Tessellation based Terrain Rendering

Andreas Kvalsvik Jakobsen

Master of Science in Computer Science
Submission date: June 2012
Supervisor: Theoharis Theoharis, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Modern Graphics Processing Units (GPU) exhibit a high degree of parallelism and over the years have grown to adopt an increasing number of techniques to speed-up photorealistic rendering. One such technique is tessellation, i.e. the recursive subdivision of object elements into finer or coarser parts with the aim of achieving the appropriate amount of detail.

The aim of this thesis is an adaptive tessellation algorithm for terrain rendering which can highlight important parts of the terrain while maintaining reasonable performance. This algorithm needs to be crack-free to avoid pixel faults between adjacent patches. A prototype was created in OSG, with GLSL shaders. The adaptive tessellation used three tessellation selection factors based on distance, a tessellation map and normals. An OSG program with five GLSL shaders (vertex, tessellation control, tessellation evaluation, geometry and fragment) was created for the tessellation of the terrain. The greatest advantage of tessellation is the reduced bandwidth between memory and GPU. Tessellation improves FPS, because it's faster to control vertices on the GPU than on the CPU.

Preface

This thesis is part of the master study at Norwegian University of Science and Technology. The problem description for this thesis is to make an adaptive tessellation algorithm for terrain to highlight important parts. The assignment was given by Theoharis Theoharis at IDI and Jo Skjermo from Statens Vegvesen. The time period was 20 weeks. I would like to thank my supervisors Theoharis Theoharis and Jo Skjermo for help, support and motivation. I will also like to thank my family and my girlfriend for supporting me through these weeks.

Contents

List of Figures

List of Tables

Listings

1 Introduction

In this chapter the motivation for this project will be presented together with the problem description and the structure of this thesis. The main goal for this thesis is to make a tessellated terrain for Vegvesenets car simulator. Vegvesenet use the simulator to simulate traffic and road projects, therefore the terrain need to be more accurate where the road meets the terrain. The solution was to make a texture that follows the road and highlights important places, to specify the level of tessellating. Vegvesenet needs terrain as close to the real environment as they can get on their simulator, because the different tests need to look real to the driver.

One of the largest problems in terrain rendering is performance. Today the terrain is static, which means that wherever you are, the terrain will still be the same. To speed up the FPS (Frames Per Second), the terrain is made less detailed away from the road. Tessellation controls the details based on factors that makes a good LOD (Level Of Detail). The factors made from specified needs, such as more tessellation, are needed close. Every factor is used in the tessellation formula to decide the final level of tessellation. The tessellation can change the terrain based on the position of the camera and how detailed the height map is.

The product of this master thesis is an OSG (OpenSceneGraph) program with GLSL (Graphics Library Shading Language) shaders that use adaptive tessellation on the terrain. The terrain program uses light, texture and one camera. The camera is needed to change the view matrix in a way that we can move over the terrain. This is just for debugging purposes because in the simulation the camera will be mounted on a car.

1.1 Problem description

The objective of this thesis is the adaptive tessellation of terrain data for a driving simulator at Vegvesen, in order to provide high-quality rendering of terrain data in real time. It will be based on a height map that defines the terrain, a 3D road network model and textures for making the terrain realistic. It will involve GPU tessellation and OSG technology.

1.2 Structure of the thesis

The second chapter is about background and related work. This is the chapter where the programming language is briefly presented and hardware is specified. The third chapter contains the related work, while the fourth chapter holds the method, the description of the vertex shaders and textures used in the terrain. Later on the results are discussed in chapter five. Last comes the conclusion and further work, in chapter six. In the appendix the shader code is provided.

2 Background

2.1 Developing languages

To understand OSG which was specified in the problem description, it's important to understand OpenGL, since OSG is based on OpenGL. OSG is just a higher level language. The main part of this chapter is about languages since it is a large part of the project to understand the interface.

2.1.1 OpenGL

OpenGL (Open Graphics Library) [1] is a language to control graphics hardware. It has commands to make primitives, matrix and all that we need to create 3dimensional objects. The objects are made from primitives like points, lines, triangles and patches. OpenGL added tessellation to version 4.0 [2]. This is done by using tessellation shaders and the new primitive patches. The number of vertices for GL_PATCHES can vary from 1 - 32. The only primitive that is supported by tessellation shaders is the primitive patches. The tessellation shaders consist of two programmable shaders, tessellation control and tessellation evaluation. Between them in the pipeline there is a fixed shader called tessellation primitive generator. On the internet there are a lot of OpenGL tutorials. The tutorials explain basic things like how to create a camera or add textures and lighting [3] [4].

2.1.2 OpenSceneGraph

A scene graph is a data structure, it contains different nodes that relate to each other [5] [6]. It is most common to have one parent and multiple child nodes. OSG was started by Don Burns when he was experimenting to make his hang gliding simulator available to affordable hardware. In the nineties, Burns met a fellow hang glider fan, Robert Osfield, who wanted to collaborate on the project. Osfield wanted to make a stand-alone OSG and took over the leadership. By 2006 there were over 1500 users on the email list. OSG is used in advanced visualization and simulation applications.

2.1.3 OpenGl Shading Language

GLSL (OpenGl Shading Language) [20] [21] is used to make shaders that control the OpenGL processing pipeline. The five processors that are programmable are: vertex, tessellation control, tessellation evaluation, geometry and fragment. The shader that controls the vertex processor is called vertex shader, tessellation control shader and so on. In the vertex shader we can change vertices, one at a time, often just passes through the vertices, or changes the position. Tessellation control shader is the shader that decides the level of tessellation. It is therefore an important shader when optimization is needed. The numbers of tessellation drastically reduces frame rate. The maximum number for tessellation is 64. That means that every edge is divided by 64. If the inner tessellation also is set to 64, each quad will be divided 64 by 64. The tessellation evaluation shader decides the position and attributes to the vertices generated by the tessellation primitive generator. The position of the vertices can be found by interpolation between the main vertices, by parametric patches or by height map. Geometry shader gets primitives in and can do displacement on vertices and add attributes. Fragment shader or pixel shader uses positions, colours, textures and calculates light to fill the fragment with right colour.

To interact with the shaders from the code we use uniform variables. It could be textures, matrix, floats and vectors. We need these if we have to change parameters during the run. Each shader has built-in-functions that are available only for them. The geometry shader uses for instance Emit Vertex and End primitive. This is used to send vertices and complete the primitive.

2.2 Hardware

To use hardware tessellation on GPU requires a graphic card which supports DirectX 11 or OpenGL 4.0. The figures in this assignment is made with MSI GeForce GTX 560Ti 2GB PhysX [25] and runs in 880MHz. This card supports DirectX 11and OpenGL 4.1, and was chosen based on price, performance and power consumption.

3 Related work

Here are similar approaches to terrain rendering presented.

3.1 Subdivision

Subdivision is to divide surfaces with a scheme. Catmull-Clark subdivision is one of the most famous subdivision schemes [26]. The rules to subdivide a quad are to split each face of the control mesh into four faces, by adding a vertex in the middle and one new vertex at each edge. The position can be calculated by using a weight mask. There are two main types of subdividing: interpolating and approximating. Interpolating means that the control points remain in the subdivided mesh. In the approximating approach the new points and the old points are moved in every step. Catmull-Clark is an approximating subdivision [27].

3.2 Tessellation

One of the biggest feature to DirectX 11 was tessellation [28] [29]. Tessellation is to divide a polygon into smaller pieces. While subdivision needs to calculate one step at a time, tessellation calculates the entire dividing in one step. The most popular way to use tessellation is to use a displacement map for changing the height of the tessellated polygon. LOD becomes better with tessellation since we can vary the level smoothly while the program runs.

Boesch [30] made a terrain by using Python and GLSL shaders. He used one image for normals and height. The red, green and blue components vas the normal and the alfa value stored the height. One of his optimizations was to set the patches out of view to zero. Rideout have two articles about tessellation [31] [32], both triangles and quads, and some good explanations to the most important parameters that needs to be set for tessellation.

3.2.1 Adaptive tessellation

Losasso and Hoppe [33] use a clipmap to decide the tessellation level. This has similarities with texture mapping. The geometry clipmap caches the terrain in a set of nested regular grids centred about where the viewer is. The mask has different levels of tessellation at the power of two, and moves with the viewer. This is a LOD, based on viewer distance.

Livny et.al [34] generates a sample grid. The resolution to the grid is determined by the hardware to the system, and is set to the resolution that the system manages to run within the desired FPS. The grid remains fixed, usually for the entire session. The persistent grid is mapped onto the z plane, by sending rays from the viewport through the grid, on to the z plane which is the terrain base plane.

3.2.2 Tessellating with displacement map

Displacement mapping is used to apply geometric detail to a simpler surface base. This technique makes it easier to increase the complexity of the geometry. Moule and McCool [35] use displacement map and adaptive tessellating. They find the edges and use a user-defined threshold to set the tessellation factors. Displacement maps can also be used to make 3d polygons, Jan Kautz and Hans-Peter Seidel [36]render polygon from displacement maps by slicing through the volume. Wang et.al. [37] presents an approach to rendering the displacement mapped surface based on the complexity of the displacement map. They use a minimal set of triangles which reduces memory usage and computational costs. The triangle density is greatest where the details are. The cracks made from the connected low and high vertices, are avoided by forcing the triangulation to include the edge. The edge is found by looking at neighbour points for similar height.

3.2.3 Tessellating with Bézier patches

Bézier patches [38] are used for tessellating when a curved patch needs positions for the new vertices. The control points (initially vertices) decide how the patch turns out. Dyken et.al. [39] uses Bézier patches when they adaptively tessellate based on silhouette. Their method tests the normals from the triangle patches with the viewpoint position. The silhouette is found, if one point is front-facing while the other is back-facing. The tessellated geometry changes when the viewpoint is moved.

3.3 Marching Cubes

The voxel-based rendering system Marching Cubes (MC) [40] [41], can be used for making a terrain from a three dimensional image. MC looks at one cube at a time, which has a value in every corner, and use a lookup table to find the number of primitives to use. The lookup table is made from 14 different cubes that are rotated to get all possible situations. If we count the 15th cube, which have zero triangles, this makes 256 cases from 0 to 255. The vertices that lies on the edges are placed by interpolations of the two corners. If one corner has value -0.5 and the other have 0.5 the edge vertex is placed right in the middle on the zero value. MC is often used in water simulations where the voxels can make several clusters.

4 Proposed Tessellation Method

Five shaders are used to tessellate the terrain: the vertex shader, the tessellation control shader, the tessellation evaluation control shader, the geometry shader and the fragment shader. Each shader is linked to the OSG program, and sends information with uniform variables (see table 1 and 2). Every shader will be explained in this section. Figure 1 shows how the shaders are linked together.

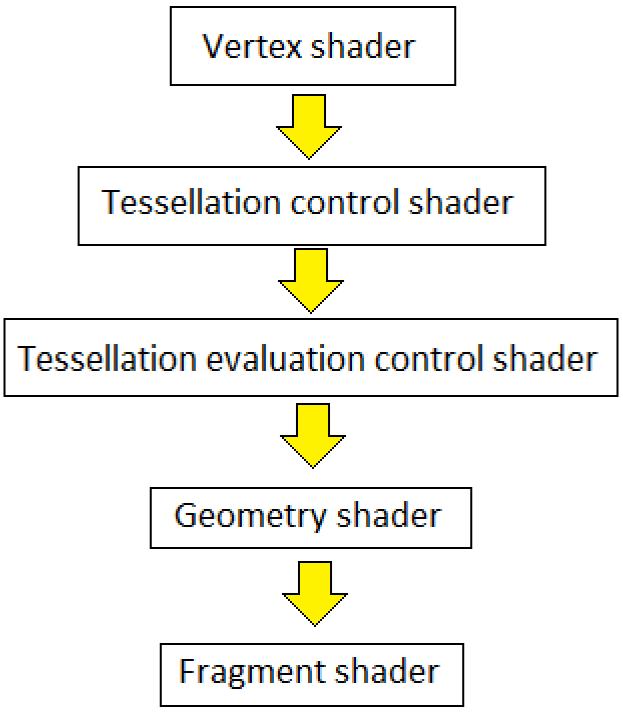
Vertex shader
Tessellation control shader
Tessellation evaluation control shader
Geometry shader
Fragment shader

Figure 1: Shader connections

4.1 Vertex shader

When the patches arrive the vertex shader, all the vertices are in one plane, the xy plane. Every vertex has the z value zero. Usually the height is also sent to the vertex shader, but in this case there are two uniform variables, scaleXY and

Table 1: Uniform variable table, part 1

Uniform name	Variable type	Description
heightMap	sampler2D	Texture with terrain height values
normalMap	sampler2D	Texture with terrain normal
tessellationMap	sampler2D	Texture deciding tessellation level
diffuse	sampler2D	Main texture
canyon_rock01	sampler2D	Rock texture
canyon_rock02	sampler2D	Rock texture
dry_grass09	sampler2D	Grass texture
dry_grass12	sampler2D	Grass texture
Romanian_sand05	sampler2D	Sand texture
VertRock_0_90_mask	sampler2D	Texture mask for rock textures
VertDetail_0_90_mask	sampler2D	Texture mask for grass and sand textures
Grass_mask	sampler2D	Texture mask for grass
Flow_mask	sampler2D	Texture mask for sand
AmbientOcc	sampler2D	Texture mask for ambient light

Table 2: Uniform variable table, part 2

Uniform name	Variable type	Description
lightPosition	Vec3	Position for light
ambientMaterial	Vec3	Ambient light properties
diffuseMaterial	Vec3	Diffuse light properties
specularMaterial	Vec3	Specular light properties
shininess	float	Shininess light properties
rotationMatrix	Matrix	Rotate texture coordinates
mvp	Matrix	Projection * View * Model
normalMatrix	Matrix3	The left upper 3 by 3 matrix from Model matrix
scaleXY	float	Scale factor for x and y axes
scaleZ	float	Scale factor for z axis
cameraPosition	Vec3	Position of the camera
userDecidedLevel	float	Controls tessellation level

scaleZ. These two variables are used to scale the terrain in both xy plane and in z direction. Since this can change during the run of the program, there is no need to pass height to the vertex processor when we don't know the scale yet. The texture coordinate is set by dividing the vertex position by scaleXY. If scaleXY is 10, the height map will cover 10 times longer in both x and y direction. ScaleZ controls the height. The height map contains values from 0 to 1 and is multiplied with scaleZ, so if scaleZ is 250 the z value goes from 0 to 250. The height map is a black and white image, which means that all the values red, green and blue have the same value at the same pixel. In this case the red r component is used. When the height is found, the complete position is being sent to the tessellation control shader.

4.2 Tessellation control shader

This is the most important shader for tessellating. The tessellation control shader decides how the patches will be tessellated. To get the best quality with an acceptable FPS, we need a formula that adaptively decides the tessellation level for the terrain. The formula was based on distance, tessellation map and normals from the patch vertices.

4.2.1 Tessellation based on distance

To imagine how the distance is affecting the tessellation factor, think of a bounding box or a bounding sphere in this case. When the middle of a quad edge comes in contact with, or is inside the sphere, it will be tessellated. The amount of tessellation is then calculated based on the distance to the camera position. Figure 2 is a patch from the tessellation control shader. The blue dots are the vertices 0,1,2,3 clockwise. And the ab, dc, ad and bc are four vectors that complete the patch. It is important that the parallel vectors have the same direction, because

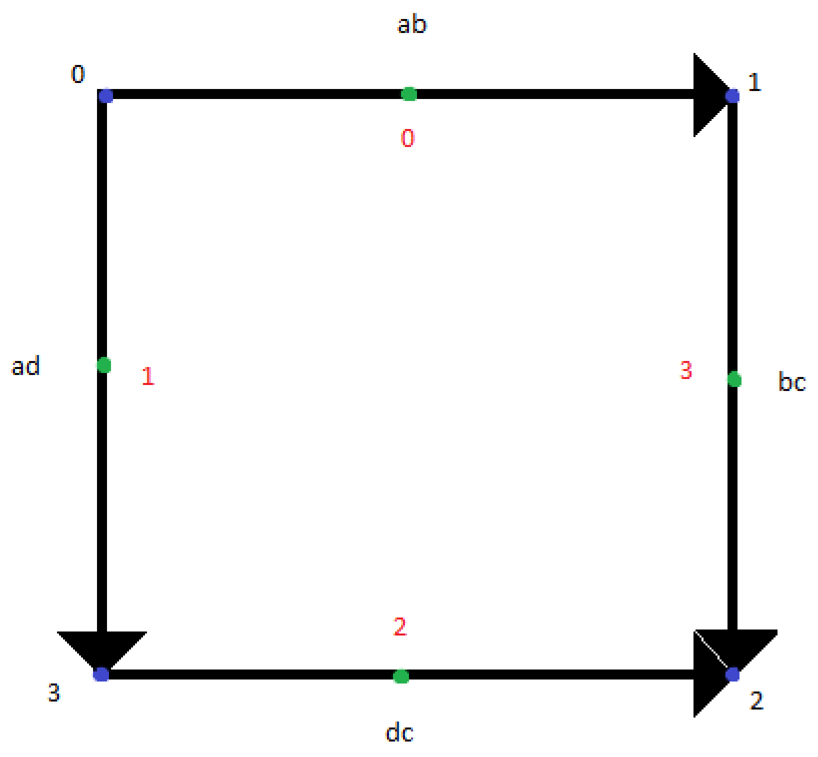

Figure 2: Vectors on a patch

the edge combines the two patches. This means that the calculation happens two times for every edge, and needs to be accurate to get the same tessellating factor. The green dots are where the distance is checked. The position is calculated by adding a half vector to the starting vertex position. For instance, the position for vertex 0 added a half ab vector, as we can see in equation (1). The green dot on the ad vector is abMidPos. vPosition[0] holds the position of the vertex 0 and vPosition[1] for vertex 1 (vertices is the blue dots in figure 2).

$$abMidPos = vPosition[0].xyz + \frac{vPosition[1].xyz - vPosition[0].xyz}{2} \tag{1}$$

When we have the distance to the centre point of every edge, the distance is calculated from the point to the camera position. The tessellation factor is then calculated based on the distances and in this case set to the gl_TessLevelOuter[0], bc to the gl_TessLevelOuter[3] and so on. The red numbers in figure 2 show where the vectors corresponds with the gl_TessLevelOuter. The vertices are numbered clockvice and the tessellation level is numbered counterclockwise. The inner tessellation is set to be the average of the four edges.

4.2.2 Tessellation based on tessellation map

Vegvesenet use the simulator to simulate traffic and road projects, therefore the terrain need to be more accurate where the road meets the terrain. The solution was to make a texture that follows the road and highlights important places, to specify the level of tessellating. The idea of a tessellation map sprung out of the need for more detailed tessellation, close and on the roads. A grey value scaled from 0 to 255 is assigned on a 2d image, based on how important the point on the map is. Since we have to make the tessellation map ourselves, it does not only limit us to draw roads, but other important areas such as towns. In the control shader the point tested against is the middle of the edge. The middle is found by calculating each vector and dividing it by 2, then add it to the corresponding corner, as in the distance tessellation factor. It is important to calculate the vectors in the same direction on the opposite side. This is crucial because each edge will be calculated 2 times since the patches share the edge between them. Both edges need to be tessellated the same amount to prevent cracks between them. The point or points could vary as long as they are calculated the same way. The parallel vectors must also have the same number of points. In this tessellation control shader, one point is used for every edge because the size of the road is larger than the size of the patch. If the road is larger than the patch, the middle of the patch edge will naturally be in the road. If the road is smaller than the patch then you need to add more check points around the edge to prevent the road slipping undetected between them.

4.2.3 Tessellation based on normals

Equally to the previous two sections, each edge is calculated twice, where the two quads meet. The algorithm makes four new points for each edge, see figure 3. The red dots are the existing vertices and the blue dots are the new points, where we also check the normals. From figure 4 we can see that the four dots make two squares, one on each side of the vector, in this case the bc vector. Listing 1 shows how the translation vectors are made. The distance between the vertices in current patch is used to find the position to the vertices of the neighbour patches.

```
// Vectors to make parallel patch edge vectors to compare
    normals.
vec2 left = vec2(vPosition[0].xy -vPosition[1].xy);
vec2 right = vec2(vPosition[1].xy-vPosition[0].xy);
vec2 up = vec2(vPosition[1].xy-vPosition[2].xy);
vec2 down = vec2( vPosition[2].xy-vPosition[1].xy);
```

Listing 1: Translation vectors from tessellation control shader

This is done for all four edges. The green lines in figure 4 shows which point vertex 2 is tested with. The red lines show the test points for vertex 1. This is put in a struct-array that contains the normal and distance to the camera. The distance is used, if we want to use tessellation in a specified radius. In the figure, the diagonal lines show the matching corners. The array is sent to normalTessFactor and the angle between the normals from the diagonal vertices is calculated.

4.2.4 Tessellation control formula

The three tessellation factors described above are put together by formula (2). Every factor has a weight to make it easy to decide how the tessellation is spread. The constantTessellationWeight is for the patches that do not qualify in the other tessellation factors. In this way we avoid that some patches are divided by 64 and some others set to 1, at the same time.

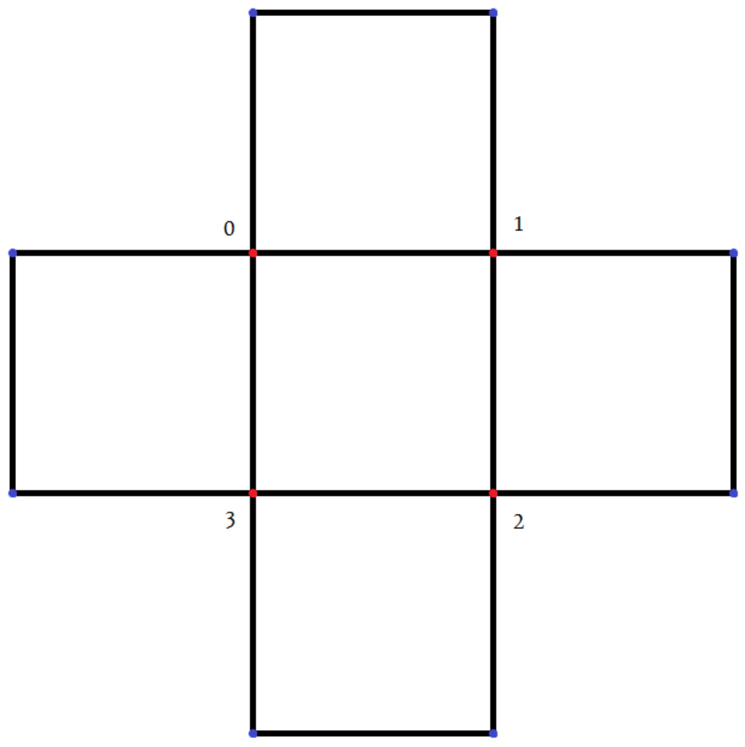

Figure 3: Points used in normal tessellation factor

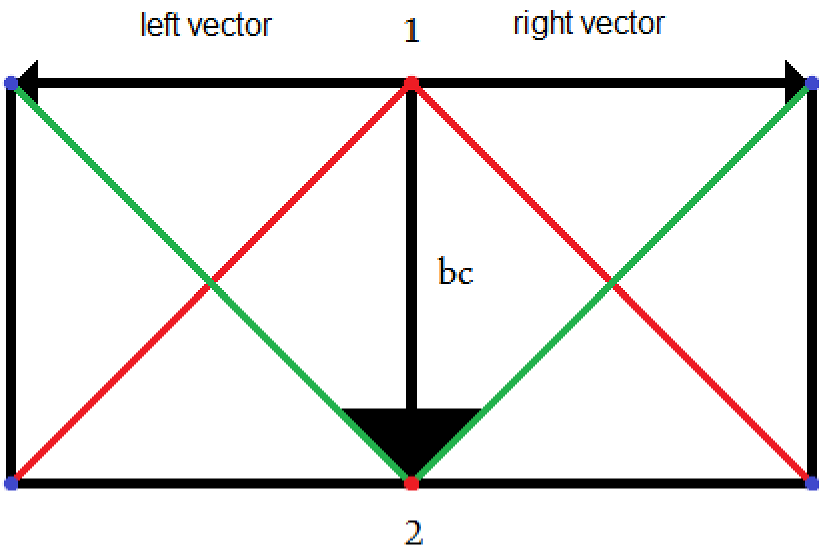

Figure 4: Points used in normal tessellation factor, for one edge

- cTW = constantTessellationWeight (user decided weight, set in shader)
- nT = normalTessellation (from 0 to 1 and set by tessellation based on the normals)
- nTW = normalTessellationWeight (user decided weight, set in shader)
- tT = textureTessellation (from 0 to 1 and set by tessellation based on tessellation map)
- tTW = textureTessellationWeight (user decided weight, set in shader)
- DT = distanceTessellation (from 0 to 1 and set by tessellation based on distance)
- dTW = distanceTessellationWeight (user decided weight, set in shader)
- userDecidedLevel = (from 0 to 63 changed by keyboard, while the program runs)

$$TessLevel = 1 + \frac{cTW + nT * nTW + tT * tTW + DT * dTW}{cTW + nTW + tTW + dTW} * userDecidedLevel \tag{2}$$

4.3 Tessellation evaluation shader

This shader can use fractional or integer for tessellation. When we choose integer, the patches will be divided by 1, 2, 3, 4 and so on. This works when the tessellation is constant, but if the tessellation factor is based on camera position, the patches will jump between stages. This will look strange when the camera glides over the terrain.

As we can see from figure 5, two vectors are calculated: the ad vector and the bc vector. The gl_TessCoord.x was used on both of the vectors to get two points.

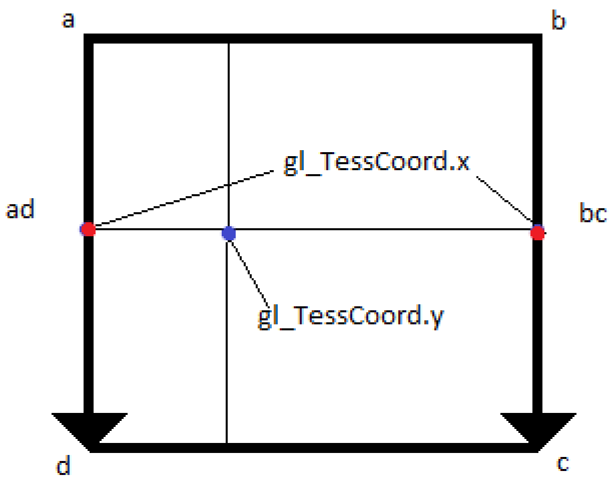

Figure 5: Vectors used in tessellation evaluation shader

If the gl_TessCord.x (the two red dots in figure 5) was 0.5, the new tessellated points are exactly in the middle of the vectors. Then gl_TessCord.y (the blue dot in figure 5) is used to find one point between the two points we got from ab and bc vectors. This position is divided by scaleZ to create the texture coordinate. Then the texture coordinate is used to find the height from the height map. The height is multiplied by scaleZ to find the real height. Vertex position is then multiplied by the model view projection (mvp) matrix and sent to the geometry shader.

4.4 Geometry shader

The light for the terrain was a bit of a challenge, since we can solve it in many ways. The first test was to calculate one normal for all three vertices in one triangle, and send them to the fragment shader. Figure 6 is an example of this.

This looks nice when the terrain was tessellated maximum (64), because each triangle was as small as a pixel. When the terrain had no tessellation, it was easy

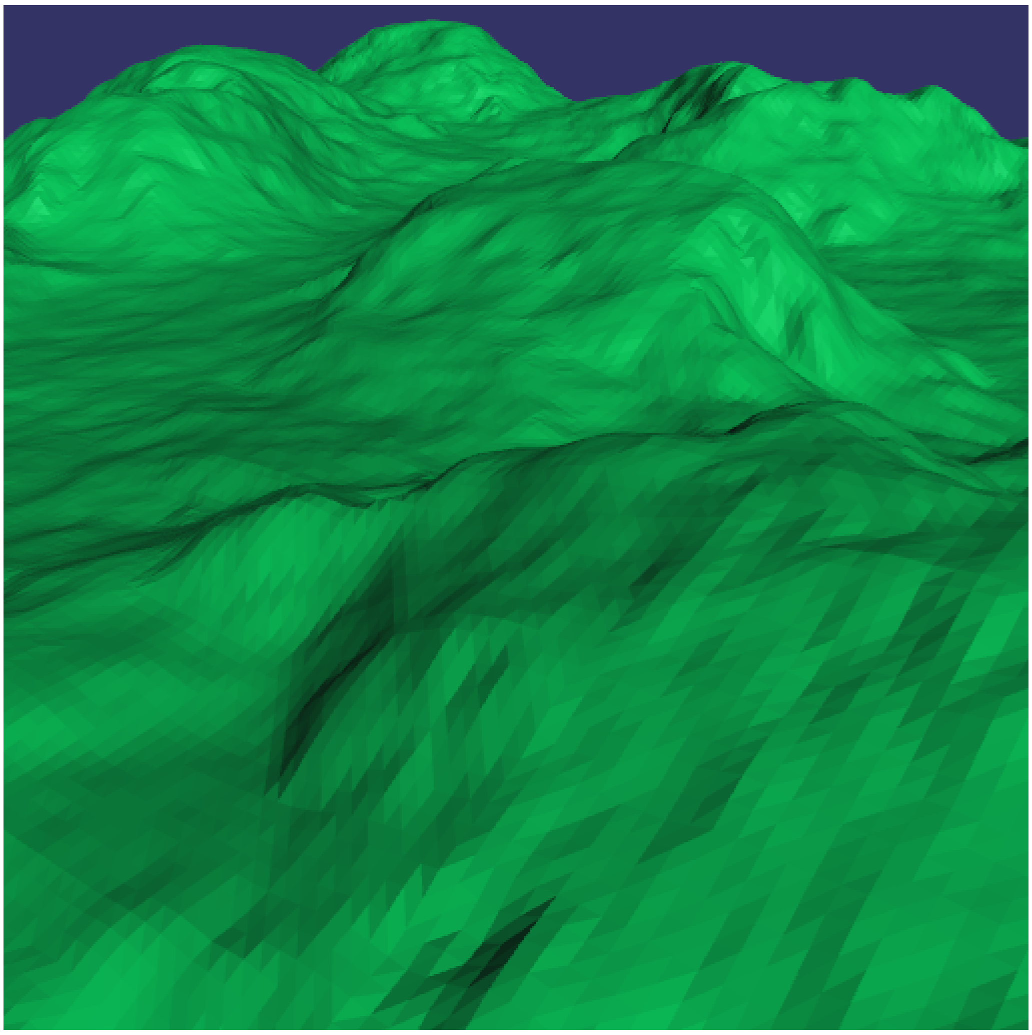

Figure 6: Light calculated based on one normal for each triangle

to spot the different colours on the triangles. This can make a cool effect and it's easy to see the structure, but this is inaccurate for a terrain, unless the tessellation level is high. Since the tessellated vertices are generated in the control shader, a normal map is needed in the geometry or pixel shader.

The next test was to calculate a normal map from the height map image. It was sent to the fragment shader and got the normal for every pixel. This looks good, but the problem with this approach was that we got the shadow-side for peaks, that did not exist in the current tessellation level. When the terrain tessellation level was high, the results were like the last test, good. The shadow-side got a peak to lie behind. The chosen method uses the geometry shader to get in the normals, one for every vertex. The light is then interpolating between these vertices in the fragment shader. This will make the dark side of the peak less visible, if the neighbour vertices lie by the side of the peak. When the normal is read in the three components of the vector r g b, it varies from zero to one, because the image only has positive values. The function NormalFromTexture takes in the colour value from the normal map and subtract 0.5, since the normal map zero starts at 0,5. Then we get the values from - 0,5 to 0,5. To the end the vector is normalized. The normal, position and texture coordinates are sent to the fragment shader. This can be done in a for-loop that runs three times, one for every vertex. This made the program drop by several FPS, in one case from 216 FPS down to 210 FPS. On the low FPS this made little difference to the FPS, it dropped from 58 to 57.

4.5 Multi texture Fragment shader

Light and texture is the main job for the fragment shader, also known as pixel shader. This shader takes in six textures. Five of these are used both horizontally and vertically. The rock texture, horizontally and vertically is mixed by using the VertRock_0_90_mask. This mask follows the terrain in a way that we can't see that the texture is repeated. To make it even better, there are two rock textures that lie on top of each other, with a different scale. This is done by multiplying the texture coordinate for each texture by different numbers. The two grass textures are also used horizontally and vertically, but these have a grass mask that tells where it should be grass. To the end the light is multiplied by the texture to complete the fragment.

4.6 Single texture Fragment shader

This is a simplified version of the multi texture fragment shader that calculates the light and uses one texture. The result of this is that FPS increases drastically when using single texture.

4.7 Terrain

The terrain is built from at least four images: the texture, height map, tessellation map and the normal map. Four images were made to make a terrain over an island, called Sekken. The texture and the height-values were found on the website Norway in images [49].

4.7.1 Texture

When single textures are used on a terrain, there are two options: one is to have a small image repeated several times on the terrain, this makes an unwanted pattern. The other is to use one large image that covers the whole terrain. This image can either use real terrain images like figure 7 or make it from multiple other textures in the same way as the multi texture fragment shader.

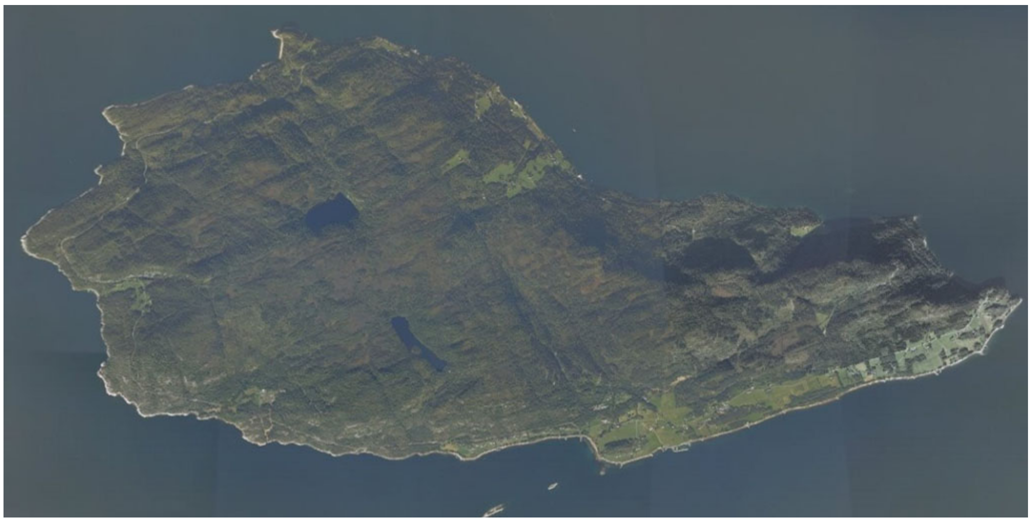

Figure 7: Texture of Sekken

4.7.2 Height map

The terrain starts as a plane, see figure 8. This is an image from an early stage when the patches were just put together. This height map was made by filling each elevation curve with a gray shade that fades to white on the highest values. It is important to scale the heights of the terrain, so that the whole spectre is used from 0 to 1, or 0 to 255. This is to make it more accurate. We have to remember to set the correct height scale in the code. If not, the terrain will have wrong proportions.

4.7.3 Tessellation map

Tessellation map is created manually, see figure 10. A good start is to draw the main roads in white, then the smaller roads in gray and keep the uninteresting areas in black. The image-values varies from 0 – 255 and will affect the patches according to these. White (255) will have most effect and black (0) will have no tessellation.

4.7.4 Normal map

Calculations of the normal map image are used in both the fragment shader and control shader. We take in the height map image to get the right size for the height map. Then the four position vectors are declared. A for-loop goes through every pixel from the height map and calculates the four vectors (one for each neighbour pixel). The for-loop sets the x and y component while the z component is read from the height map by using x and y. Vectors to neighbour pixels are calculated. Then 2 by 2 vectors are crossed. These have to be in 90 degrees of each other. This makes four normals for the current pixel, these are summed together and normalized to make the final normal for the pixel. The normal is given the colour value. Negative goes from 0 to 127, positive goes from 127 to 255. This is stored

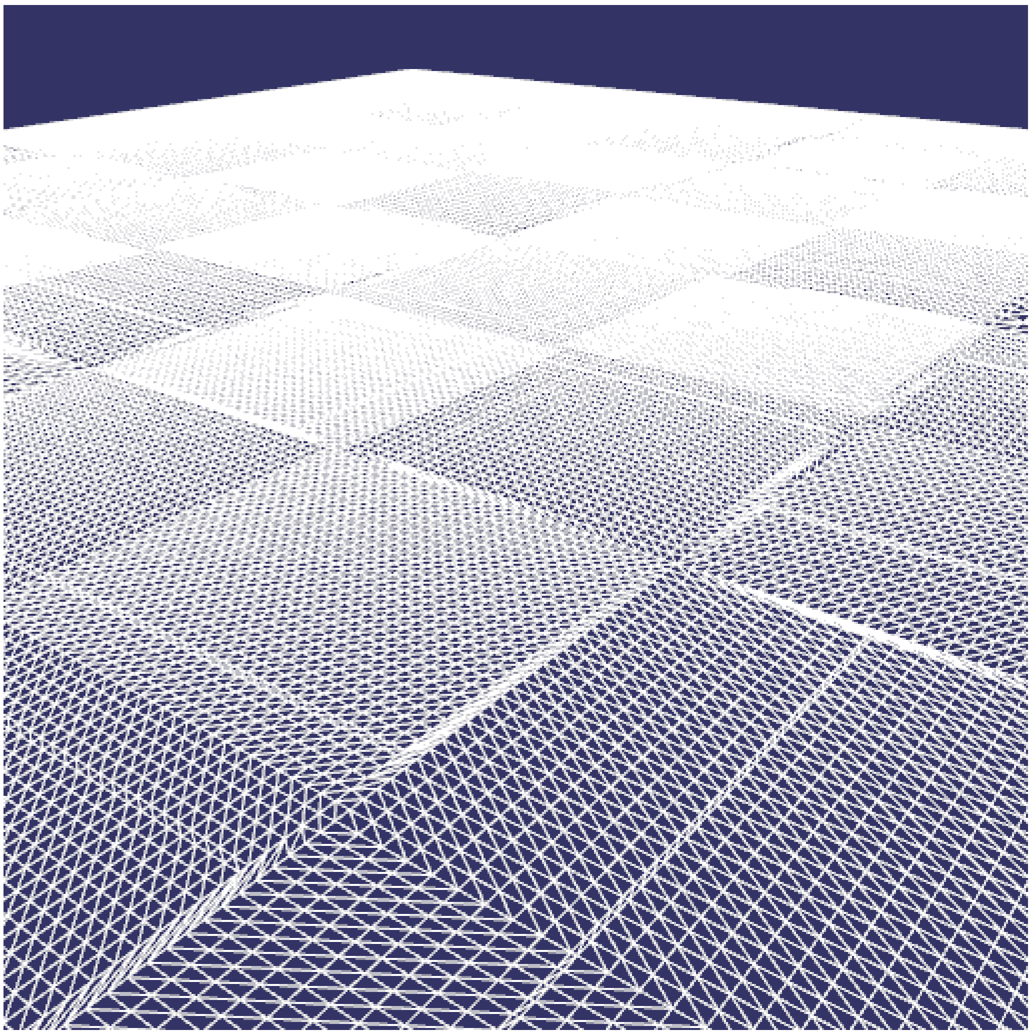

Figure 8: Plane of patches

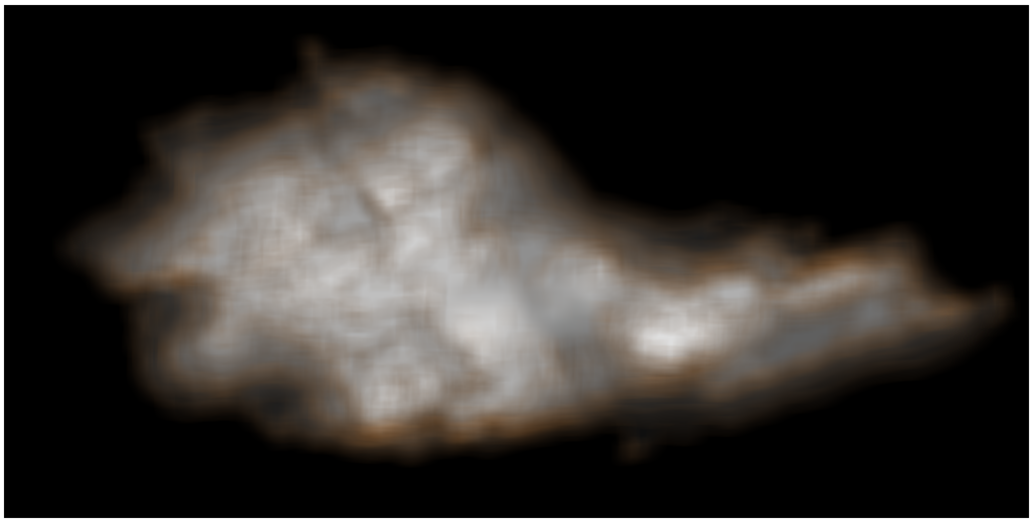
Figure 9: heightMap

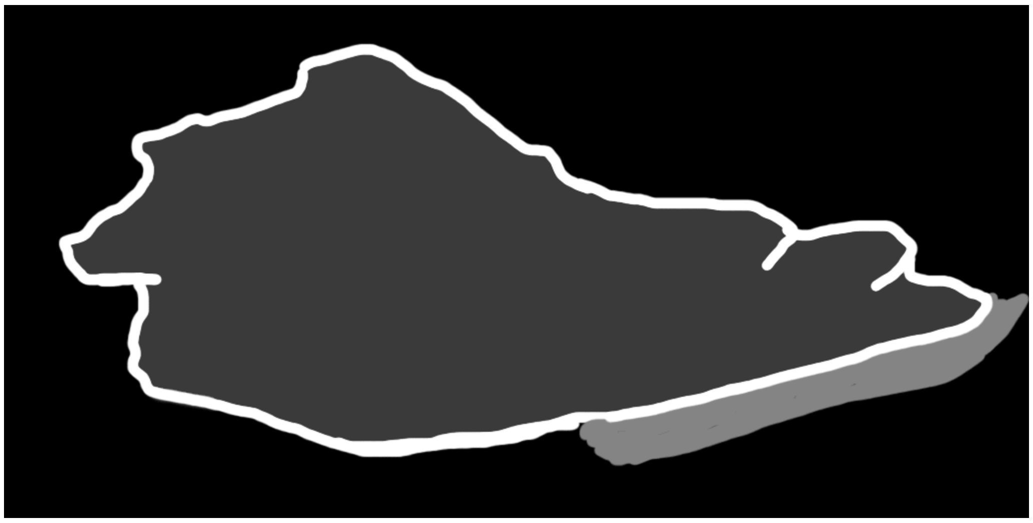
Figure 10: tessellationMap

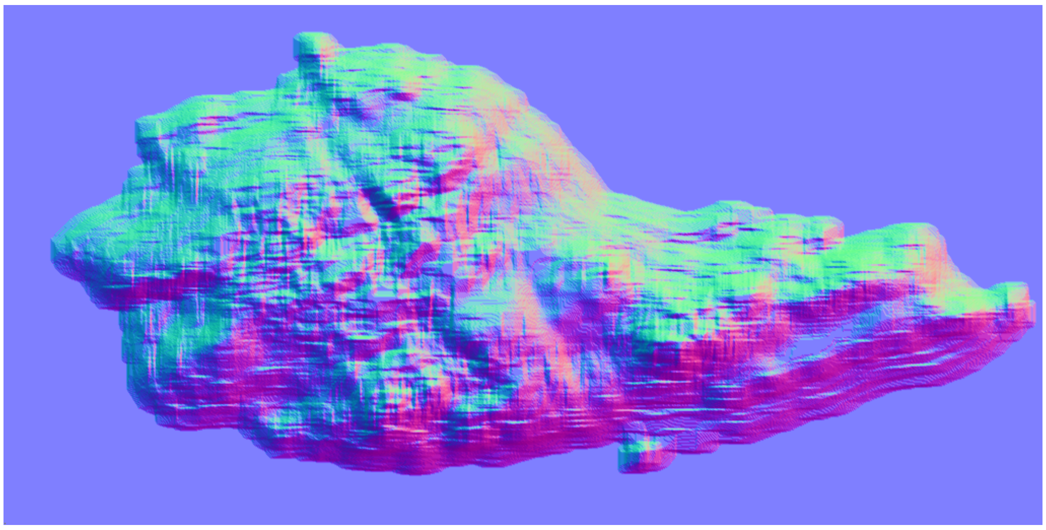

Figure 11: normalMap

in the normal map image. One example for a height map is viewed in figure 11.

4.8 Camera and input

The camera was the main reason to have an input handler. The key description is listed in table 3. The camera creates the view matrix with help from makeLookAt in OSG. The makeLookAt takes three arguments: camera position, target position and up vector. The up vector is set to (0,0,1) to specify that z is the up axis. The calculations of the target position and camera position are viewed in code listing 2. To visualize how this is done, see figure 12. The blue dot is the camera position and will always stay in the middle of the sphere. Target position is the green dot, and will move according to specified theta and phi. The length from camera position to camera position will always be one, therefore the target position is in the sphere surface at all time.

```
// View matrix.
// Calculate look-at-vector based on
// the two angles set by the input handler.
targetVector.x() = sin((theta * 3.14 / 180));
targetVector.y() = cos((theta * 3.14 / 180));
targetVector.z() = tan((phi * 3.14 / 180));
targetVector.normalize();

// Calculate translation.
cameraPosition += targetVector * speed;
speed = 0;// Set speed to zero again.

Vec3 targetPosition = cameraPosition + targetVector;
positionUniform->set(cameraPosition);// Set uniform.

Vec3 upVector = Vec3(0,0,1);
Matrix view;
view.makeLookAt(cameraPosition, targetPosition , upVector);
```

Listing 2: View matrix code from OSG

4.9 Light

Per pixel lighting means that each pixel has its own normal, when calculating the light. In this program the best way to calculate light is by using one normal for every vertex. The terrain changes in different tessellation levels. With a correct normal for every vertex, the light is calculated by interpolating between them, see listing 3. The fragment shaders light calculation is based on a shader by Rideout [32]. The maximum light occur, when the normal and light vector are in

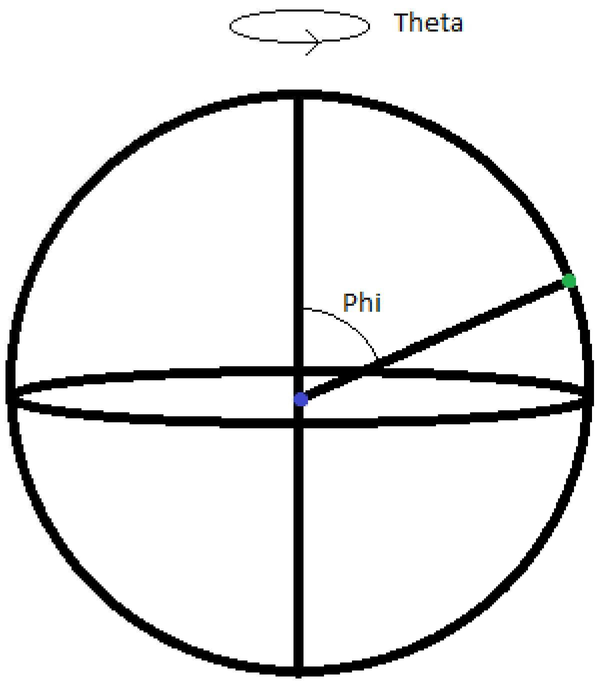

Figure 12: Camera vector

Table 3: Key description

Key	Description
E	Lifts camera in z direction
Q	Lower camera in z direction
A	Rotate camera left
D	Rotate camera right
W	Move camera forward in view direction
S	Move camera backward in view direction
R	Rotate camera up
F	Rotate camera down
T	Add one to userDecidedLevel
G	Subtract one from userDecidedLevel
X	Set render mode to wireframe
Z	Set render mode to fill
O	Add one to scaleZ
L	Subtract one from scaleZ
I	Add one to scaleXY
K	Subtract one from scaleXY

```
// Light calculation
vec3 h = normalize(lightPosition + vec3(0, 0, 1));
float df = abs(dot(gNormal, lightPosition));
float sf = abs(dot(gNormal, normalize(lightPosition + vec3
    (0, 0, 1))));
sf = pow(sf, shininess);

// Calculate the light based on the normal and light
    position.
 vec3 light = ambientMaterial + df * diffuseMaterial + sf *
      specularMaterial;
```

Listing 3: Light calculation from fragment shader

5 Results

This chapter shows the result of the tessellation program. Every figure shows different tessellation modes viewed in both fill mode and line mode (wireframe). This is because we need to render it as wireframe to get all the details of how the patches are divided and tessellated together. But it's necessary to show how the terrain is in fill mode to know how the terrain is adapting to the tessellation modes. Also to reveal that the terrain is rendered smooth, with no gaps between the patches. It is important to know the difference between triangles and patches. Patches are what you can see in figure 13. Two triangles make the quadratic patch. This patch can be divided in 64 * 64, and still being one patch, but it is 64 * 64 * 2 triangles.

5.1 Multi textures

In this section the figures are made by multiple textures and texture masks. The data was provided by Jo Skjermo. Table 4 shows the FPS for the figures from this section. Subsection shows different tessellation factors by themselves. First, we compare tessellation level 1 with 64, and then each factor shows how they affect the terrain. To the end the final results are viewed.

5.1.1 Image tessellation 1 vs 64

In figure 13 we can see the terrain passed through the control shader with tessellation level one. This means that no extra vertices are made. This is exactly how the vertices left the vertex shader.

Figure 14, shows the terrain when tessellation level is set to 64, which is the maximum level. In the back it seems that the terrain is filled. It means that the size of each triangle is close to the size of one pixel.

Table 4: FPS table multi textures

Tessellating factor	Tessellation level	Line or Fill	FPS	Screen resolution
Constant	1	F	312	1280 x 800
Constant	1	L	485	1280 x 800
Constant	64	F	23	1280 x 800
Constant	64	L	108	1280 x 800
Normal	2	F	181	1280 x 800
Normal	2	L	374	1280 x 800
Distance	4	F	257	1280 x 800
Distance	4	L	422	1280 x 800
Tessellation Map	4	F	407	1280 x 800
Tessellation Map	4	L	381	1280 x 800
Tessellation Map	18	F	373	1280 x 800
Tessellation Map	18	L	219	1280 x 800
All Factors	10	F	215	1280 x 800
All Factors	10	L	161	1280 x 800

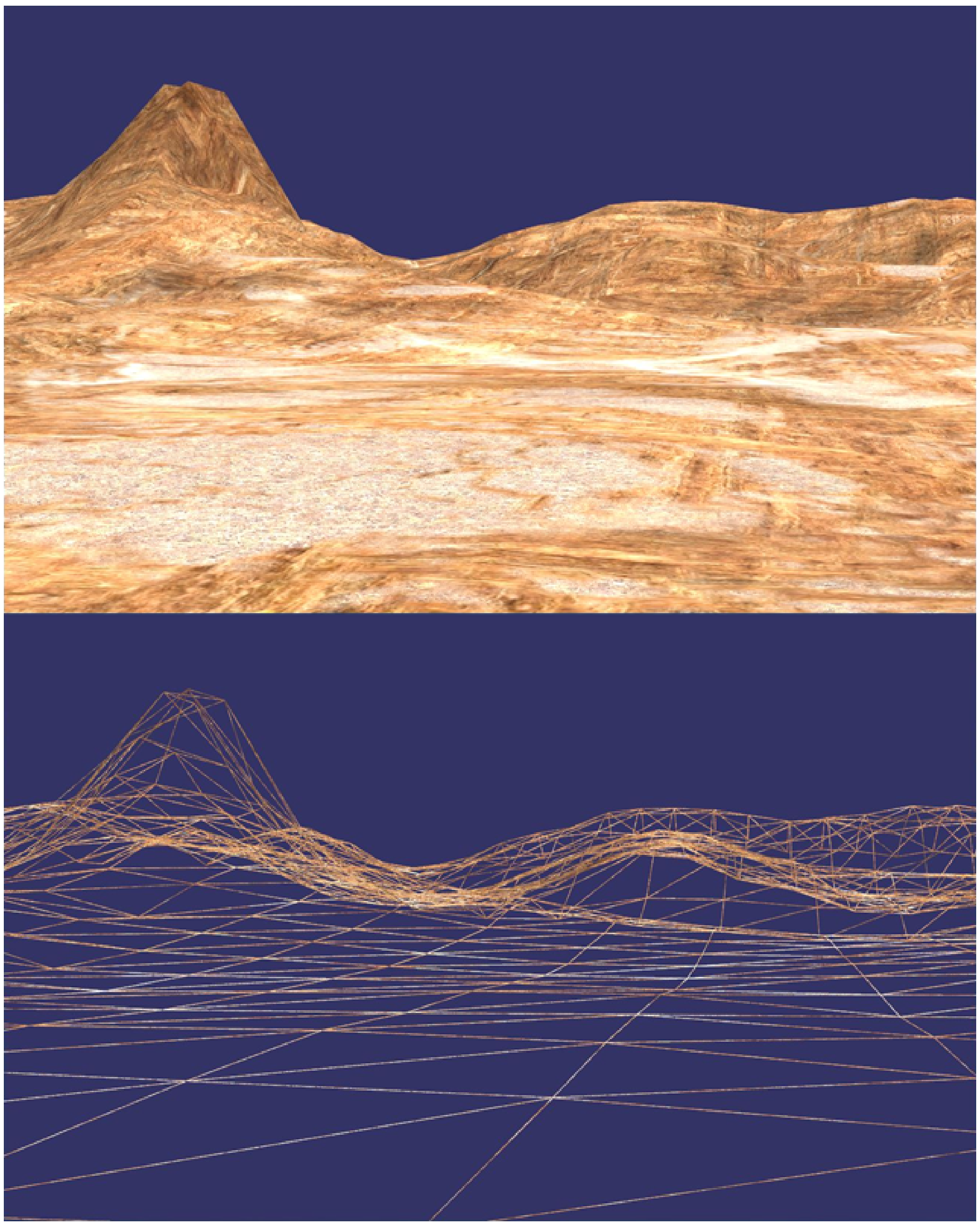

Figure 13: Tessellation level set to 1

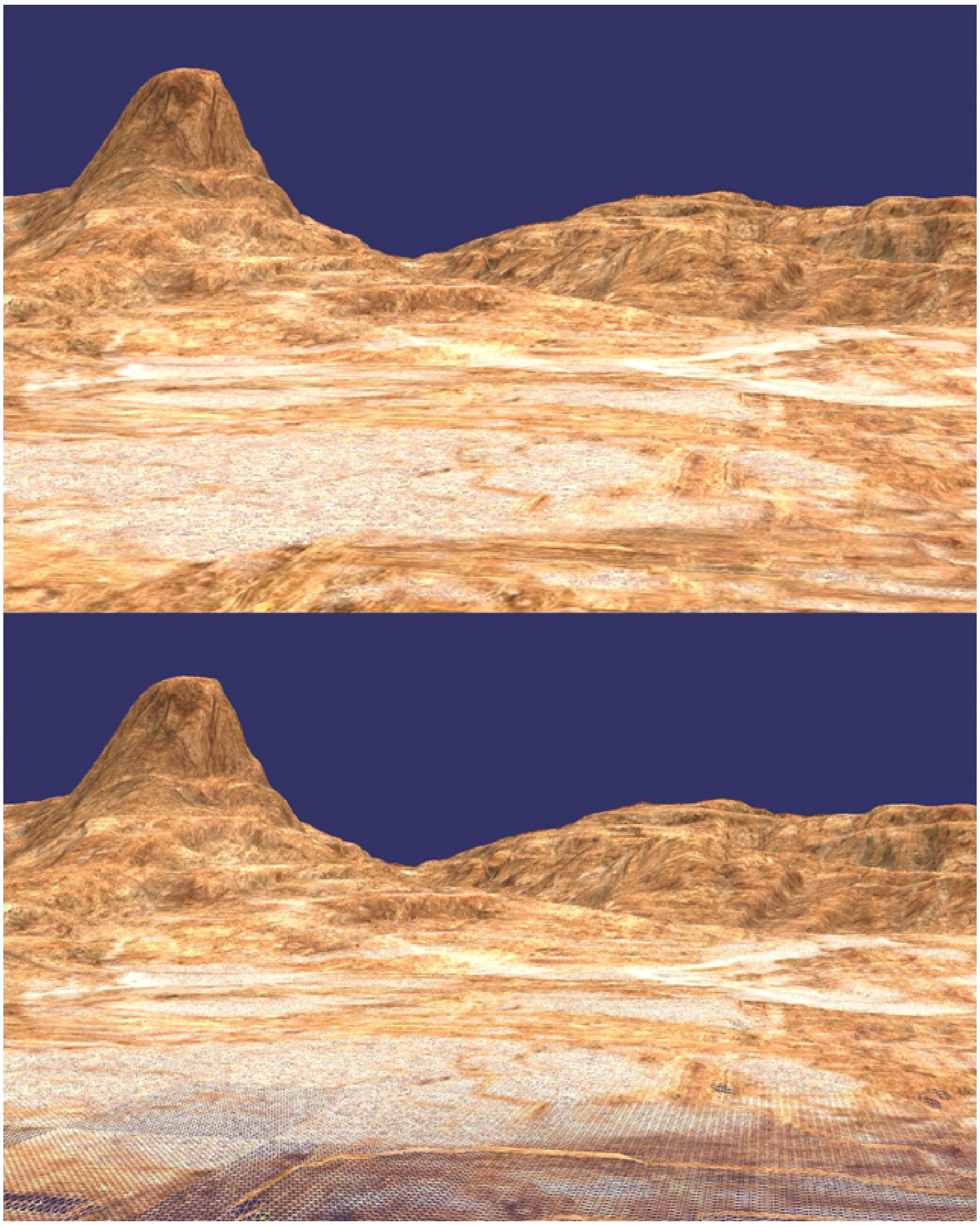

Figure 14: Tessellation level set to 64

5.1.2 Distance tessellation

This is the level of detail tessellation mode. The closer we are to the patch, the more it tessellates. As we can see in figure 15, the distance factor is multiplied by 4 (userDecidedLevel = 4). This could be set up to 64, but it is easier to see the triangles like this in the demonstration image. The evaluation shader takes in fractional numbers in this example. Both radius and tessellation level should be set according to hardware and terrain to get the best result.

5.1.3 Tessellation map

It is important to consider camera position and tessellated area when evaluating the FPS from table 4. To show how tessellation map factor works, the camera is lifted to get an overview. Because of this, the FPS will be different. The view is changed and we see less or more patches. In figure 16 we can see a more tessellated area, that could be a road, in the wireframe model. When other models are put on the terrain, the terrain needs to be accurate to prevent the models to either fly over or be buried under the ground. Vegvesenet have more accuracy on the road than the terrain. This is a problem when the terrain is coming up through the road. The only way to fix this is to raise the road, or with the tessellation we could send in values of the road to lower the ground.

Figure 17 shows how the tessellation map factor adapts to the terrain. It is able to reach high tessellation in just one patch. Note that the patches have the same tessellation level as neighbouring patches against shared border.

5.1.4 Normal tessellation

Figure 18 shows the normal tessellation factor in use. When we use this factor there is no need for extra tessellation on the silhouette. The normal tessellation factor will kick in when the terrain goes over a peak, because of the changes

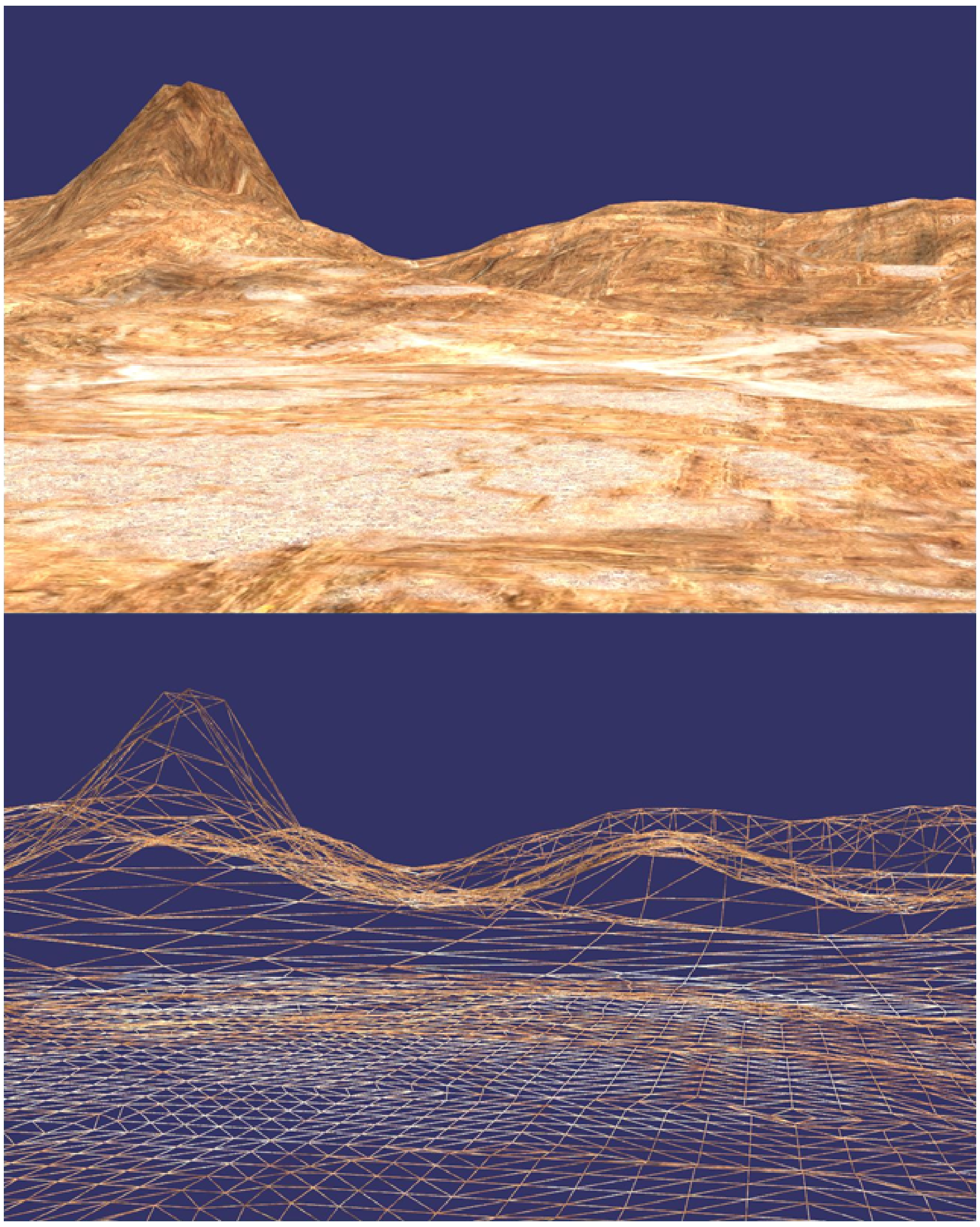

Figure 15: Distance tessellation with userDecidedLevel set to 4

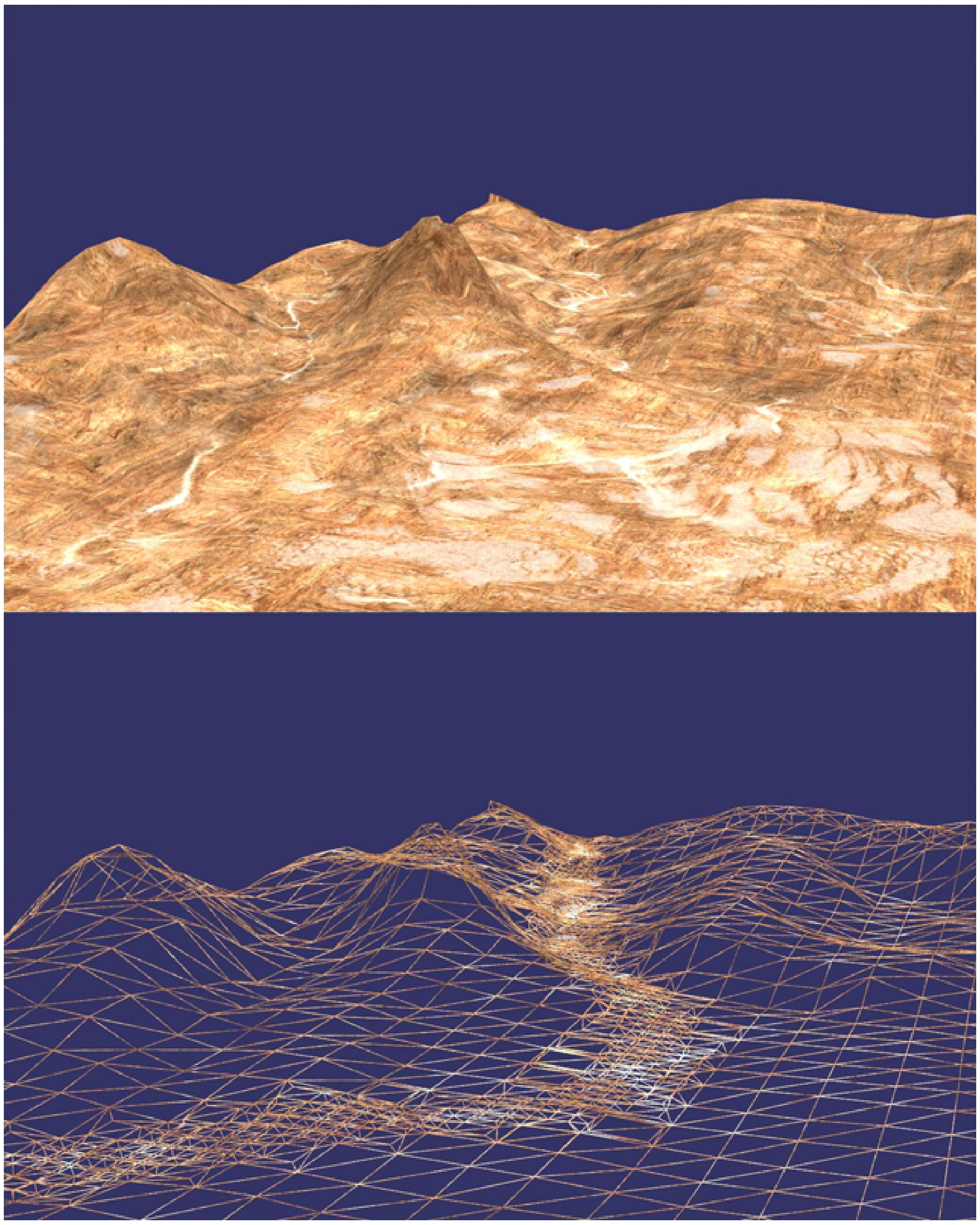

Figure 16: Tessellation map with userDecidedLevel set to 4

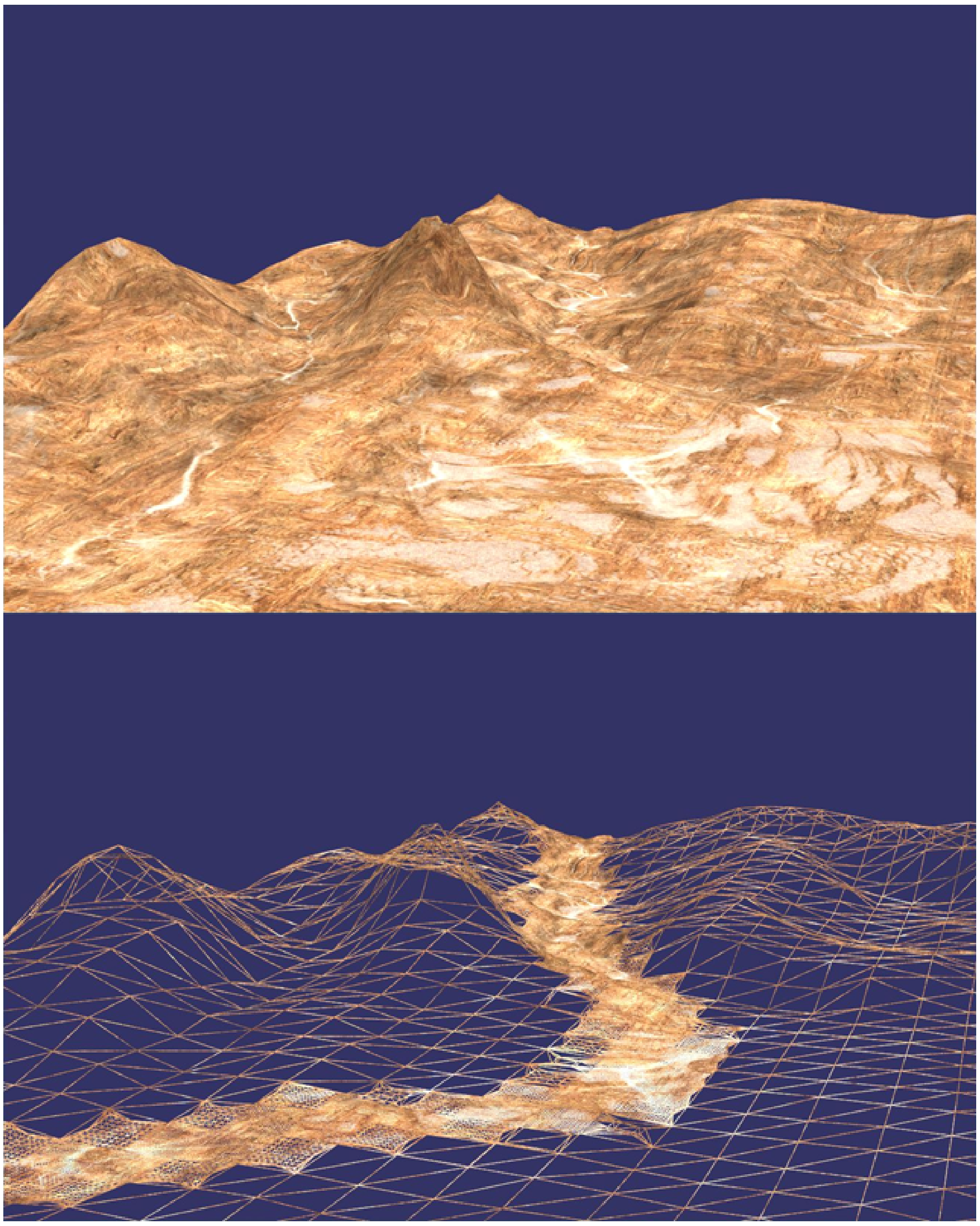

Figure 17: Tessellation map with userDecidedLevel set to 18

in normals. If we look at the figure we can see that the step has tessellation factor 1. This is because the normal is in the same direction. If you see in the hillside where the terrain alters rapidly, the tessellation factor is set up to give an accurate representation of the terrain. This tessellation factor will make sure that every important detail is shown. The normal factor is the most complicated factor of them all and uses a lot of calculating in the control shader, because it checks 24 normals for every patch. To improve this, we can set a distance for where we should use the normal tessellation and we could go over to silhouette tessellation instead, because it just checks 4 normals for every patch. When we use silhouette tessellation the only areas that will be tessellated are the patches touching the silhouette, in addition to normal tessellation that tessellates more where the terrain bends.

5.1.5 Tessellation formula

The results of all the factors combined together with tessellation formula is shown in figure 19. For all the factors: normal, distance, the tessellation map and the static factor creates this image. The closest patches are divided more than the flat patches far away. The road is tessellated, and the silhouette that consists of mountains is smooth, because the normal factor detects the changes when following over the peak. When the userDecidedLevel is changed, it will affect all the factors. In this figure, the userDecidedLevel is set to 10, but could be set up to 64. This is the final result for this thesis and can be used for tessellation rendering in simulations and games. Each factor can also change their weight to dominate more and to make it easier to switch between different programs.

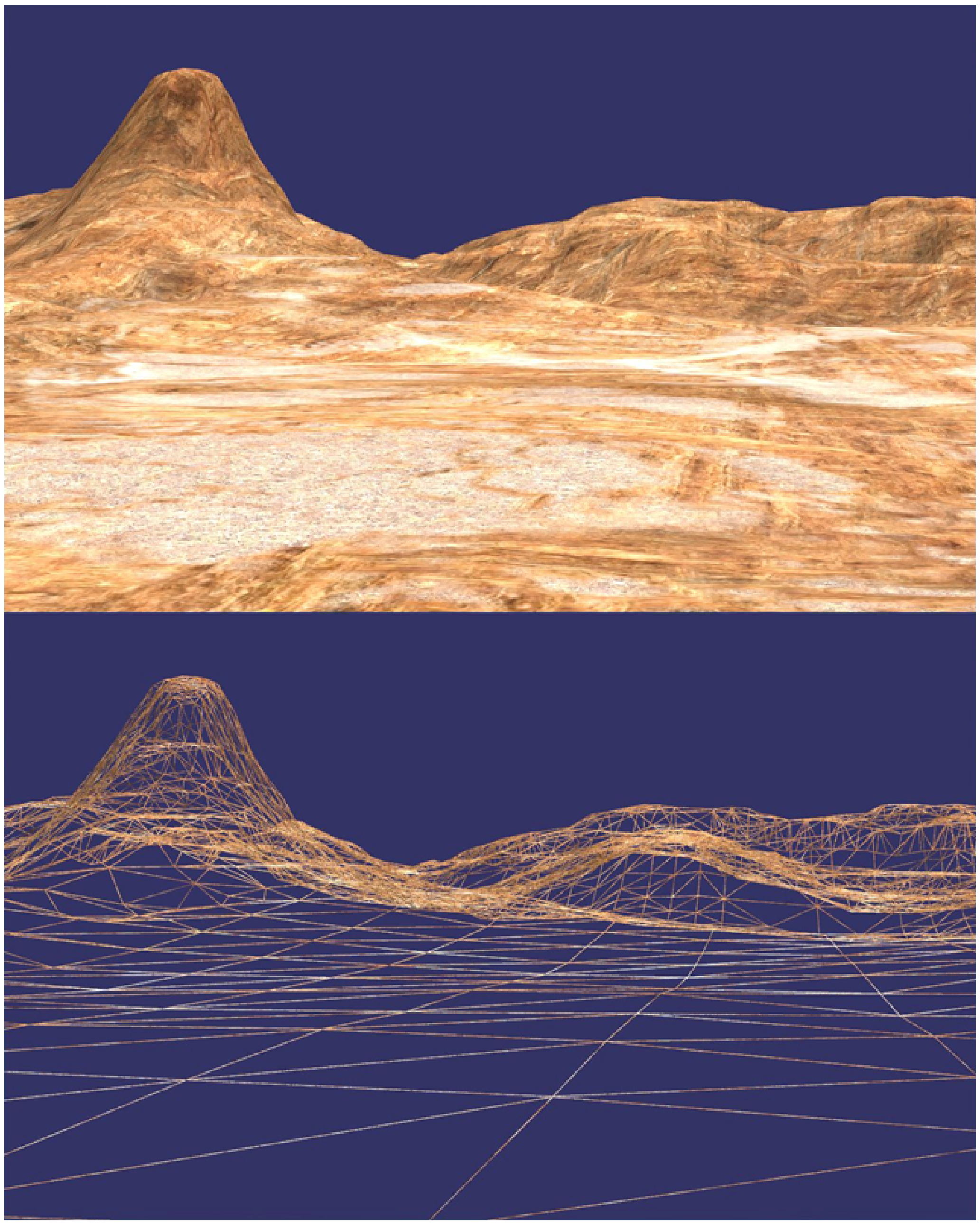

Figure 18: Normal tessellation with userDecidedLevel set to 2

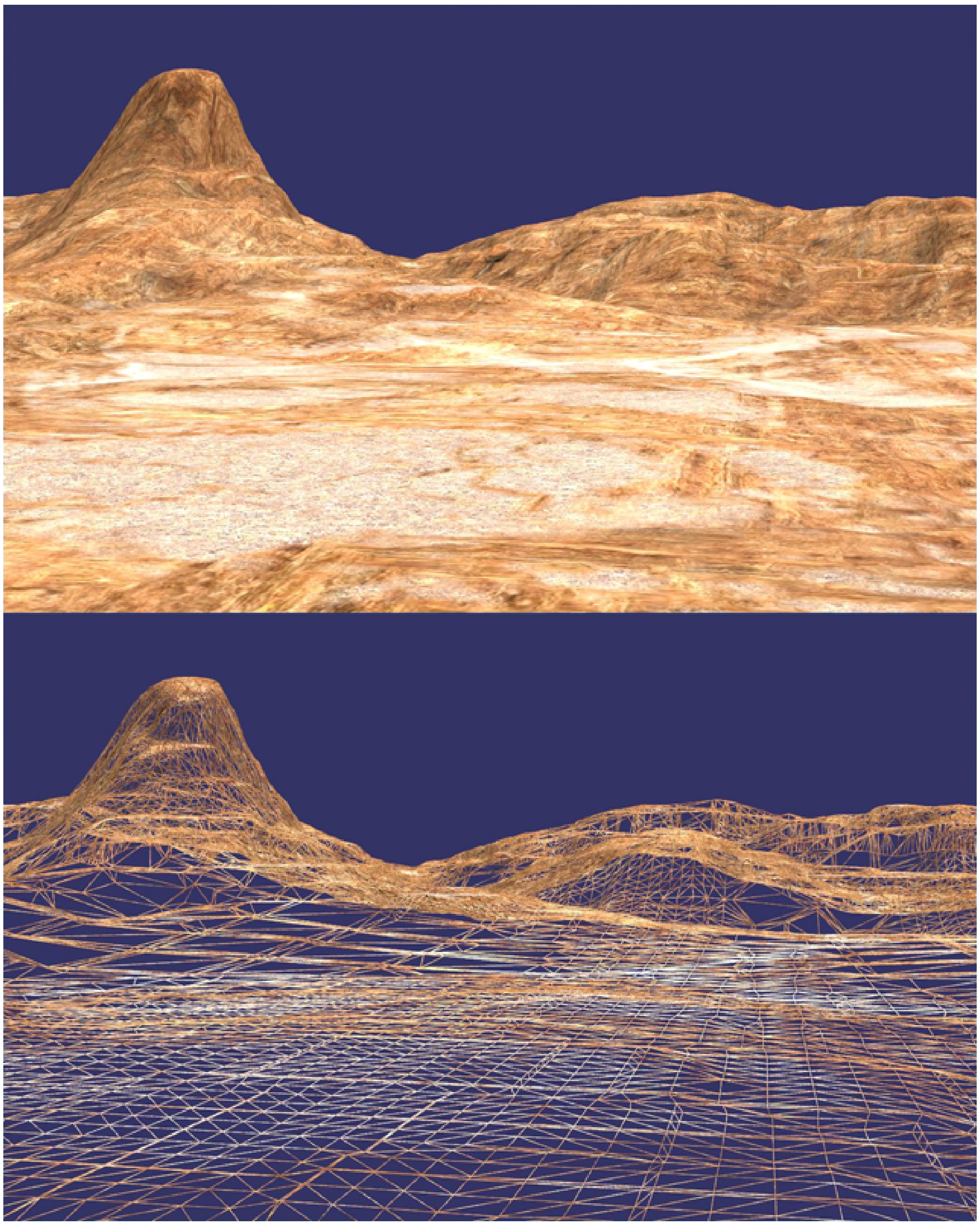

Figure 19: Tessellation formula with userDecidedLevel set to 10

5.2 Single texture terrain

The single texture terrain gives a simpler fragment shader that runs faster. In these images the line mode (wireframe) is slower than the fill mode. The FPS is shown in table 5. These images use the tessellation formula with all factors. All the factors have the same weight in this example.

Table 5: FPS table single texture

Location	Tessellating factor	Tessellation level	Line or Fill	FPS	Screen resolution
Sekken	All Factors	16	F	470	1280 x 800
Sekken	All Factors	16	L	386	1280 x 800
Trondheim	All Factors	16	F	461	1280 x 800
Trondheim	All Factors	16	L	356	1280 x 800

5.2.1 Sekken

This island from figure 20, is made from the height map made by contour lines. We can see how the ocean is just tessellated by the constant factor. The camera is too far away for the distance factor to have any effect.

5.2.2 Trondheim

Trondheim, in figure 21, shows another terrain with single texture. This is one of the current terrains used in the car and truck simulator at Vegvesenet. The data was given by Jo Skjermo. The camera is placed over the city heading for Gløshaugen. It is hard to see the contours of the terrain at this camera height, but it is to show how the texture fits the terrain. When the camera is placed closer to the ground, we can see the terrain better.

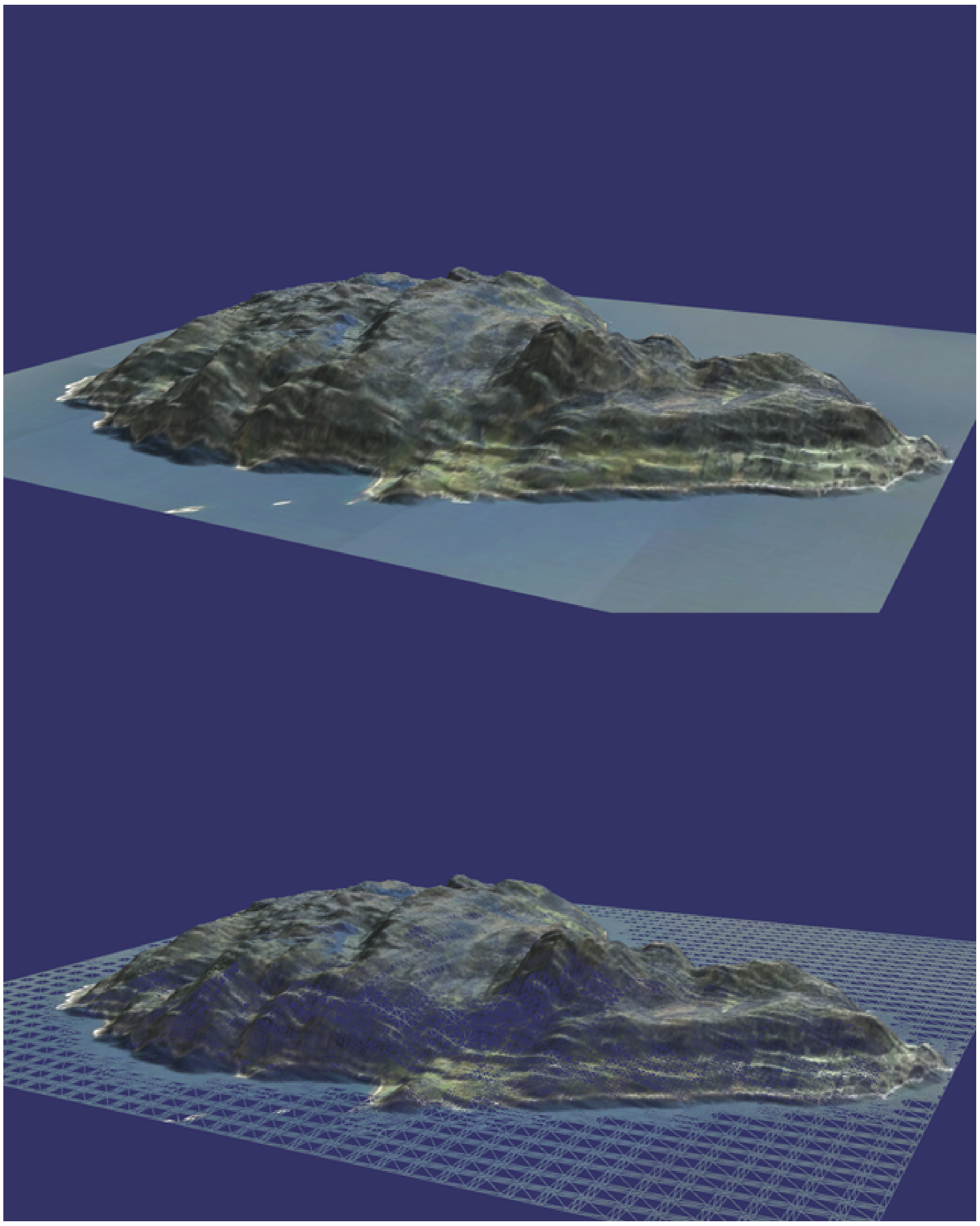

Figure 20: Sekken

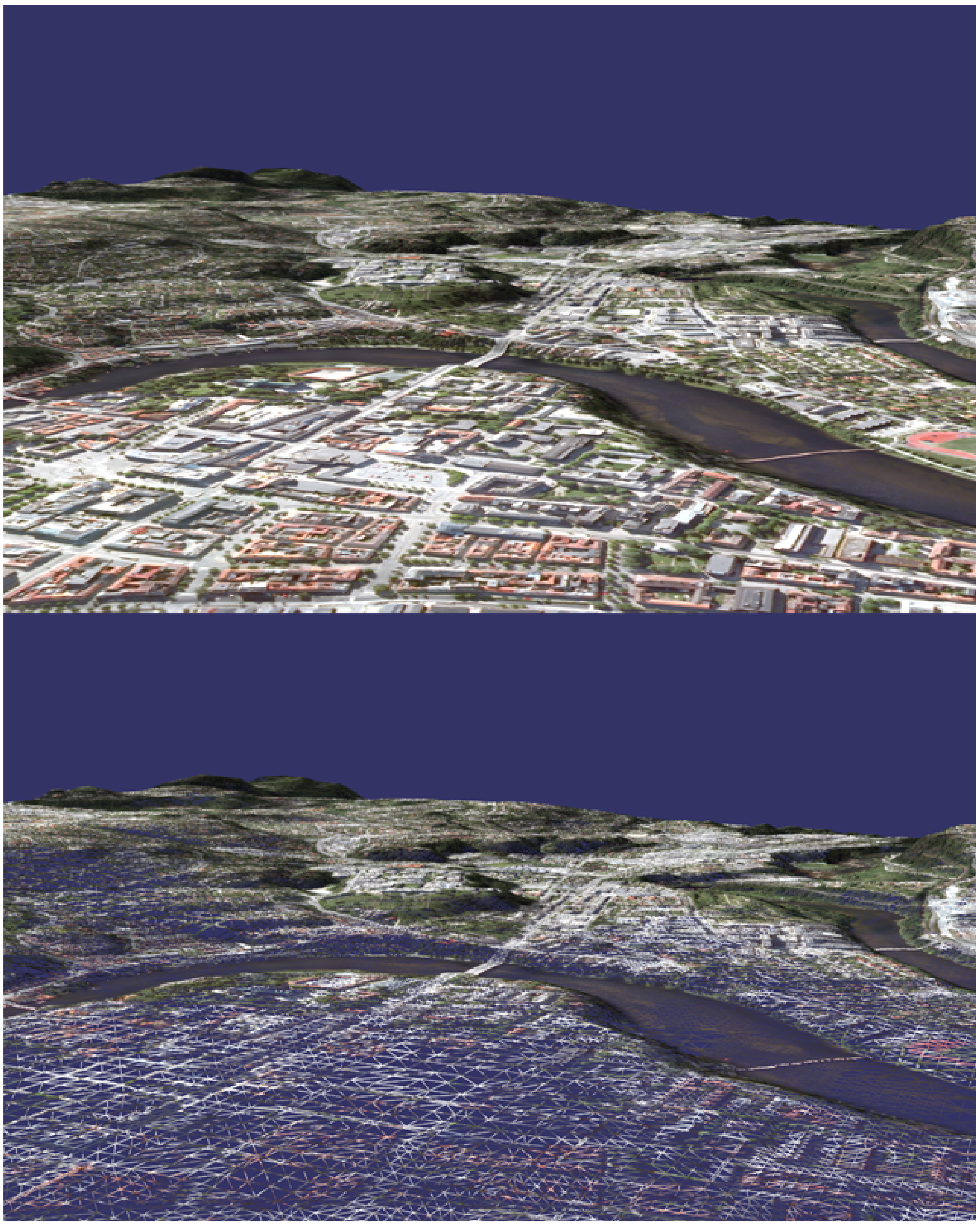

Figure 21: Trondheim

6 Discussion

The result of this work is an executable terrain tessellation program with OSG and GLSL shaders. The control shader has three tessellation factors: distance, a tessellation map and the normals to the patch. The fourth factor is a constant decided by user. This constant helps to divide the patches which never satisfy the conditions to be tessellated. This constant helps to make a smoother terrain when the tessellation level is set close to maximum. All of these factors can vary from zero to one. Control shader also includes weights for each factor. The weights are used to specify how much each factor affects the tessellation level, because the terrain can be built in many ways. The weights create a more general formula and can be used for all terrain and simulators. Changing the weights gives the user opportunity to adjust for the best result. The sum of all factors and weights should be between zero and one, and then multiplied with our userDecidedLevel, set from input, which varies from zero to sixty three. This is done because the final edge tessellation level must be between zero and sixty four. The formula starts at one and goes up to sixty four. If one patch has zero, it will not be drawn. This is something that can be done in further work, by setting patches known for sure are out of view to zero. This program can help the OSG environment to use tessellation because there is little work done by now. OSG 3,0 is based on OpenGL 4,0, and therefore supports tessellation. To use tessellation in OSG has been harder than OpenGL, because the parameters are set in a different way, and when we look in the documentation for the OSG, some of the set-methods haven't been made yet. This is because we can go to other set methods in OSG that can do the same. The first tessellation program was in OpenGL, because it was better documented and some tutorial was available. OpenGL helped me understand tessellation and some of the error codes from the shaders. When I had the OpenGL program running, I knew that the hardware supported the shaders.

Then I could concentrate on the OSG to look for errors in reading in shaders and execute them. This program can help people, starting with tessellation in OSG. The threshold to create a program that can tessellate gets smaller when a similar program exists. The most challenging problem in the start of this project was to get OSG and Visual Studio correctly set up. All the packages and parameters needs to be correct. This work proves that OSG and the tessellation shaders cooperate well together. This is important for continuing using OSG, since tessellation has come to stay. Tessellation is important because it makes a good LOD. Terrain is difficult to render since it consist of large models and needs to be continuous with no gaps between the parts. An ideal frame would be that all primitives have the same size, no matter if you're looking at the horizon or the close terrain. With the exception: the parts that need higher accuracy, like silhouette and roads. The tessellation formula presented in this thesis creates a good LOD, because of the distance factor. The silhouette becomes smooth because we have high level of tessellation where the derivation of the terrain curve is large. This is checked with the normal factor that looks at the changes on the normals to decide tessellation level. These tessellation factors will make a more detailed terrain, and at the same time checks for over-tessellation, which will increase the FPS. FPS tables for the multi textures and single textures, shows that the fragment shaders for the multi textures are complex and uses too many textures. If we had to improve the FPS, this would be the place to start. In the single textures we can see that the fill-mode has higher FPS than the wireframe / line mode, but in the multi textures the wireframe modes have significantly higher FPS. This because the multi texture fragment shader takes in too many textures, and becomes slower than in the wireframe mode. Three different terrains are viewed in this thesis to show how easy it is to modify the code to take in other terrain. Either to change the number of patches, or number of textures. If you look at figure 19, which is the complete results of all factors with tessellation level ten, you can see that the closest patches are divided more than the flat patches far away. If you remember where the road went you can see that the terrain is tessellated more there too, and the silhouette that consists of mountains are smooth because of the change in normals when you follow over the peak. The figures of Sekken and Trondheim also use the full formula of tessellations at the userDecidedLevel of sixteen. Each resolution of the window was 1280 * 800. If the resolution is set down, the FPS will increase drastically and the other way around if the resolution is set higher.

7 Conclusion and further work

Tessellation is a powerful feature. Numbers of vertices is reduced when sent to the graphic card, and this reduces bandwidth and memory usage. In this thesis, tessellation was used to get a better accuracy of the terrain. The thesis looks at adapting tessellation for terrain rendering. The solution is a program with three variable tessellation factors and one static. We can decide how much each factor will affect the tessellation of the terrain. The evaluation shader uses a height map to displace the position of each new vertex. At the start Bèzier patches was used to find the positions, but that became too inaccurate, because the data from Vegvesenet only has a height for every tenth meter. If we skip sixty four pixels in the height map, it means that we have only one correct position for every 160 metres, if we use sixteen vertices to make one patch. The height map is available in the evaluation shaders and can find exact values for all new vertices. Now we have an accurate terrain where we need it. In the control shader, a struct is holding the distance to every vertex, where we test for the normal. Normal based tessellation is used on the whole terrain, just based on normals, but in further work we could for instance restrain the distance of where the normal factor should be applied. Another thing that could be done is to use bump maps together with texture mask. If for instance the terrain has cobblestone road, the bump map will adjust the heights so that the stones will stand out. Other factors that can be used are to calculate tessellation factor for silhouettes, or make a grid for the camera that tessellates based on the screen space. This means that the closest patches will be tessellated more and the distant patches tessellated less. This has been done before by Livny et.al. [34]. We can also check if the patches are shown on the screen. If they don't show, we set the tessellation factor to zero. Then none of the vertices for this patch will be passed on.

References

[1] Dave Shreiner OpenGL Programming Guide

[2] Overview of OpenGL 4.0 http://ptgmedia.pearsoncmg.com/imprint_downloads/informit/promotions/siggraph2010/OpenGL4Overview.pdf [May 11, 2012]

[3] http://www.lighthouse3d.com/opengl/terrain/index.php3?heightmap [May 11, 2012]

[4] OpenGL Tutorial http://ogldev.atspace.co.uk/ [May 11, 2012]

[5] http://en.wikipedia.org/wiki/Scene_graph [May 11, 2012]

[6] Paul Martz OpenSceneGraph Quick Start Guide A Quick Introduction to the Cross-Platform Open Source Scene Graph API, 2007

[7] http://www.stackedboxes.org/l̃mb/asittbpo-open-scene-graph/chapter-3-more-state-lights-textures-and-shaders/ [May 11, 2012]

[8] http://www.cuboslocos.com/ [February 15, 2012]

[9] Tim Moore Open Scene Graph States and StateSets http://www.bricoworks.com/articles/stateset/stateset.html [February 15, 2012]

[10] Rui Wang Xuelei Qian OpenSceneGraph: Advanced Scene Graph Components http://www.packtpub.com/article/openscenegraph-advanced-scene-graph-components [June 4, 2012]

[11] http://binglongx.wordpress.com/2011/04/24/openscenegraph-for-visual-studio-on-windows-setup/ [June 4, 2012]

[12] http://forum.openscenegraph.org/ [June 4, 2012]

[13] OpenSceneGraph Documentation http://www.fi.muni.cz/ xbezdeka/files/osg-doxygen-3.0.0/core/index.html [June 4, 2012]

[14] http://www.openscenegraph.org/documentation/NPSTutorials/ [June 4, 2012]

[15] http://www.openscenegraph.org/projects/osg/wiki/Support/ProgrammingGuides [June 4, 2012]

[16] http://www.openscenegraph.org/projects/osg/wiki/Support/Tutorials [June 4, 2012]

[17] Mourad Boufarguine http://www.mail-archive.com/osg-users@lists.openscenegraph.org/msg30381.html [June 4, 2012]

[18] Rui Wang and Xuelei Qian OpenSceneGraph 3.0 Beginner's Guide, December 2010

[19] Mikael Drugge OpenSceneGraph Tutorial, 2006

[20] John Kessenich The OpenGL Shading language Luanguage Version 4.10

[21] http://www.swiftless.com/tutorials/opengl4/2-opengl-shaders.html [April 4, 2012]

[22] Patrick Cozzi Introduction to GLSL

[23] Jacobo Rodriguez Villar OpenGL Shading Language Course

[24] Mike Weiblen GLSL Shading with OpenSceneGraph, 2005

[25] http://www.msi.com/product/vga/N560GTX-Ti-Twin-Frozr-II-OC.html [May 11, 2012]

[26] Michael Bunnell http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter07.html [May 11, 2012]

[27] Natalya Tatarchuk Real-Time Tessellation on GPU

[28] nvidia http://www.nvidia.com/object/tessellation.html [May 11, 2012]

[29] Daniel Rákos http://rastergrid.com/blog/2010/09/history-of-hardware-tessellation/ [May 11, 2012]

[30] Florian Boesch http://codeflow.org/entries/2010/nov/07/opengl-4-tessellation/ [May 11, 2012]

[31] Philip Rideout Triangle Tessellation with OpenGL 4.0 http://prideout.net/blog/?p=48 [May 11, 2012]

[32] Philip Rideout Quad Tessellation with OpenGL 4.0 http://prideout.net/blog/?p=49 [May 11, 2012]

[33] Frank Losasso and Hugues Hoppe Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids

[34] Yotam Livny, Neta Sokolovsky, Tal Grinshpoun and Jihad El-Sana A GPU persistent grid mapping for terrain rendering

[35] Kevin Moule and Michael D. McCool Efficient Bounded Adaptive Tessellation of Displacement Maps

[36] Jan Kautz and Hans-Peter Seidel Hardware Accelerated Displacement Mapping for Image Based Rendering

[37] Xiaohuan CorinaWang, Jèrôme Maillot, Eugene Fiume Victor Ng-Thow-Hing, AndrewWoo and Sanjay Bakshi Feature-based DisplacementMapping

[38] Bézier http://en.wikipedia.org/wiki/Bézier_curve [June 9, 2012]

[39] Christopher Dyken and Martin Reimers and Johan Seland Real-Time GPU Silhouette Refinement using adaptively blended Bézier Patches

[40] Ryan Geiss GPU Gems 3

[41] Timothy S. Newman and Hong Yi A survey of the marching cubes algorithm

[42] Mike Bailey http://web.engr.oregonstate.edu/ mjb/cs519/ [May 11, 2012]

[43] http://recreationstudios.blogspot.com/2010/03/simple-tessellation- example.html

[44] Microsoft http://msdn.microsoft.com/en-us/library/windows/ desktop/ff476340(v=vs.85).aspx [May 11, 2012]

[45] OpenGL http://www.clockworkcoders.com/oglsl/ [May 11, 2012]

[46] Multi-Texture http://www.lighthouse3d.com/tutorials/glsl-tutorial/multi-texture/ [May 11, 2012]

[47] Mourad Boufarguinel create a image http://www.mail-archive.com/osg-users@lists.openscenegraph.org/msg30381.html [May 11, 2012]

[48] World, View and Projection Matrix Unveiledl http://robertokoci.com/world-view-projection-matrix-unveiled/ [May 11, 2012]

[49] http://www.norgeibilder.no/ [May 11, 2012]

[50] http://rastertek.com/tutindex.html [May 11, 2012]

[51] Michael Doggett and Johannes Hirche Adaptive View Dependent Tessellation of Displacement Maps

[52] Yotam Livny, Zvi Kogan and Jihad El-Sana Seamless patches for GPU-based terrain rendering

[53] John McDonald Tessellation on Any Budget

[54] Iain Cantlay DirectX 11 Terrain Tessellation

[55] Bill Bilodeau and Peter Lohrmann Tessellation in a Low Poly World

[56] Jon Story and Cem Cebenoyan Tessellation Performance

[57] Edward Angel Interactive Computer Graphics, fifth edition

[58] Hearn Baker Computer Graphics with OpenGL, third edition

[59] Dave Shreiner OpenGL Programming Guide, seventh edition

[60] David Wolff OpenGL 4.0 Shading Language Cookbook

A Appendix

A.1 Vertex shader

```
//
// Vertex shader.
//

// Set GLSL version. Minimum version that supports
   tessellation is 4.0.
#version 400

// Gets in position.
in vec4 position;

// Sends vertex position out to tessellation control shader
   .
out vec4 vPosition;

// Uniform set by OSG.
uniform sampler2D heightMap;
uniform float scaleXY;
uniform float scaleZ;

//
// Main
//
void main(void)
```

```
{
  // Set texture coordinate based on position and scale in
     XY plane.
  vec2 textureCoordinate = position.xy / scaleXY;

  // Use texture coordinate to find the value on the height
      Map,
  // multiply it with scale (along the Z axis) to find the
     height.
  float height = texture(heightMap, textureCoordinate).r *
     scaleZ;

  // Set the new vertex position.
  vec4 positionWithHeight = vec4(position.x, position.y,
     height, 1.0);

  // Sends vertex position to the tessellation control
     shader.
  vPosition = positionWithHeight;
}
```

A.2 Tessellation control shader

```
//
// Tessellation control shader.
//

// Set GLSL version. Minimum version that supports
   tessellation is 4.0.
#version 400

// Specify numbers of vertices out.
layout(vertices = 4) out;

// Gets in vertex position from the vertex shader.
in vec4 vPosition[];

// Sends vertex position out to the tessellation
   evaluations shader.
out vec4 tcPosition[];

// Uniforms.
uniform float tessellationLevel;
uniform float scaleXY;
uniform float scaleZ;

// Uniform cameraPosition used with distance calculations.
uniform vec3 cameraPosition;

```

```
// Uniform tessellationMap and normalMap.
uniform sampler2D tessellationMap;
uniform sampler2D normalMap;

// Get ID.
#define ID gl_InvocationID

//
// NormalFromTexture takes in the colour value from normal
   map and subtract 0.5,
// since the normal Map zero starts on 0,5, then we get the
    values from - 0,5 to 0,5.
// To the end we normalize the vector.
//
vec3 NormalFromTexture(vec3 normal)
{
  normal.x = normal.x - 0.5;
  normal.y = normal.y - 0.5;
  normal.z = normal.z - 0.5;
  normal = normalize(normal);
  return normal;
};

//
// DistanceFactor
//
float DistanceFactor(float factor)
```

```
{
  int limit = 50;

  if(factor > limit)
  {
    factor = 0;
  }
  else
  {
    factor = (50 - factor)/10;
  }
  return factor;
};

//
//  vertex struct contains normal and distance to a vertex
    from cameraPosition.
//
struct vertex
{
  vec3 vertexNormal;
  float distance;
};

//
// normalTessFactor
//
```

```
float normalTessFactor(vertex [6] array)
{
  // calculate the angles between opposite corners.
  float testangle = acos( dot(array[0].vertexNormal, array
      [3].vertexNormal));
  float testangle90 = acos( dot(array[2].vertexNormal,
      array[1].vertexNormal));
  float testangle2 = acos( dot(array[0].vertexNormal, array
      [5].vertexNormal));
  float testangle902 = acos( dot(array[1].vertexNormal,
      array[4].vertexNormal));


  float factor = 0;

  if( testangle > 0.90 || testangle90 > 0.90|| testangle2 >
       0.90 || testangle902 > 0.90)// uses radians
  {
    return (testangle + testangle90 + testangle2 +
       testangle902);
  }
  return 0;

};

//
// Main.
```

```
//
void main()
{
  // Passes through the vertex position.
  tcPosition[ID] = vPosition[ID];

  // Test that ID is zero.
  if (ID == 0)
  {

    //
    // Tessellation based on normals.
    //

    // Declare vertex struct for each side of the patch.
    vertex [6] abvertices;
    vertex [6] advertices;
    vertex [6] dcvertices;
    vertex [6] bcvertices;

    /*
    // Vectors to make parallel patch edge vectors to
       compare normals.
    vec2 left = vec2(-2,0 );
    vec2 right = vec2(2,0 );
    vec2 up = vec2(0,2);
    vec2 down = vec2( 0,-2 );
    */
    // Vectors to make parallel patch edge vectors to
       compare normals.
    vec2 left = vec2(vPosition[0].xy -vPosition[1].xy);
    vec2 right = vec2(vPosition[1].xy-vPosition[0].xy);
    vec2 up = vec2(vPosition[1].xy-vPosition[2].xy);
    vec2 down = vec2( vPosition[2].xy-vPosition[1].xy);

    // abVector.
    // Set normal and distance to the two vertices in the
       abVector.
    abvertices[0].vertexNormal = NormalFromTexture(texture(
       normalMap,(vPosition[0].xy )/ scaleXY).xyz);
    abvertices[0].distance = distance((vPosition[0].xy),
       cameraPosition.xy);
    abvertices[1].vertexNormal = NormalFromTexture(texture(
       normalMap,(vPosition[1].xy  )/ scaleXY).xyz);
    abvertices[1].distance =  distance((vPosition[1].xy ),
       cameraPosition.xy);

    // Set normal and distance to the two vertices (in the
       abVector) added the left vector.
    abvertices[2].vertexNormal = NormalFromTexture(texture(
       normalMap,(vPosition[0].xy + left)/ scaleXY).xyz);
    abvertices[2].distance =  distance((vPosition[0].xy +
       left), cameraPosition.xy);
```

```
abvertices[3].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[1].xy+ left)/ scaleXY).xyz);
abvertices[3].distance = distance((vPosition[1].xy+
    left), cameraPosition.xy);

// Set normal and distance to the two vertices (in the
    abVector) added the right vector.
abvertices[4].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[0].xy+ right)/ scaleXY).xyz);
abvertices[4].distance =  distance((vPosition[0].xy+
    right), cameraPosition.xy);
abvertices[5].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[1].xy + right)/ scaleXY).xyz);
abvertices[5].distance = distance((vPosition[1].xy+
    right), cameraPosition.xy);


// adVector
// Set normal and distance to the two vertices in the
    adVector.
advertices[0].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[0].xy )/ scaleXY).xyz);
advertices[0].distance = distance((vPosition[0].xy),
    cameraPosition.xy);
advertices[1].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[3].xy  )/ scaleXY).xyz);
advertices[1].distance =  distance((vPosition[3].xy ),
    cameraPosition.xy);

// Set normal and distance to the two vertices (in the
    adVector) added the down vector.
advertices[2].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[0].xy + down)/ scaleXY).xyz);
advertices[2].distance =  distance((vPosition[0].xy +
    down), cameraPosition.xy);
advertices[3].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[3].xy+ down)/ scaleXY).xyz);
advertices[3].distance = distance((vPosition[3].xy+
    down), cameraPosition.xy);

// Set normal and distance to the two vertices (in the
    adVector) added the up vector.
advertices[4].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[0].xy+ up)/ scaleXY).xyz);
advertices[4].distance =  distance((vPosition[0].xy+ up
    ), cameraPosition.xy);
advertices[5].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[3].xy+ up)/ scaleXY).xyz);
advertices[5].distance = distance((vPosition[3].xy+ up)
    , cameraPosition.xy);

// dcVector. Do the same to dcVector as abVector
   because they are parallel.
dcvertices[0].vertexNormal = NormalFromTexture(texture(
   normalMap,(vPosition[3].xy )/ scaleXY).xyz);
dcvertices[0].distance = distance((vPosition[3].xy),
   cameraPosition.xy);
dcvertices[1].vertexNormal = NormalFromTexture(texture(
   normalMap,(vPosition[2].xy  )/ scaleXY).xyz);
dcvertices[1].distance =  distance((vPosition[2].xy ),
   cameraPosition.xy);

dcvertices[2].vertexNormal = NormalFromTexture(texture(
   normalMap,(vPosition[3].xy + left)/ scaleXY).xyz);
dcvertices[2].distance =  distance((vPosition[3].xy +
   left), cameraPosition.xy);
dcvertices[3].vertexNormal = NormalFromTexture(texture(
   normalMap,(vPosition[2].xy+ left)/ scaleXY).xyz);
dcvertices[3].distance = distance((vPosition[2].xy+
   left), cameraPosition.xy);

dcvertices[4].vertexNormal = NormalFromTexture(texture(
   normalMap,(vPosition[3].xy+ right)/ scaleXY).xyz);
dcvertices[4].distance =  distance((vPosition[3].xy+
   right), cameraPosition.xy);
dcvertices[5].vertexNormal = NormalFromTexture(texture(
   normalMap,(vPosition[2].xy+ right)/ scaleXY).xyz);
dcvertices[5].distance = distance((vPosition[2].xy+
    right), cameraPosition.xy);


// bcVector. Do the same to bcVector as adVector
    because they are parallel.
bcvertices[0].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[1].xy )/ scaleXY).xyz);
bcvertices[0].distance = distance((vPosition[1].xy),
    cameraPosition.xy);
bcvertices[1].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[2].xy  )/ scaleXY).xyz);
bcvertices[1].distance =  distance((vPosition[2].xy ),
    cameraPosition.xy);

bcvertices[2].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[1].xy + down)/ scaleXY).xyz);
bcvertices[2].distance =  distance((vPosition[1].xy +
    down), cameraPosition.xy);
bcvertices[3].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[2].xy+ down)/ scaleXY).xyz);
bcvertices[3].distance = distance((vPosition[2].xy+
    down), cameraPosition.xy);

bcvertices[4].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[1].xy+ up)/ scaleXY).xyz);
bcvertices[4].distance =  distance((vPosition[1].xy+ up
    ), cameraPosition.xy);
bcvertices[5].vertexNormal = NormalFromTexture(texture(
    normalMap,(vPosition[2].xy+ up)/ scaleXY).xyz);
bcvertices[5].distance = distance((vPosition[2].xy+ up)
    , cameraPosition.xy);

// Send the arrays to the normalTessFactor.
float abNormalTessellation  = normalTessFactor(
    abvertices);
float adNormalTessellation  = normalTessFactor(
    advertices);
float dcNormalTessellation  = normalTessFactor(
    dcvertices);
float bcNormalTessellation  = normalTessFactor(
    bcvertices);

// Calculate the average to the inner tessellation.
float innerNormalTessellation  = (abNormalTessellation
     + adNormalTessellation  + dcNormalTessellation  +
    bcNormalTessellation )/4;

//
// Distance
//

// calculate the position to the edges of the patch.
// Use the coordinate from the terrain.
vec3 abMidPos = vec3(vPosition[0].xyz + ((vPosition[1].
   xyz - vPosition[0].xyz)/2));
vec3 adMidPos = vec3(vPosition[0].xyz + ((vPosition[3].
   xyz - vPosition[0].xyz)/2));
vec3 dcMidPos = vec3(vPosition[3].xyz + ((vPosition[2].
   xyz - vPosition[3].xyz)/2));
vec3 bcMidPos = vec3(vPosition[1].xyz + ((vPosition[2].
   xyz - vPosition[1].xyz)/2));

// Get tessellation factor by sending it to
   DistanceFactor.
float abDistanceTessellation = DistanceFactor(distance(
   abMidPos, cameraPosition.xyz));
float adDistanceTessellation = DistanceFactor(distance(
   adMidPos, cameraPosition.xyz));
float dcDistanceTessellation = DistanceFactor(distance(
   dcMidPos, cameraPosition.xyz));
float bcDistanceTessellation = DistanceFactor(distance(
   bcMidPos, cameraPosition.xyz));

// Calculate the average to the inner tessellation.
float innerDistanceTessellation = (
   abDistanceTessellation + adDistanceTessellation +
   dcDistanceTessellation + bcDistanceTessellation)/4;

//
// TessellationMap
//

// These coordinates is scaled to be used on the
   texture.
vec2 abMidTexPos = vec2((vPosition[0].xy + ((vPosition
   [1].xy - vPosition[0].xy)/2))/ scaleXY);
vec2 adMidTexPos = vec2((vPosition[0].xy + ((vPosition
   [3].xy - vPosition[0].xy)/2))/ scaleXY);
vec2 dcMidTexPos = vec2((vPosition[3].xy + ((vPosition
   [2].xy - vPosition[3].xy)/2))/ scaleXY);
vec2 bcMidTexPos = vec2((vPosition[1].xy + ((vPosition
   [2].xy - vPosition[1].xy)/2))/ scaleXY);

float abTextureTessellation = texture(tessellationMap,
   abMidTexPos).r;
float adTextureTessellation = texture(tessellationMap,
   adMidTexPos).r;
float dcTextureTessellation = texture(tessellationMap,
   dcMidTexPos).r;
float bcTextureTessellation = texture(tessellationMap,
   bcMidTexPos).r;

// Calculate the average to the inner tessellation.
float innerTextureTessellation = (abTextureTessellation
    + adTextureTessellation + dcTextureTessellation +
   bcTextureTessellation )/4;

// Tessellation with all the factors.

float normalTessellationWeight = 5;
float textureTessellationWeight = 5;
float distanceTessellationWeight = 5;
float constantTessellationWeight = 1;

gl_TessLevelInner[0] =  1 + ((
   constantTessellationWeight + innerNormalTessellation
    * normalTessellationWeight +
   innerTextureTessellation * textureTessellationWeight
    + innerDistanceTessellation *
   distanceTessellationWeight)/(
   constantTessellationWeight +
   normalTessellationWeight + textureTessellationWeight
    + distanceTessellationWeight ))*tessellationLevel;
gl_TessLevelInner[1] =  1 + ((
   constantTessellationWeight + innerNormalTessellation
    * normalTessellationWeight +
   innerTextureTessellation * textureTessellationWeight
    + innerDistanceTessellation *
   distanceTessellationWeight)/(
   constantTessellationWeight +
   normalTessellationWeight + textureTessellationWeight
    + distanceTessellationWeight ))*tessellationLevel;

gl_TessLevelOuter[0] = 1 + ((constantTessellationWeight
    + abNormalTessellation * normalTessellationWeight +
    abTextureTessellation * textureTessellationWeight +
    abDistanceTessellation * distanceTessellationWeight
   )/(constantTessellationWeight +
   normalTessellationWeight + textureTessellationWeight
    + distanceTessellationWeight ))*tessellationLevel;
gl_TessLevelOuter[1] = 1 + ((constantTessellationWeight
    + adNormalTessellation * normalTessellationWeight +
    adTextureTessellation * textureTessellationWeight +
    adDistanceTessellation * distanceTessellationWeight
   )/(constantTessellationWeight +
   normalTessellationWeight + textureTessellationWeight
    + distanceTessellationWeight ))*tessellationLevel;
gl_TessLevelOuter[2] = 1 + ((constantTessellationWeight
    + dcNormalTessellation * normalTessellationWeight +
    dcTextureTessellation * textureTessellationWeight +
    dcDistanceTessellation * distanceTessellationWeight
   )/(constantTessellationWeight +
   normalTessellationWeight + textureTessellationWeight
    + distanceTessellationWeight ))*tessellationLevel;
gl_TessLevelOuter[3] = 1 + ((constantTessellationWeight
    + bcNormalTessellation * normalTessellationWeight +
    bcTextureTessellation * textureTessellationWeight +
    bcDistanceTessellation * distanceTessellationWeight
   )/(constantTessellationWeight +
    normalTessellationWeight + textureTessellationWeight
    + distanceTessellationWeight ))*tessellationLevel;


/*
// Tessellation with distance.
gl_TessLevelInner[0] = tessellationLevel *
    innerDistanceTessellation + 1;
gl_TessLevelInner[1] = tessellationLevel *
    innerDistanceTessellation + 1;

gl_TessLevelOuter[0] = tessellationLevel *
    abDistanceTessellation + 1;
gl_TessLevelOuter[1] = tessellationLevel *
    adDistanceTessellation + 1;
gl_TessLevelOuter[2] = tessellationLevel *
    dcDistanceTessellation + 1;
gl_TessLevelOuter[3] = tessellationLevel *
    bcDistanceTessellation + 1;
*/

/*
// Tessellation with Normal.
gl_TessLevelInner[0] = tessellationLevel *
    innerNormalTessellation  + 1;
gl_TessLevelInner[1] = tessellationLevel *
    innerNormalTessellation  + 1;

gl_TessLevelOuter[0] = tessellationLevel *
  abNormalTessellation  + 1;
gl_TessLevelOuter[1] = tessellationLevel *
  adNormalTessellation  + 1;
gl_TessLevelOuter[2] = tessellationLevel *
  dcNormalTessellation  + 1;
gl_TessLevelOuter[3] = tessellationLevel *
  bcNormalTessellation  + 1;
*/

/*
// Tessellation with TessellationMap.
gl_TessLevelInner[0] = tessellationLevel *
  innerTextureTessellation + 1;
gl_TessLevelInner[1] = tessellationLevel *
  innerTextureTessellation + 1;

gl_TessLevelOuter[0] = tessellationLevel *
  abTextureTessellation + 1;
gl_TessLevelOuter[1] = tessellationLevel *
  adTextureTessellation + 1;
gl_TessLevelOuter[2] = tessellationLevel *
  dcTextureTessellation + 1;
gl_TessLevelOuter[3] = tessellationLevel *
  bcTextureTessellation + 1;
*/
```

```

        /*
        // No Tessellation.
        gl_TessLevelInner[0] = tessellationLevel + 1;
        gl_TessLevelInner[1] = tessellationLevel + 1;

        gl_TessLevelOuter[0] = tessellationLevel + 1;
        gl_TessLevelOuter[1] = tessellationLevel + 1;
        gl_TessLevelOuter[2] = tessellationLevel + 1;
        gl_TessLevelOuter[3] = tessellationLevel + 1;
        */
    }
}
```

A.3 Tessellation evaluations shader

```
//
// Tessellation evaluations shader.
//

// Set GLSL version. Minimum version that supports
   tessellation is 4.0.
#version 400

// Decide how you want the patches divided fractional or
   integer.
// Divided by fractional.
layout(quads, fractional_odd_spacing, ccw) in;

// Divided by integer.
//layout(quads) in;

// Gets in vertex position from the tessellation control
   shader.
in vec4 tcPosition[];

// Sends texture coordinate to the geometry shader.
out vec2 textureCoordinate;

// Uniforms from OSG.
uniform sampler2D heightMap;
uniform mat4 mvp;
uniform float scaleXY;
uniform float scaleZ;

//
// Main.
//
void main()
{
  // Gets the position in the ad and the bc vector.
  vec4 adPosition = mix(tcPosition[0], tcPosition[3],
     gl_TessCoord.x);
  vec4 bcPosition = mix(tcPosition[1], tcPosition[2],
     gl_TessCoord.x);

  // Then finds the position between the adPosition and the
      bcPosition point.
  vec4 position = mix(adPosition, bcPosition, gl_TessCoord.
     y);

  // Multiply the found position with scale in the xy-plane
     .
  textureCoordinate = position.xy / scaleXY;

  // Use texture coordinate to find the value on the height
      Map,
  // multiply it with scale (along the Z axis) to find the
     height.
  float height = texture(heightMap, textureCoordinate).r *
     scaleZ;

  // Set the vertex position.
  vec4 positionWithHeight = vec4(position.x, position.y,
     height, 1.0);

  // Sends vertex position multiplied mvp matrix.
  gl_Position = mvp * positionWithHeight;
}
```

A.4 Geometry shader

```
//
// Geometry shader.
//

// Set GLSL version. Minimum version that supports
   tessellation is 4.0.
#version 400

// Gets triangle in.
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

// textureCoordinate from tessellation evaluation shader.
in vec2 textureCoordinate[3];

// Sends normal and texture coordinate to fragment shader.
out vec3 gNormal;
out vec2 gTextureCoordinate;

// Uniforms from OSG.
uniform mat3 normalMatrix;
uniform sampler2D normalMap;

//
// NormalFromTexture takes in the colour value from normal
   map and subtract 0.5,
// since the normal Map zero starts on 0,5, then we get the
    values from - 0,5 to 0,5.
// To the end we normalize the vector.
//
vec3 NormalFromTexture(vec3 normal)
{
  normal.x = normal.x - 0.5;
  normal.y = normal.y - 0.5;
  normal.z = normal.z - 0.5;
  normal = normalMatrix * normalize(normal);
  return normal;
};

//
// Main.
//
void main()
{
  // Get the normal for given vertex position and send
     normal.
  gNormal = NormalFromTexture(texture(normalMap,
     textureCoordinate[0]).rgb);
  gl_Position = gl_in[0].gl_Position;// Sends through
     position.
  gTextureCoordinate = textureCoordinate[0];// Sends
     through textureCoordinate.
  EmitVertex();// Sends this vertex.
```

```

// Get the normal for given vertex position and send
   normal.
gNormal = NormalFromTexture(texture(normalMap,
   textureCoordinate[1]).rgb);
gl_Position = gl_in[1].gl_Position; // Sends through
   position.
gTextureCoordinate = textureCoordinate[1];  // Sends
   through textureCoordinate.
EmitVertex();// Sends this vertex.

// Get the normal for given vertex position and send
   normal.
gNormal = NormalFromTexture(texture(normalMap,
   textureCoordinate[2]).rgb);
gl_Position = gl_in[2].gl_Position; // Sends through
   position.
gTextureCoordinate = textureCoordinate[2];// Sends
   through textureCoordinate.
EmitVertex();// Sends this vertex.

/*
// Can do this in a for-loop, but looses a couple of fps.
for(int i = 0; i<3; i++)
{
  // Get the normal for given vertex position and send
     normal.
```

```
    gNormal = NormalFromTexture(texture(normalMap,
       textureCoordinate[i]).rgb);
    gl_Position = gl_in[i].gl_Position;// Sends through
       position.
    gTextureCoordinate = textureCoordinate[i];// Sends
       through textureCoordinate.
    EmitVertex();// Sends this vertex.
  }
  */

  // Send primitive.
  EndPrimitive();
}
```

A.5 Multi texure fragment shader

```
//
// Fragment shader.
//

// Set GLSL version. Minimum version that supports
   tessellation is 4.0.
#version 400

// Gets in texture coordinate and normal from geometry
   shader.
in vec2 gTextureCoordinate;
in vec3 gNormal;

// Sends out fragment.
out vec4 fragment;

// Light uniforms
uniform vec3 lightPosition;
uniform vec3 diffuseMaterial;
uniform vec3 ambientMaterial;
uniform vec3 specularMaterial;
uniform float shininess;

// Rotation matrix to rotate texture-coordinates 90 degrees
   .
uniform mat4 rotationMatrix;
```

```

// Gets in all textures.
uniform sampler2D diffuse;
uniform sampler2D canyon_rock01;
uniform sampler2D canyon_rock02;
uniform sampler2D dry_grass09;
uniform sampler2D dry_grass12;
uniform sampler2D romanian_sand05;

// Load in the texture masks.
uniform sampler2D VertRock_0_90_mask;
uniform sampler2D VertDetail_0_90_mask;
uniform sampler2D Grass_mask;
uniform sampler2D Flow_mask;

// Ambient-light-map.
uniform sampler2D AmbientOcc;

//
// Main.
//
void main()
{
  // Light calculation with help from the fragment shader
  // in Quad Tessellation with OpenGL 4.0.
  // http://prideout.net/blog/?p=49
  vec3 h = normalize(lightPosition + vec3(0, 0, 1));
```

```
float df = abs(dot(gNormal, lightPosition));
float sf = abs(dot(gNormal, normalize(lightPosition + vec3(0, 0, 1))));
sf = pow(sf, shininess);

// Calculate the light based on the normal and light position.
vec3 light =  texture(AmbientOcc, gTextureCoordinate).rgb + df * diffuseMaterial + sf * specularMaterial;

// Alternative light if we dont have a texture for the ambient light.
// vec3 light = ambientMaterial + df * diffuseMaterial + sf * specularMaterial;


// Diffuse texture.
vec4 diffuseTexture = texture(diffuse, gTextureCoordinate);

// Make a texture coordinate for the rotated images with the rotation matrix.
vec2 textureCoordinate90 = (vec4(gTextureCoordinate,1,1)*rotationMatrix).xy;

// Use VertRock_0_90_mask to mix the texture with the texture, rotated 90 degrees.
```

```
vec4 textureRock = mix(texture(canyon_rock01,
    textureCoordinate90*20),texture(canyon_rock01,
    gTextureCoordinate*20),texture(VertRock_0_90_mask,
    gTextureCoordinate).r);
vec4 textureRock2  = mix(texture(canyon_rock02,
    textureCoordinate90*10),texture(canyon_rock02,
    gTextureCoordinate*10),texture(VertRock_0_90_mask,
    gTextureCoordinate).r);

// Use VertDetail_0_90_mask to mix the texture with the
    texture, rotated 90 degrees.
vec4 texturegrass1  = mix(texture(dry_grass12,
    textureCoordinate90*50),texture(dry_grass12,
    gTextureCoordinate*50),texture(VertDetail_0_90_mask,
    gTextureCoordinate).r);
vec4 texturegrass2  = mix(texture(dry_grass09,
    textureCoordinate90*100),texture(dry_grass09,
    gTextureCoordinate*100),texture(VertDetail_0_90_mask,
    gTextureCoordinate).r);
vec4 texturesand  = mix(texture(romanian_sand05,
    textureCoordinate90*50),texture(romanian_sand05,
    gTextureCoordinate*50),texture(VertDetail_0_90_mask,
    gTextureCoordinate).r);

// Set fragment colour.
vec4 fragmentColour = diffuseTexture;
```

```
  // Mix in rock textures.
  fragmentColour = mix(fragmentColour, textureRock, 0.8);
  fragmentColour = mix(fragmentColour, textureRock2, 0.5);

  // Mix in grass textures with the Grass_mask.
  fragmentColour = mix(fragmentColour, texturegrass1,
     texture(Grass_mask, gTextureCoordinate).r) ;
  fragmentColour = mix(fragmentColour, texturegrass2,
     texture(Grass_mask, gTextureCoordinate).r/2) ;

  // Mix in sand textures with the Flow_mask.
  fragmentColour = mix(fragmentColour,texturesand,texture(
     Flow_mask, gTextureCoordinate).r);



  // Add light.
  fragmentColour *= vec4(light,1);

  // Sends fragment.
  fragment = fragmentColour;
}
```

A.6 Single texture fragment shader

```
//
// Fragment shader.
//

// Set GLSL version. Minimum version that supports
   tessellation is 4.0.
#version 400

// Gets in texture coordinate and normal from geometry
   shader.
in vec2 gTextureCoordinate;
in vec3 gNormal;

// Sends out fragment.
out vec4 fragment;

// Light uniforms
uniform vec3 lightPosition;
uniform vec3 diffuseMaterial;
uniform vec3 ambientMaterial;
uniform vec3 specularMaterial;
uniform float shininess;

// Get in texture.
uniform sampler2D diffuse;

```

```
//
// Main.
//
void main()
{
  // Light calculation with help from the fragment shader
  // in Quad Tessellation with OpenGL 4.0.
  // http://prideout.net/blog/?p=49
  vec3 h = normalize(lightPosition + vec3(0, 0, 1));
  float df = abs(dot(gNormal, lightPosition));
  float sf = abs(dot(gNormal, normalize(lightPosition +
     vec3(0, 0, 1))));
  sf = pow(sf, shininess);

  // Calculate the light based on the normal and light
     position.
   vec3 light = ambientMaterial + df * diffuseMaterial + sf
      * specularMaterial;

  // Diffuse texture.
  vec4 diffuseTexture = texture(diffuse, gTextureCoordinate
     );

  // Set fragment colour.
  vec4 fragmentColour = diffuseTexture;

  // Add light.
```

```
    fragmentColour *= vec4(light,1);

    // Sends fragment.
    fragment = fragmentColour;
}
```