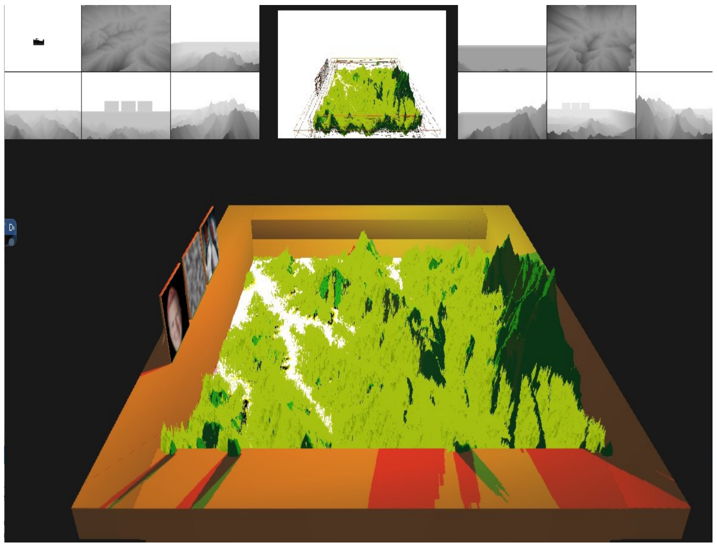


*Figure 1: Scene overview*

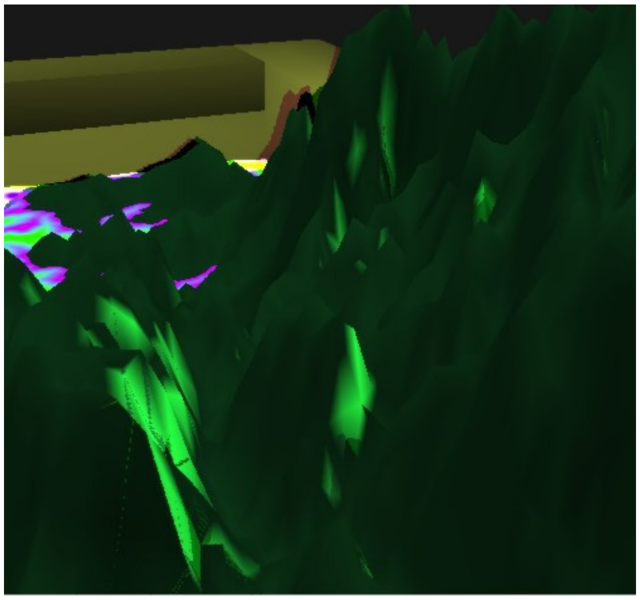


*Figure 2: Non tessellated terrain*

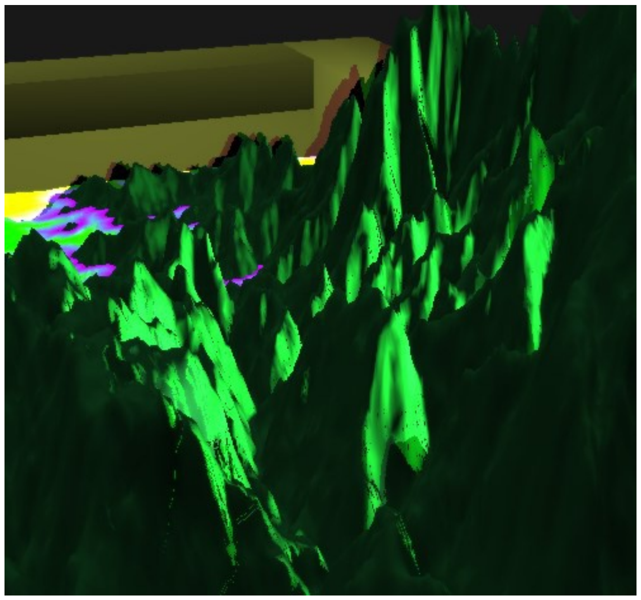


*Figure 3: Tessellated terrain*

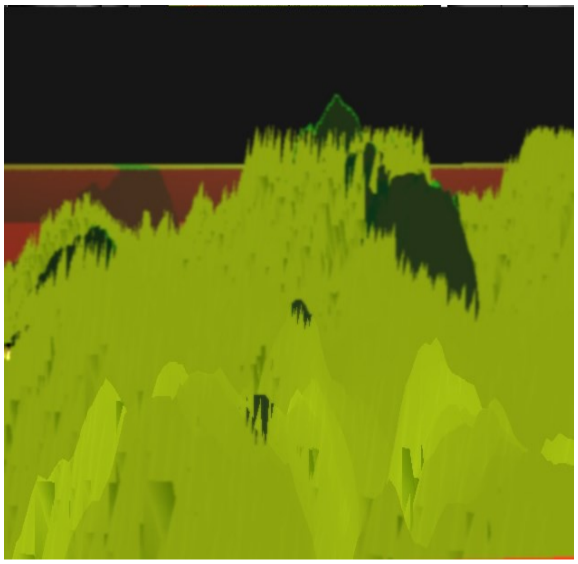


*Figure 4: Near the camera*

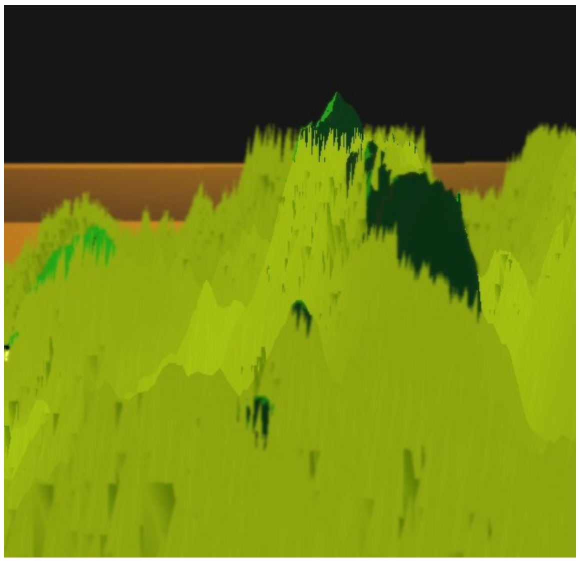


*Figure 5: Center of screen*

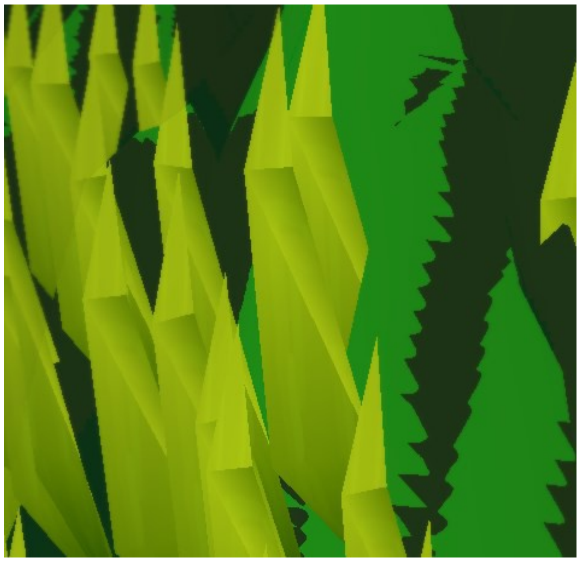
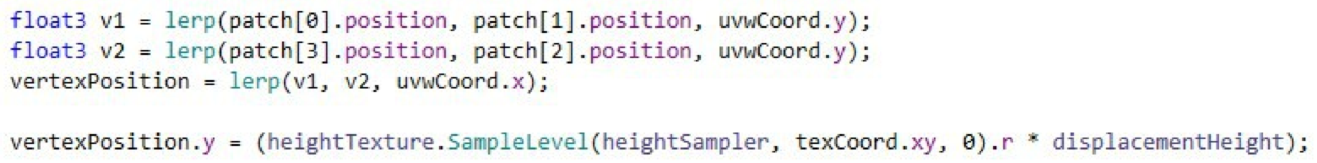
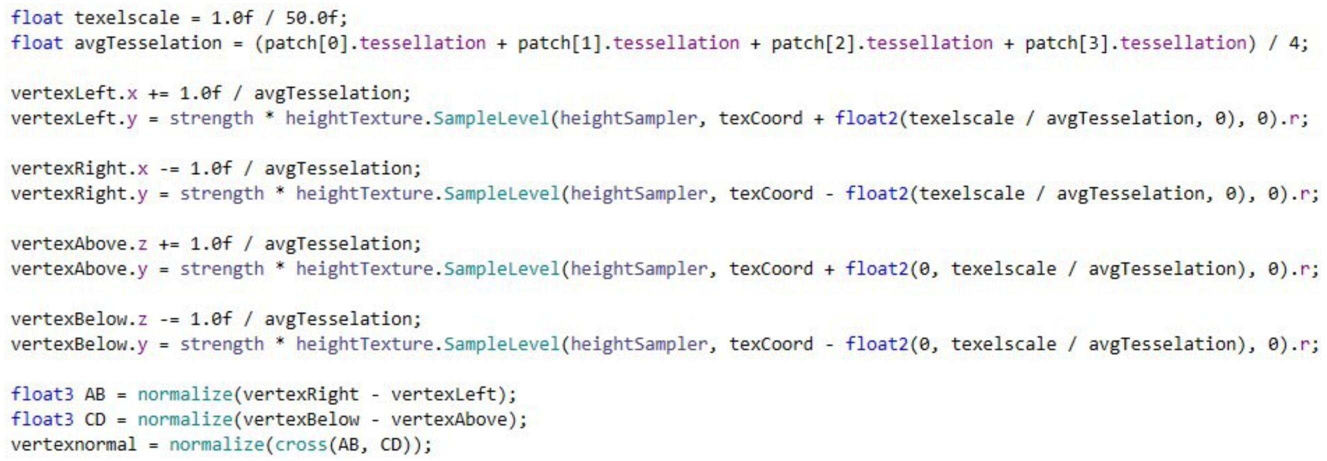


*Figure 6: A single blade of grass*



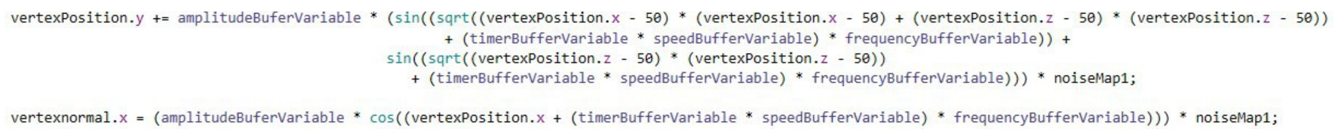
```
float3 v1 = lerp(patch[0].position, patch[1].position, uvwCoord.y);
float3 v2 = lerp(patch[3].position, patch[2].position, uvwCoord.y);
vertexPosition = lerp(v1, v2, uvwCoord.x);

vertexPosition.y = (heightTexture.SampleLevel(heightSampler, texCoord.xy, 0).r * displacementHeight);
```

*Figure 7: Terrain displacement calculation*



```
float texelscale = 1.0f / 50.0f;
float avgTesselation = (patch[0].tessellation + patch[1].tessellation + patch[2].tessellation + patch[3].tessellation) / 4;

vertexLeft.x += 1.0f / avgTesselation;
vertexLeft.y = strength * heightTexture.SampleLevel(heightSampler, texCoord + float2(texelscale / avgTesselation, 0), 0).r;

vertexRight.x -= 1.0f / avgTesselation;
vertexRight.y = strength * heightTexture.SampleLevel(heightSampler, texCoord - float2(texelscale / avgTesselation, 0), 0).r;

vertexAbove.z += 1.0f / avgTesselation;
vertexAbove.y = strength * heightTexture.SampleLevel(heightSampler, texCoord + float2(0, texelscale / avgTesselation), 0).r;

vertexBelow.z -= 1.0f / avgTesselation;
vertexBelow.y = strength * heightTexture.SampleLevel(heightSampler, texCoord - float2(0, texelscale / avgTesselation), 0).r;

float3 AB = normalize(vertexRight - vertexLeft);
float3 CD = normalize(vertexBelow - vertexAbove);
vertexnormal = normalize(cross(AB, CD));
```

**Figure 8: New normal calculations**



```
vertexPosition.y += amplitudeBuferVariable * (sin((sqrt((vertexPosition.x - 50) * (vertexPosition.x - 50) + (vertexPosition.z - 50) * (vertexPosition.z - 50))
                                                + (timerBufferVariable * speedBufferVariable) * frequencyBufferVariable)) +
                                              sin((sqrt((vertexPosition.z - 50) * (vertexPosition.z - 50))
                                                 + (timerBufferVariable * speedBufferVariable) * frequencyBufferVariable))) * noiseMap1;

vertexnormal.x = (amplitudeBuferVariable * cos((vertexPosition.x + (timerBufferVariable * speedBufferVariable) * frequencyBufferVariable))) * noiseMap1;
```

**Figure 9: Water displacement calculation**

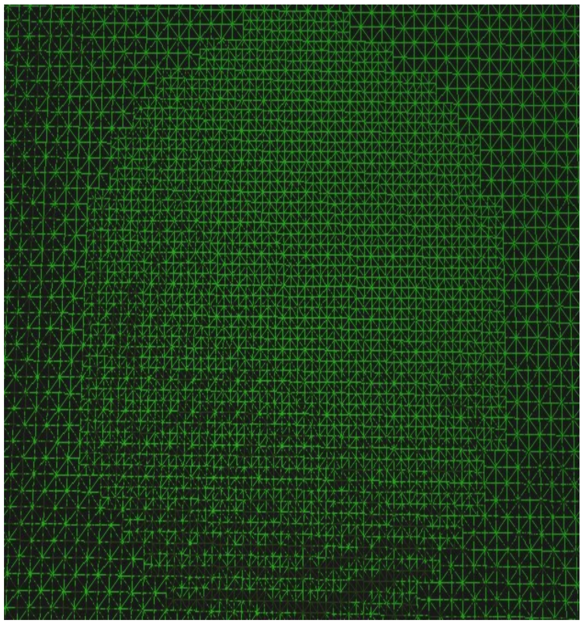


**Figure 10: Tessellation 1 to 2**

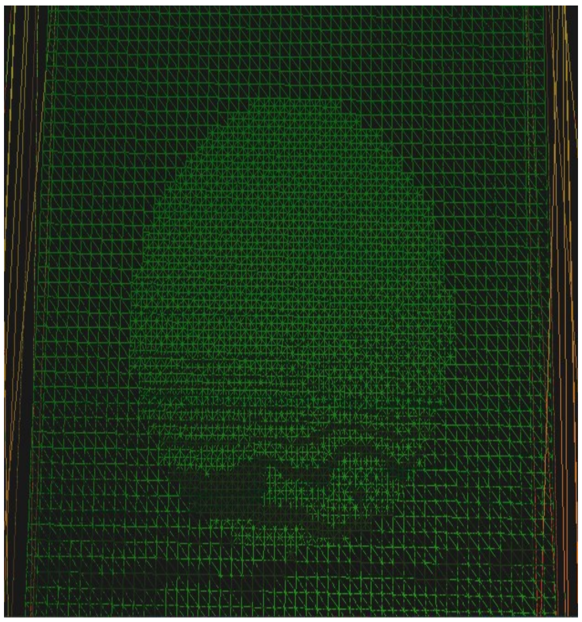
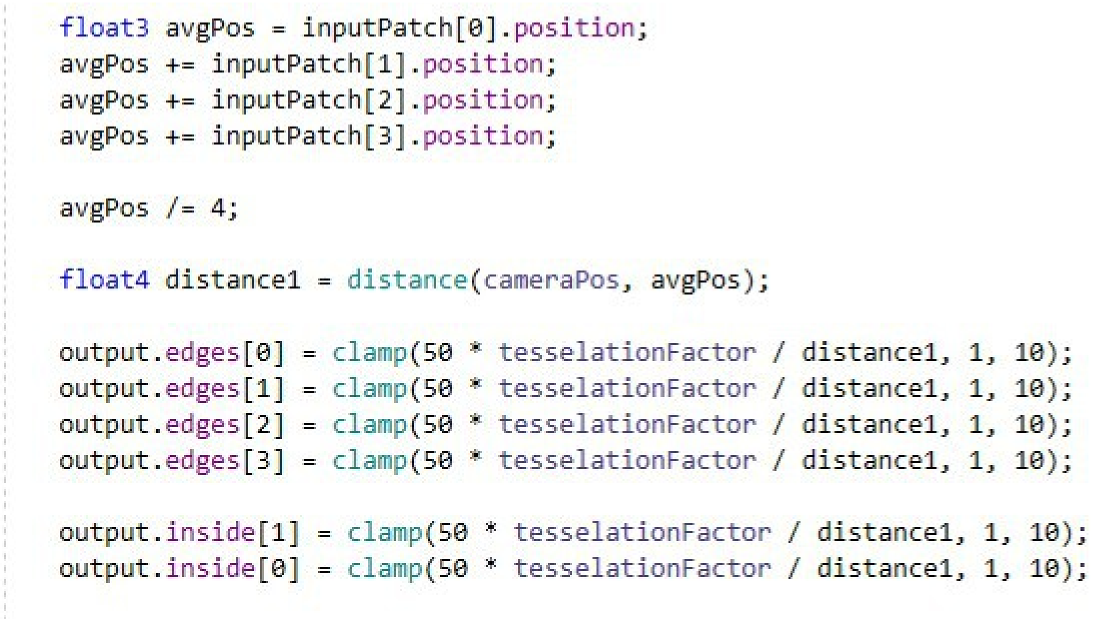


**Figure 11: Tessellation from 2 to 3**



```
float3 avgPos = inputPatch[0].position;
avgPos += inputPatch[1].position;
avgPos += inputPatch[2].position;
avgPos += inputPatch[3].position;

avgPos /= 4;

float4 distance1 = distance(cameraPos, avgPos);

output.edges[0] = clamp(50 * tesselationFactor / distance1, 1, 10);
output.edges[1] = clamp(50 * tesselationFactor / distance1, 1, 10);
output.edges[2] = clamp(50 * tesselationFactor / distance1, 1, 10);
output.edges[3] = clamp(50 * tesselationFactor / distance1, 1, 10);

output.inside[1] = clamp(50 * tesselationFactor / distance1, 1, 10);
output.inside[0] = clamp(50 * tesselationFactor / distance1, 1, 10);
```

**Figure 12: Dynamic tessellation calculation**

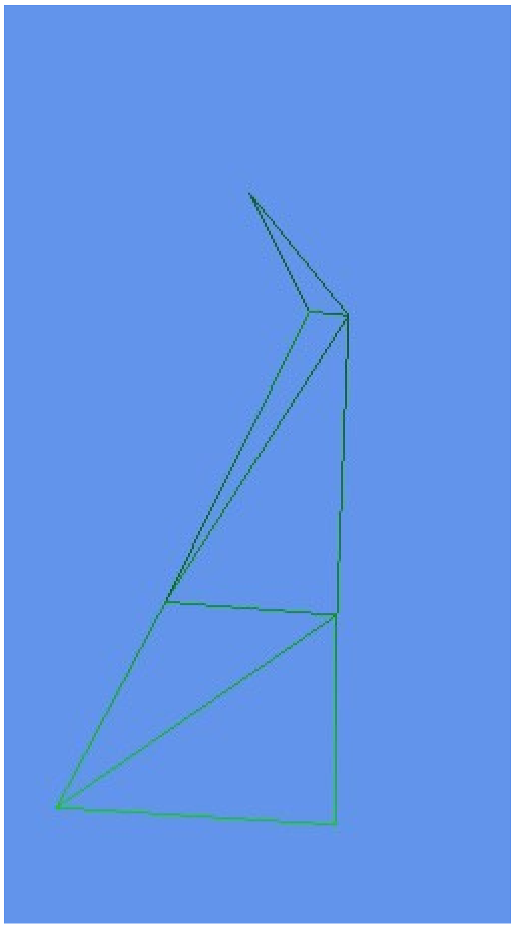
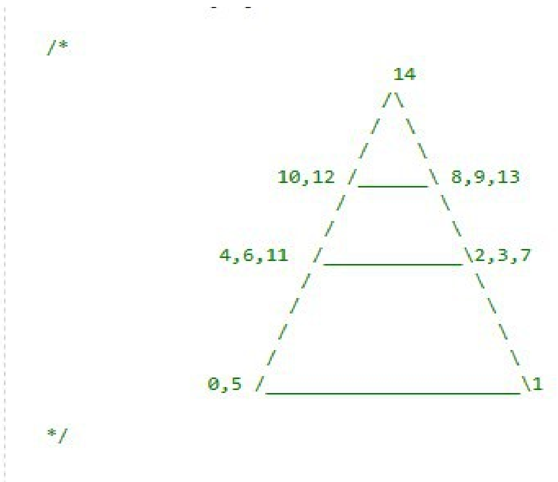
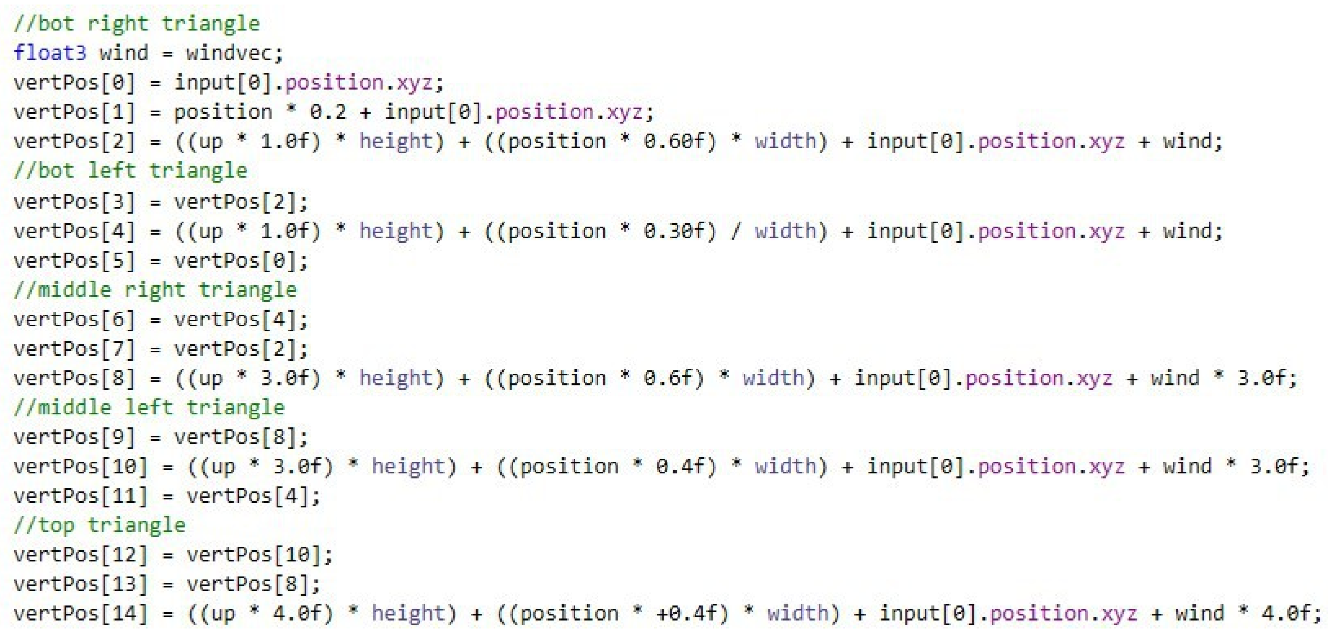


**Figure 13: Grass blade triangle composition**



```
/*
                         14
                         /\
                        /  \
                       /    \
             10,12 /______\ 8,9,13
                     /        \
                    /          \
         4,6,11  /____________\2,3,7
                  /              \
                 /                \
                /                  \
               /                    \
        0,5 /______________________\1

*/
```

**Figure 14: Visual aid for creating blade of grass**



```
//bot right triangle
float3 wind = windvec;
vertPos[0] = input[0].position.xyz;
vertPos[1] = position * 0.2 + input[0].position.xyz;
vertPos[2] = ((up * 1.0f) * height) + ((position * 0.60f) * width) + input[0].position.xyz + wind;
//bot left triangle
vertPos[3] = vertPos[2];
vertPos[4] = ((up * 1.0f) * height) + ((position * 0.30f) / width) + input[0].position.xyz + wind;
vertPos[5] = vertPos[0];
//middle right triangle
vertPos[6] = vertPos[4];
vertPos[7] = vertPos[2];
vertPos[8] = ((up * 3.0f) * height) + ((position * 0.6f) * width) + input[0].position.xyz + wind * 3.0f;
//middle left triangle
vertPos[9] = vertPos[8];
vertPos[10] = ((up * 3.0f) * height) + ((position * 0.4f) * width) + input[0].position.xyz + wind * 3.0f;
vertPos[11] = vertPos[4];
//top triangle
vertPos[12] = vertPos[10];
vertPos[13] = vertPos[8];
vertPos[14] = ((up * 4.0f) * height) + ((position * +0.4f) * width) + input[0].position.xyz + wind * 4.0f;
```

**Figure 15: Code for blade creation**

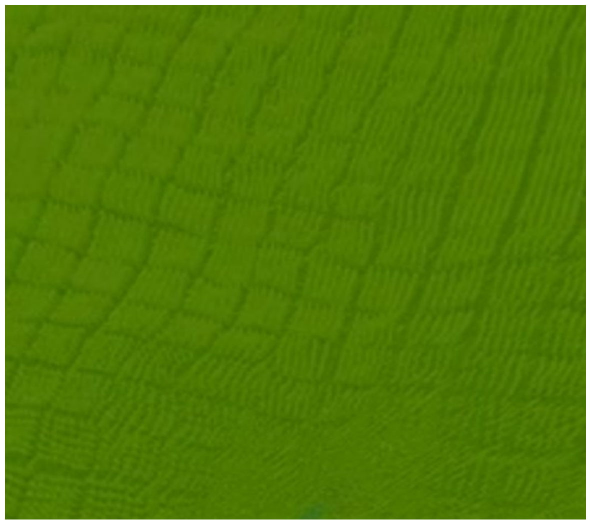


**Figure 16: Before noise map implementation**

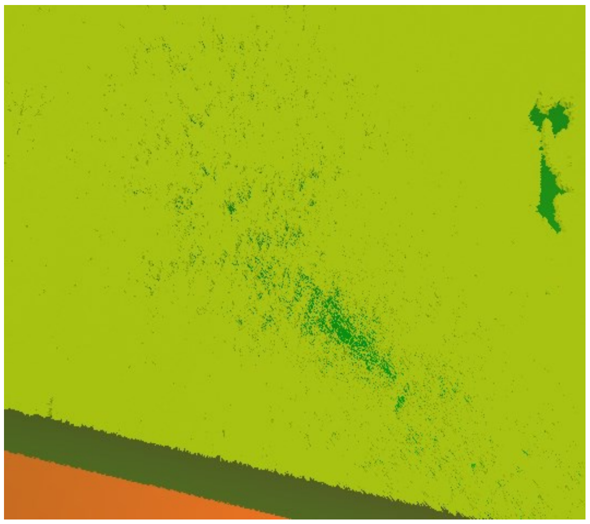
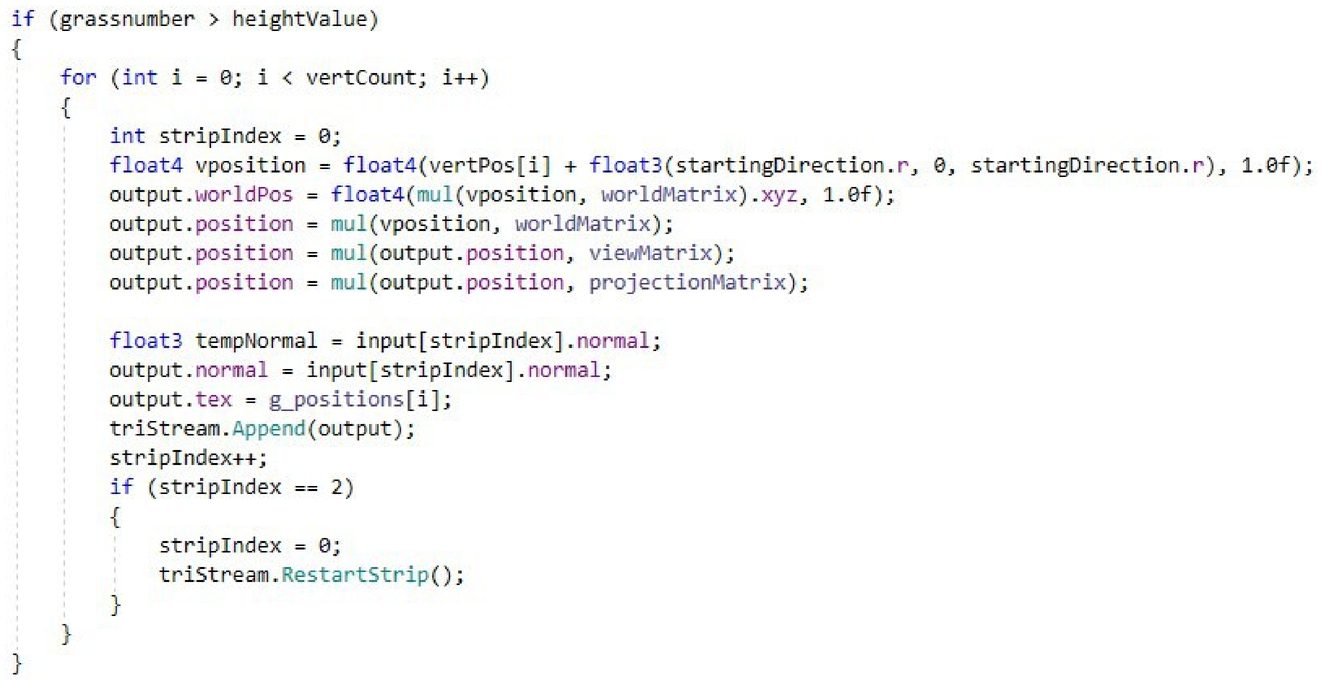


**Figure 17: After noise map implementation**



```
if (grassnumber > heightValue)
{
    for (int i = 0; i < vertCount; i++)
    {
        int stripIndex = 0;
        float4 vposition = float4(vertPos[i] + float3(startingDirection.r, 0, startingDirection.r), 1.0f);
        output.worldPos = float4(mul(vposition, worldMatrix).xyz, 1.0f);
        output.position = mul(vposition, worldMatrix);
        output.position = mul(output.position, viewMatrix);
        output.position = mul(output.position, projectionMatrix);

        float3 tempNormal = input[stripIndex].normal;
        output.normal = input[stripIndex].normal;
        output.tex = g_positions[i];
        triStream.Append(output);
        stripIndex++;
        if (stripIndex == 2)
        {
            stripIndex = 0;
            triStream.RestartStrip();
        }
    }
}
```

**Figure 18: Geometry generation**