Terrain and water rendering with hardware tessellation

Xavier Bonaventura Brugués

May 20, 2010

Abstract

This thesis studies the hardware tessellator that is a recent addition to the incremental rendering pipeline of graphics processing units, only available using the latest hardware. The main field of application for hardware tessellation is level-of-detail rendering of complex geometries like natural phenomena and procedurally generated features.
This thesis applies this technique in two important fields: the rendering of a large and detailed terrain with different levels of detail depending on the distance, and the simulation and visualization of ocean wave on a large open water surface with physically-based motion and Fresnel reflections.

Contents

1 Introduction **4**
- 1.1 DirectX 11 Graphics Pipeline . . . 5
  - 1.1.1 Hull Shader Stage . . . 5
  - 1.1.2 Tessellator Stage . . . 5
  - 1.1.3 Domain Shader Stage . . . 8

2 Implementation **9**
- 2.1 Previous Work . . . 9
  - 2.1.1 Differences between DirectX 10 and DirectX 11 . . . 9
  - 2.1.2 Effects11 and DXUT . . . 9
- 2.2 Terrain rendering with tessellation . . . 10
  - 2.2.1 Definition of terrain geometry . . . 10
  - 2.2.2 Tessellation Factor . . . 11
  - 2.2.3 Vertex position and texture coordinates . . . 13
  - 2.2.4 Vertex normals . . . 14
  - 2.2.5 Tessellation correction depending on the camera angle . . . 14
  - 2.2.6 Graphic User Interface . . . 16
- 2.3 Ocean rendering with tessellation . . . 18
  - 2.3.1 Definition of ocean geometry . . . 18
  - 2.3.2 Tessellation Factor . . . 18
  - 2.3.3 Vertex position and texture coordinates . . . 20
  - 2.3.4 Vertex normals . . . 20
  - 2.3.5 Ocean texturing . . . 20
  - 2.3.6 Graphic User Interface . . . 21

3 Performance Testing **23**
- 3.1 Different texture sizes . . . 23
- 3.2 Tessellation Factor vs. Number of Patches . . . 23

4 Results **25**

5 Conclusions and future work **30**
- 5.1 Conclusions . . . 30
- 5.2 Future work . . . 30

5.2.1 Terrain and ocean rendering 30
5.2.2 Terrain rendering 31

A Terrain fx file **33**

B Ocean fx file **46**

List of Figures

1.1 DirectX 10 and DirectX 11 Pipelines 6
1.2 Tessellation depending on the factors 7

2.1 Division of the terrain into a grid: Vx and Ex represents vertices and edges respectively in every patch where x is the index to access. Inside 0 and 1 represents the direction of the tessellation inside a patch. 10
2.2 Lines between patches when tessellation factors are wrong 11
2.3 Tessellation factor depending on the distance: (b) is the function used to get the final tessellation with round to the exponent, (a) is the function of the exponent used to calculate (b). 12
2.4 Mipmap level depending on the camera distance 14
2.5 Rendering the terrain normals 15
2.6 Normals calculation 15
2.7 Terrain rendering with different options 17
2.8 Tessellation factor depending on the distance: (b) is the function used to get the final tessellation factor, (a) is the function of the exponent used to calculate (b). 19
2.9 Rendering the ocean normals 21
2.10 Ocean rendering with and without bump-mapped ripples 22
2.11 Ocean rendering with different options 22

4.1 Ocean rendering without angle correction 26
4.2 Ocean rendering with angle correction 26
4.3 Terrain rendering without angle correction 27
4.4 Terrain rendering with angle correction 27
4.5 Ocean rendering: Min Tess Factor 2, Max Tess Factor 64 28
4.6 Ocean rendering: Min Tess Factor 4, Max Tess Factor 16 28
4.7 Terrain rendering: Min Tess Factor 2, Max Tess Factor 64 29
4.8 Terrain rendering: Min Tess Factor 4, Max Tess Factor 16 29

Chapter 1

Introduction

Nowadays, one of the biggest problems in computer graphics is the detail into scenes. To get more realistic scenes you need high details models but then the computer goes slow. To increase the number of frames per second you can use low detail models but then it doesn't seems realistic. The solution is to combine high models near to the camera and low models far to the camera but this is not easy.
One of the most used technique consist of a set of different detail models and in the runtime change it depending on the distance of the camera. This process it's done into the CPU and this is a problem because the CPU it's not intended for this work and you waste a lot of time sending meshes from the CPU to the GPU.
In DirectX 10[1] you could change the detail of meshes into the GPU doing tessellation into the geometry shader but this shader is not intended for this kind of jobs and it's not really the best solution. The output of the geometry shader is limited and this work is serial.
The best solution to tessellate is the recently appear tessellator stage in DirectX 11. This stage together with two more allows to the programmer tessellate very quickly into the GPU. With this way you can send low level detail meshes to the GPU and generate the missing geometry to the GPU depending on the camera distance, angle or whatever you want.
First of all in the section 1.1 we will take a look to the new stages in DirectX 11 and how they work. Then into the chapter 2 we will explain the implementation of water and terrain rendering with these tools. Into the chapter 3 we analyze the performance using different size of textures and changing the relation between the size of the mesh and the tessellation factor. Chapter 4 will show the results and chapter 5 the conclusions and future works.

1.1 DirectX 11 Graphics Pipeline

The DirectX 11 graphics pipeline[2] add three new stages in the DirectX 10 which are hull shader stage, tessellator stage and domain shader stage (Figure 1.1). The first and third are programmable and the second one is configurable. These are after the vertex shader and before the geometry shader and they are intended to do tessellation into the graphic card.

1.1.1 Hull Shader Stage

The hull shader stage is the first part into the tessellation bloc and it goes after the vertex shader stage and before the tessellator stage. The data used in it is new in DirectX 11 and it uses a new primitive topology called control point patch list. As its name suggests it represents a collection of control points where the number in every patch can go from 1 to 32. These control points are responsible to define the mesh.
The output data in this stage is composed of two parts, one are the input control points that can be modified and the other are some constant data that will be used into the tessellator and domain shader stages.
To calculate the output data there are two functions, the first one is executed for every patch and there you can calculate the tessellation factor for every edge of the patch and inside it.
The other part is executed for every control point into the patch and there you can manipulate this control point. In both parts you have the information of all control points into the patch, in addition, in the second part you have the id of the control point that you are processing.
Example of hull shader headers:

```
HS_CONSTANT_DATA_OUTPUT TerrainConstantHS( InputPatch<
    VS_CONTROL_POINT_OUTPUT, INPUT_PATCH_SIZE> ip, uint
    PatchID : SV_PrimitiveID )
HS_OUTPUT hsTerrain( InputPatch<VS_CONTROL_POINT_OUTPUT,
    INPUT_PATCH_SIZE> p, uint i : SV_OutputControlPointID,
    uint PatchID : SV_PrimitiveID )
```

1.1.2 Tessellator Stage

The tessellator stage is a part of the new pipeline where the programmer only can change the behaviour setting some values. It goes after the hull shader stage and before the domain shader stage. It's executed once per patch and the input data are the control points and the tessellation factors which are the output from the hull shader. This stage is responsible to divide a quad, triangle or line in a lot of them depending on the tessellation factor and the type of partitioning defined (Figure 1.2). The output is UV coordinates which goes from 0 to 1 and they define the position of new vertices relative into the patch.

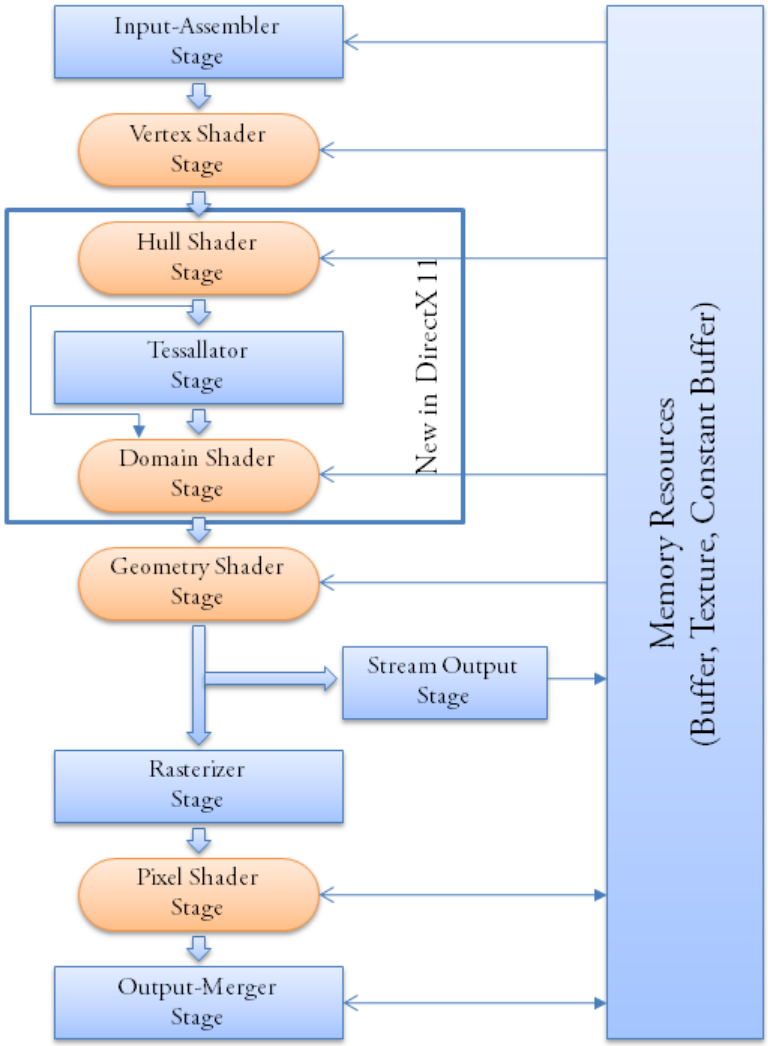


Figure 1.1: DirectX 10 and DirectX 11 Pipelines

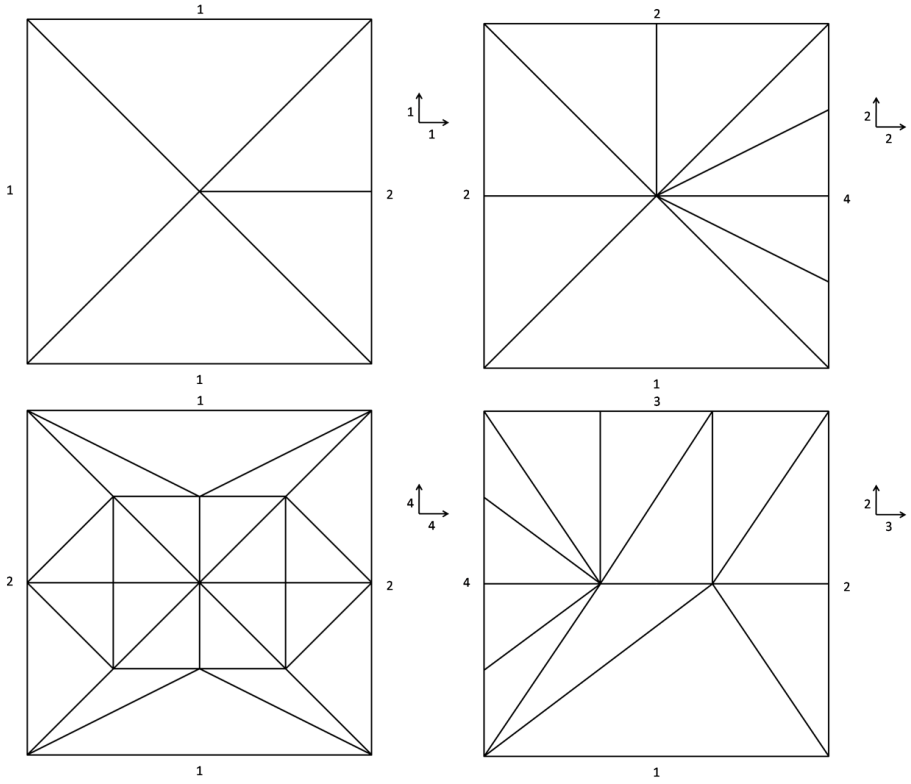


Figure 1.2: Tessellation depending on the factors

1.1.3 Domain Shader Stage

The domain shader stage is the last part to do tessellation and it goes before the geometry shader stage and after the tessellator stage. This part is executed once per UV coordinate generated into the tessellator stage and it access to the information from the hull shader stage output (control points and constant data) and UV coordinates. In this final stage the aim is to calculate the final position of every vertex generated and all the associated information as normal, colour, texture coordinate, etc.
Example of domain shader header:

```
DS_OUTPUT dsTerrain( HS_CONSTANT_DATA_OUTPUT input, float2 UV
    : SV_DomainLocation, const OutputPatch<HS_OUTPUT,
    OUTPUT_PATCH_SIZE> patch )
```

Chapter 2

Implementation

2.1 Previous Work

Before start to implementing the tessellation one of the most important works to do is to configure a DirectX 11 project. DirectX 10 and DirectX 11 have some differences and we have taken a DirectX 10 project with some basic features and we have converted it to DirectX11. This project include draw a mesh with index buffer, the application of textures to a mesh, draw a mesh with instancing, how to use a static environment map, creation of a simple terrain and geometry generation through the geometry shader.

2.1.1 Differences between DirectX 10 and DirectX 11

The main change in DirectX 11 has been that the functionalities of the device have been divided in two parts, the device and the device context.
The device is responsible to create resources and to enumerate the capabilities of a display adapter. The device context includes the other parts like rendering operations, map/unmap resources and operations to set the pipeline state.

2.1.2 Effects11 and DXUT

Into the development of this application has been used the Effects11 and DXUT frameworks to reduce the work that it's not specific of hardware tessellation. These two frameworks are used into the DirectX sample and the aim is to create a project quickly with the general default values.
With these frameworks has been possible to devote most of the time implementing shaders and not the skeleton of the application.

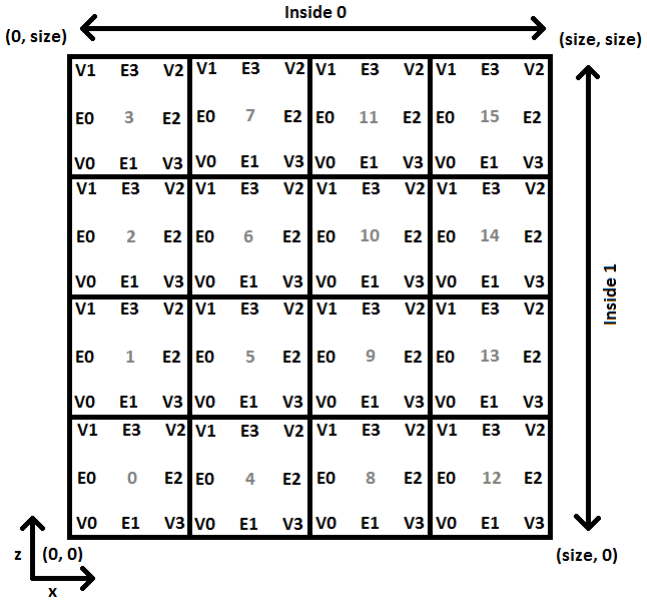


Figure 2.1: Division of the terrain into a grid: Vx and Ex represents vertices and edges respectively in every patch where x is the index to access. Inside 0 and 1 represents the direction of the tessellation inside a patch.

2.2 Terrain rendering with tessellation

Terrain rendering is a good field to apply tessellation because usually terrains are huge and to render all with the same detail when it's high it's impossible. The problem with the terrain is that usually the mesh is big and you waste a lot of time sending the information of the mesh from the CPU to the GPU. Hardware tessellation technique allows to send to the GPU practically all the information into the initialization part and then you can change the level of detail in it with a minimum interaction with the CPU.

2.2.1 Definition of terrain geometry

To tessellate the terrain we need to divide it in different patches. It can be applied to a lot of shapes but it will be used the most intuitive shape, a patch with four points. We will divide the terrain in patches of the same size like a grid (Figure 2.1). For every patch we will have to decide the tessellation factor in every edge and inside, it's very important that two patches that share the same edge they had the same tessellation factor in this one because if it doesn't happens then you will see some line between patches (Figure 2.2).

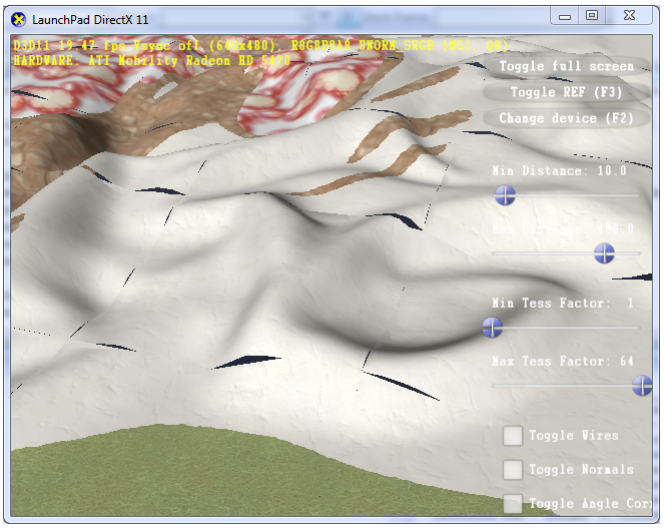


Figure 2.2: Lines between patches when tessellation factors are wrong

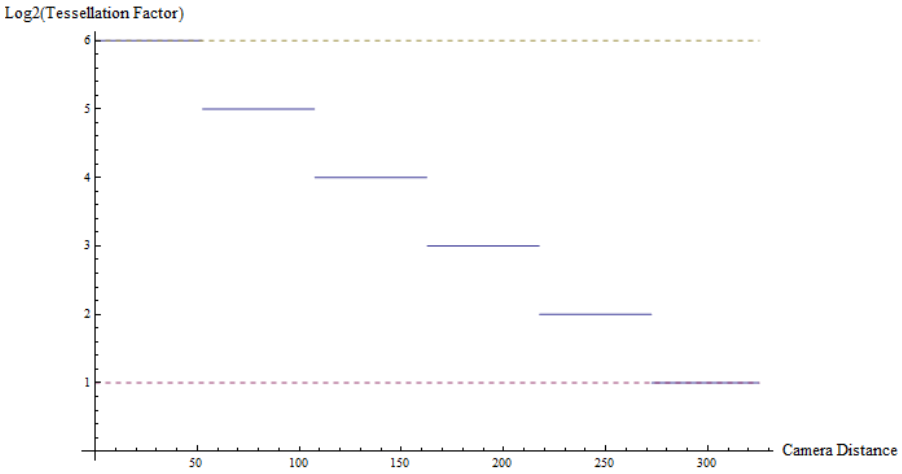
2.2.2 Tessellation Factor

In the tessellator stage you can define different kinds of tessellations (fractional_even, fractional_odd, integer or pow2). We will use the integer but in addition we will impose one restriction more, this value must be a power of two to avoid a wave effect. With this way when a new vertex appear it will be there until the tessellation factor decrease again and their x and z values will not change. The only value that will change will be the y coordinate to avoid popping.
First of all we have to define two distances, one the minimum and the other the maximum, to calculate the tessellation factor. These distances will be where the tessellation factor will be maximum and minimum respectively. In addition we have to define which will be the maximum and the minimum of tessellation factor; maybe in some cases we don't want that the minimum of tessellation factor to be 1 if there are few patches or 64 if the computer goes to slow.
Then the tessellation factor will be 2^x where x will be a number whose range will be from $\log_2 MaxTessellation$ to $\log_2 MinTessellation$ linear interpolated between the minimum and the maximum distance. This x will be round to the nearest integer to get a power of two tessellation factor (Figure 2.3).

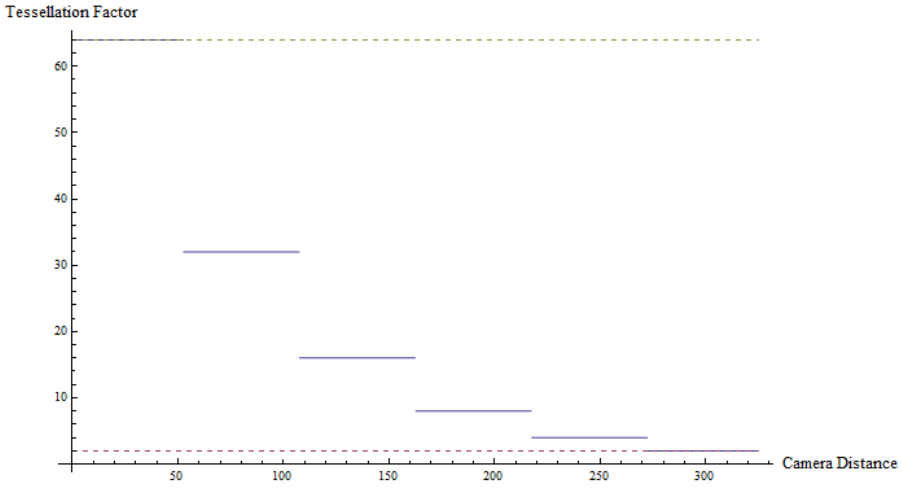
(a)

$$te_a(d) = \begin{cases} \max(te), & \text{for } d \leq \min(d), \\ \text{round}(\text{diff}(te)(1 - \frac{d-\min(d)}{\text{diff}(d)}) + \min(te)), & \text{for } \min(d) < d < \max(d), \\ \min(te), & \text{for } d \geq \max(d). \end{cases}$$

where $\text{diff}(x) = \max(x) - \min(x)$



(a) Log2 of tessellation factor with round



(b) Tessellation factor with round in it Log2 to get a power of 2 value

Figure 2.3: Tessellation factor depending on the distance: (b) is the function used to get the final tessellation with round to the exponent, (a) is the function of the exponent used to calculate (b).

(b)

$$te_b(d) = \begin{cases} 2^{\max(te)}, & \text{for } d \leq \min(d), \\ 2^{\text{round}(\text{diff}(te)(1-\frac{d-\min(d)}{\text{diff}(d)})+\min(te))}, & \text{for } \min(d) < d < \max(d), \\ 2^{\min(te)}, & \text{for } d \geq \max(d). \end{cases}$$

where $\text{diff}(x) = \max(x) - \min(x)$

As we said before it's very important that two patches that share the same edge they have the same tessellation factor in this edge. To do this we will calculate five different distances in every patch, one for every edge and one for inside. To calculate the tessellation factor for every edge we calculate the distance between the camera and the central point of the edge. With this way in two adjacent patches with the same edge the distance in the middle point will be the same because they share the two vertices that we use to calculate the middle point. To calculate the tessellation factor inside the patch in U and V direction we calculate the distance between the camera position and the middle point of the patch.

2.2.3 Vertex position and texture coordinates

In every result point we can easy calculate the x and z coordinates with a single interpolation between the position of the vertices of the patch but we also need the y coordinate that represents the height of the terrain in every point and the texture coordinates. As we have defined the terrain we only have to take the x and z final coordinates and divide by the size of the terrain. Once we have the texture coordinates to get the height of the terrain we read the information from a heightmap texture. To take this information we have to use mip-map levels or we will see some popping when new vertices appear. To reduce this popping the goal is to get the value from a texture where the concentration of points in it are the same than the concentration in the area where we have the vertex. Four patches that share a vertex have to use the same mip-map level in that vertex to be coherent, for this reason we calculate one mip-map level for every vertex in a patch. Then, to calculate the mip-map level into the other vertices we only have to interpolate between the mip-map level of the vertices of the patch.
The next formula is used to calculate the mip-map level in every vertex of the patch:

$$MipMap(d) = \begin{cases} \min(MipMap), & \text{for } d \leq \min(d), \\ \text{diff}(MipMap)\frac{d-\min(d)}{\text{diff}(d)} + \min(MipMap), & \text{for } \min(d) < d < \max(d), \\ \max(MipMap), & \text{for } d \geq \max(d). \end{cases}$$

where $\text{diff}(x) = \max(x) - \min(x)$
To calculate the minimum and the maximum value for the mip-map level variable we use these equations:

$$\min(MipmapLevel) = \log_2(textSize) - \log_2(sqrtNumPatch2^{\max(te)})$$

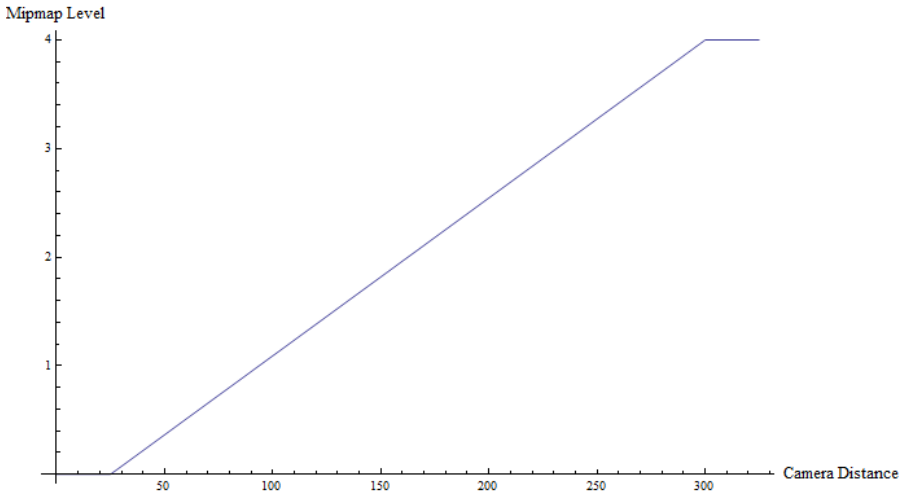


Figure 2.4: Mipmap level depending on the camera distance

$$\max(MipmapLevel) = \log_2(textSize) - \log_2(sqrtNumPatch 2^{\min(te)})$$

If the minimum value is less than 0 we take 0.

2.2.4 Vertex normals

In every vertex we have to calculate the normal vector to apply ilumination techniques later (Figure 2.5). This is read from a texture with the same size of the height map that stores normal vectors in world coordinates. To access to this texture we read from the same mip-map level that we read the height of the terrain.
These normals are calculated previously with the same way that if we want to calculate the normal of a vertex shared by 8 faces. In addition we have to use a weighted mean because the weight of the diagonal vectors is $\frac{1}{\sqrt{2}}$ times shorter than the vertical and horizontal vectors (Figure 2.6).

$$Normal(C) = \frac{\frac{\overrightarrow{C0}\times\overrightarrow{C2}+\overrightarrow{C4}\times\overrightarrow{C6}}{2} + \frac{1}{\sqrt{2}}\frac{\overrightarrow{C1}\times\overrightarrow{C3}+\overrightarrow{C5}\times\overrightarrow{C7}}{2}}{1+\frac{1}{\sqrt{2}}}$$

2.2.5 Tessellation correction depending on the camera angle

Until now we have assume that the tessellation factor only depends on the distance of the camera, nevertheless it's not the same if you see a patch from one direction or from another. For this reason we apply a correction to the tessellation factor that we have calculated previously. This correction will not be applied to the final tessellation factor because we want that the final one to

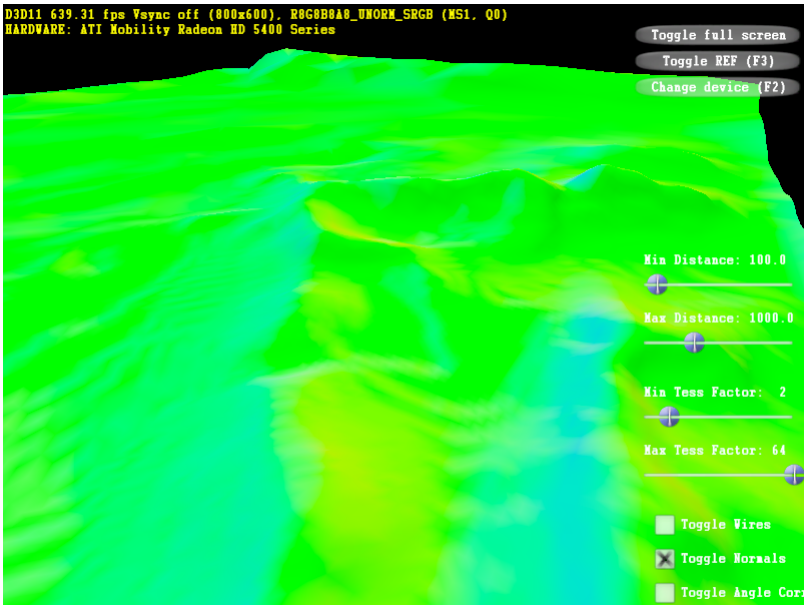


Figure 2.5: Rendering the terrain normals

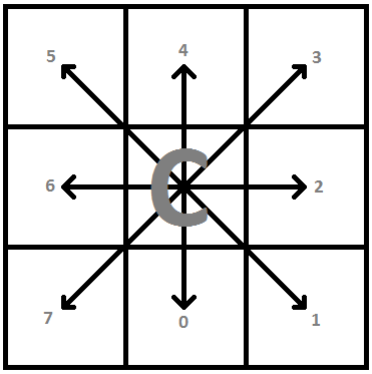


Figure 2.6: Normals calculation

be a power of two value, this correction will be applied to the value x before rounded that we use in 2^x to calculate the final tessellation factor.
The angle to calculate this correction will be the angle between the unit vector over an edge and the unit vector that goes from the middle of this edge to the camera.
To calculate the correction we will use this formula:

$$\frac{\frac{\pi}{2} - \arccos\left(|\widehat{c} \cdot \widehat{e}|\right)}{\frac{\pi}{2}} rank + 1 - \frac{rank}{2}$$

The rank value is used to decide how important is the angle in comparison with the distance is. If you decide a rank of 0.4 then the tessellation factor will be multiplied by a value between 0.8 and 1.2.

$$\frac{\frac{\pi}{2} - \frac{\arccos\left(|\widehat{c} \cdot \widehat{e1}|\right) + \arccos\left(|\widehat{c} \cdot \widehat{e2}|\right)}{2}}{\frac{\pi}{2}} rank + 1 - \frac{rank}{2}$$

To be coherent with this modification we have to apply the correction to the value that we use to access to the mipmap level in a texture. It is very important to keep the coherence that four patches that share the same vertex they have the same mipmap value in that vertex. To calculate the angle of the camera in this point we calculate the mean of the angles between the camera and every vector over the edges that share the point.

$$\frac{\frac{\pi}{2} - \frac{\arccos\left(|\widehat{c} \cdot \widehat{v0}|\right) + \arccos\left(|\widehat{c} \cdot \widehat{v1}|\right) + \arccos\left(|\widehat{c} \cdot \widehat{v2}|\right) + \arccos\left(|\widehat{c} \cdot \widehat{v3}|\right)}{4}}{\frac{\pi}{2}} rank + 1 - \frac{rank}{2}$$

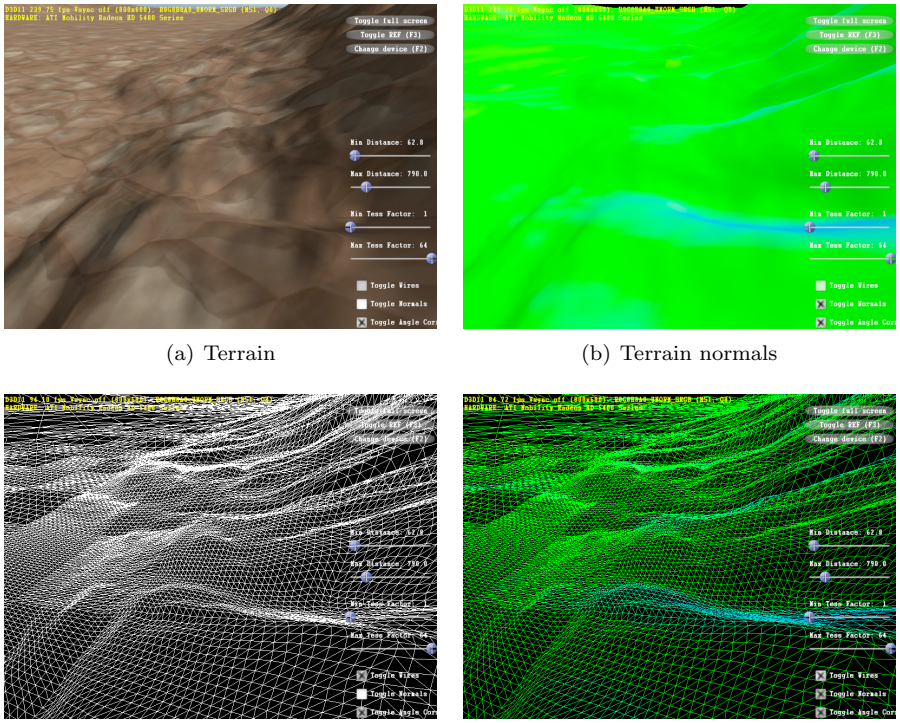
Into the hull shader we don't know the information about the vertex of the other patches but this vectors can be calculated to the vertex shader because we know the size of the terrain and the number of patches.

2.2.6 Graphic User Interface

As we have seen before we have a lot of parameters to adjust in this terrain implementation. For this reason the user can change these values through a graphic user interface.
With this interface we can change some value to adjust the rendering:

- Min Distance: The distance where the tessellation factor will be maximum
- Max Distance: The distance where the tessellation factor will be minimum
- Min Tess Factor: The minimum value for the tessellation factor
- Max Tess Factor: The maximum value for the tessellation factor

In addition the user can see the terrain normals and a wire frame terrain to see the size of every face (Figure 2.7). Finally, the user can decide to apply or not the angle correction to see the difference.



(a) Terrain (b) Terrain normals

(c) Wired terrain (d) Wired terrain normals

Figure 2.7: Terrain rendering with different options

2.3 Ocean rendering with tessellation

Now we have seen an application of hardware tessellation in a terrain rendering but it's not the best field of application because we don't have to send a huge mesh to the GPU but we have to send a huge heightmap. Into the next part we will see the application of hardware tessellation in a ocean rendering, we don't need a heightmap, we just only need some functions and a simple mesh.

2.3.1 Definition of ocean geometry

With the ocean rendering we also have to define a basic ocean geometry where will apply the tessellation depending on the distance and the angle of the camera. This will be exactly the same than in the terrain rendering in subsection 2.2.1.

2.3.2 Tessellation Factor

The tessellation factor used in ocean rendering is a little different than in the terrain rendering. There was a problem that if we didn't use a power of two tessellation factor into the terrain rendering we could see a wave effect because the x and z coordinates into the final vertices moved when we changed the tessellation factor.
This happened because the terrain is a static mesh but now we are rendering a ocean were the waves changes with the time. If the wave effect is smooth this will not be a problem because indeed we are rendering waves.
For this reason we will use a fractional_even tessellation that can take a value from 2 to 64 and as it name suggests it can be a fractional value. The way to calculate this value will be the same than in the terrain rendering, it will depend on the distance and the angle of the camera, but we will not round the exponent.

(a)

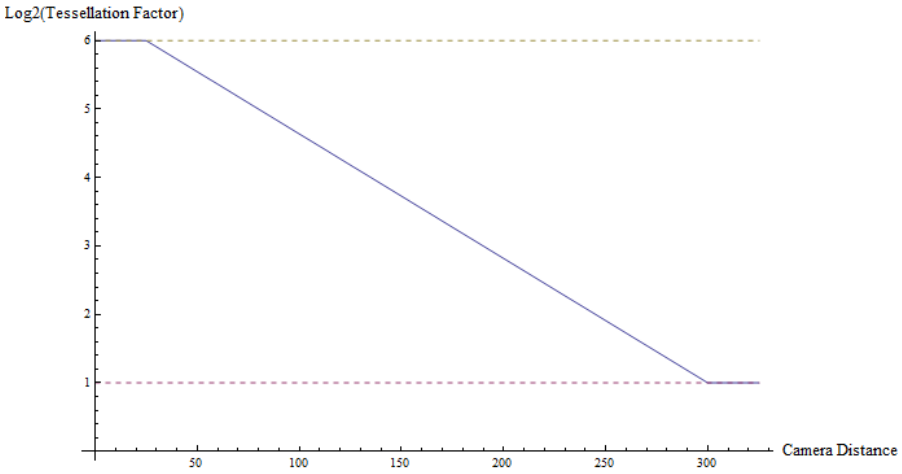
$$te_a(d) = \begin{cases} \max(te), & \text{for } d \leq \min(d), \\ \text{diff}(te)(1 - \frac{d - \min(d)}{\text{diff}(d)}) + \min(te), & \text{for } \min(d) < d < \max(d), \\ \min(te), & \text{for } d \geq \max(d). \end{cases}$$

where $\text{diff}(x) = \max(x) - \min(x)$

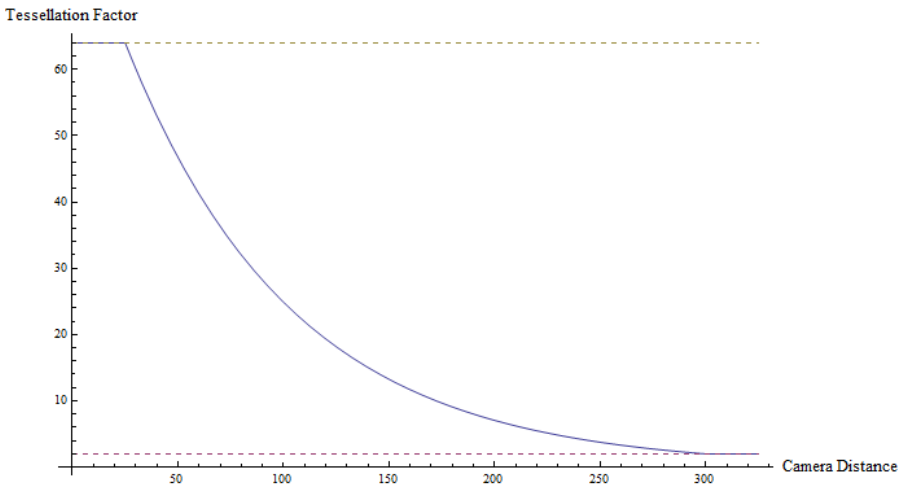
(b)

$$te_b(d) = \begin{cases} 2^{\max(te)}, & \text{for } d \leq \min(d), \\ 2^{\text{diff}(te)(1 - \frac{d - \min(d)}{\text{diff}(d)}) + \min(te)}, & \text{for } \min(d) < d < \max(d), \\ 2^{\min(te)}, & \text{for } d \geq \max(d). \end{cases}$$

where $\text{diff}(x) = \max(x) - \min(x)$



(a) Log2 of tessellation factor without round



(b) Tessellation factor without round in it Log2

Figure 2.8: Tessellation factor depending on the distance: (b) is the function used to get the final tessellation factor, (a) is the function of the exponent used to calculate (b).

2.3.3 Vertex position and texture coordinates

To calculate the final vertex position in the ocean rendering it's not so easy like in the terrain rendering. Now we don't have a heightmap and we have to calculate the final position depending on the waves and the position into the world. To get a real motion we will use the technique explained in ShaderX 6 developed by Szécsi and Arman[3].
First of all we have to imagine a single wave with a wavelength (λ), an amplitude (a) and a direction (k).
It velocity (v) can be represented with this function:

$$v = \sqrt{\frac{g\lambda}{2\pi}}$$

Then the phase (φ) at time (t) in a point (p) is:

$$\varphi = \frac{2\pi}{\lambda}(p \cdot k + vt)$$

Finally the displacement (s) to apply in that point is:

$$s = a[-\cos\varphi, \sin\varphi]$$

A ocean it's not a simple wave and we have to combine all of them to get a realistic motion:

$$p_\Sigma = p + \sum_{i=0}^{n} s(p, a_i, \lambda_i, k_i)$$

2.3.4 Vertex normals

In every vertex we need to calculate the normal (Figure 2.9) but now it's not necessary to access to a normal texture because it can be calculated by a formula. We have a parametric function where the parameters are the x and z coordinates that we have used to calculate the position, if we do the cross product of the two partial derivatives then we get the normal vector in that point.

$$N = \frac{\partial p_\Sigma}{\partial z} \times \frac{\partial p_\Sigma}{\partial x}$$

2.3.5 Ocean texturing

For texturing the ocean we have combined two techniques. The first one is Fresnel reflections to get a realistic colour of the water and the second one is the addition of some noise to the normals to get the effect when the wind creates little waves over the surface.
Fresnel reflections ($F(\theta)$) is used to combine the colour of the water (c_{water}) and the colour of the environment (c_{env}) to get the final colour depending on

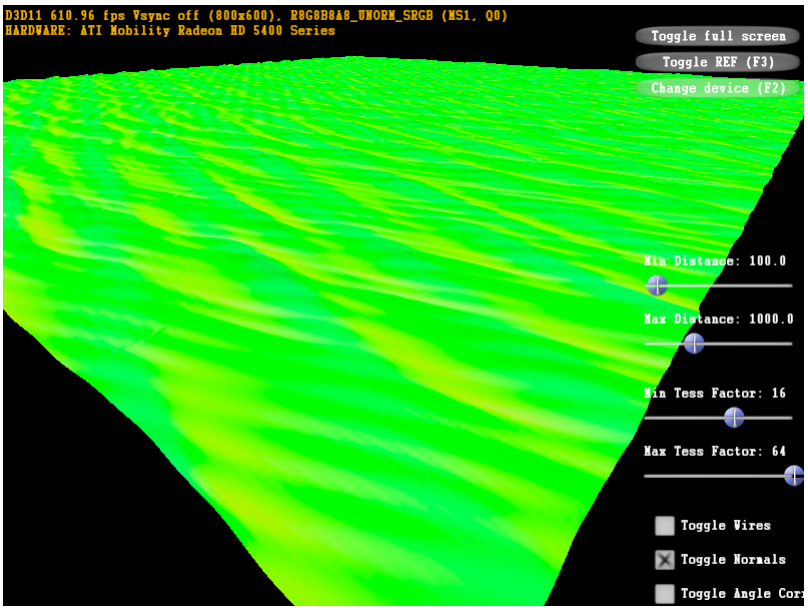


Figure 2.9: Rendering the ocean normals

the view angle of the camera (θ). In addition the colour of the water is a linear combination between the colour of the deep water (c_{deep}) and the colour of the shallow water ($c_{shallow}$).

$$F(\theta) = 0.02 + 0.98(1 - \cos\theta)^5$$

$$c_{water} = c_{deep}\cos\theta + c_{shallow}(1 - \cos\theta)$$

$$c = c_{env}F(\theta) + c_{water}(1 - F(\theta))$$

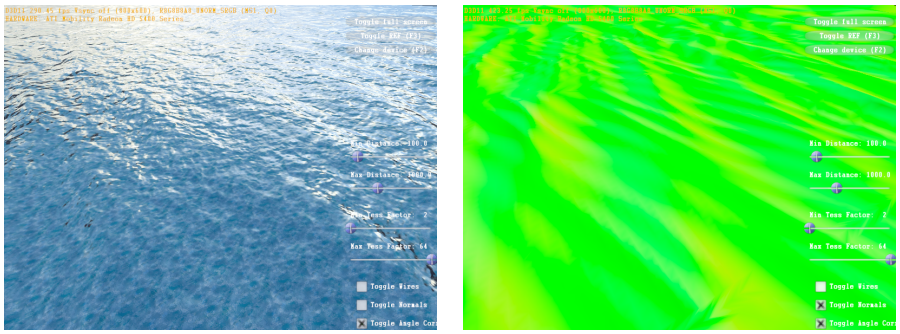
The other technique also used in the implementation of Szécsi and Arman[3] is the bump-mapped ripples. It consists of the same than a bump-mapping[4] but it's animated and with different periods to get chaotic ripples (Figure 2.10).

2.3.6 Graphic User Interface

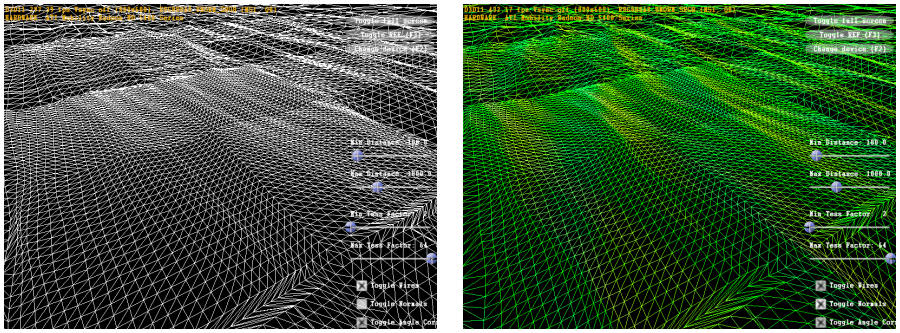
The graphic user interface used in the ocean rendering is the same than in the terrain rendering in subsection 2.2.6. The only differences are that now the minimum tessellation factors can not be less than 2 and the value can be decimal because we use the fractional_even tessellation factor and it rank is from 2 to 64 and in decimals (Figure 2.11).

(a) Ocean normals (b) Wired ocean normals

Figure 2.10: Ocean rendering with and without bump-mapped ripples



(a) Ocean (b) Ocean normals



(c) Wired ocean (d) Wired ocean normals

Figure 2.11: Ocean rendering with different options

Chapter 3

Performance Testing

We have developed two similar applications to understand how tessellation works but we have to release some performance testing to know their weakness. The first test is about the size of textures used into terrain rendering to know the height, the second one is about the relation between the tessellation factor and the number of patches used.
These tests has been released in a Easynote TJ76 (Intel Core i5 processor 430M, ATI Mobility Radeon HD 5470 Graphics 512MB and 4GB Memory). Every execution in a test has been done with a resolution of 1366x768 during 60 seconds and then we have taken the minimum, maximum and average frames per second.

3.1 Different texture sizes

The terrain rendering uses textures, one for the height and another one for the normals. The size of this texture have to be enough big to have all the information when the tessellation factor is high. If this texture it's not enough big we have to interpolate linearly the value of two texture pixels to know the final height.
In this test we can see that if the terrain texture is too big then the performance drops. The values used in the test are: minimum distance (100), maximum distance (1000), minimum tessellation factor (1) and maximum tessellation factor (64).

Texture size	Min. FPS	Max. FPS	Avg. FPS
1K x 1K	354	387	380.300
2K x 2K	297	324	320.267
4K x 4K	90	108	102.800

3.2 Tessellation Factor vs. Number of Patches

Another thing that we have to take care of is about the relation between the tessellation factor and the number of patches. If we tessellate all the terrain with

the same tessellation factor it's the same in these samples if we use 16 patches and a tessellation factor of 32 than if we use 4 patches and a tessellation factor of 64.
With these tests we want to prove if in our examples is better to use the maximum power of tessellator or if it's better to use less power and more patches. In both examples we have adjust the minimum and maximum tessellation factor to the same value to get a regular tessellation.
Test with terrain:

$\sqrt{\text{Number of patches}}$	Tess. Factor	Min. FPS	Max. FPS	Avg. FPS
4	64	299	325	321.150
8	32	293	315	308.617
16	16	288	310	306.400
32	8	279	302	298.150
64	4	243	262	260.617

Test with ocean:

$\sqrt{\text{Number of patches}}$	Tess. Factor	Min. FPS	Max. FPS	Avg. FPS
4	64	166	175	171.367
8	32	162	171	169.217
16	16	127	133	131.033
32	8	126	136	134.250
64	4	127	137	133.350

In both examples we can see that it's better to use the maximum power of tessellator than use less power and more patches.

Chapter 4

Results

With these images we will show you the results in the terrain and ocean rendering and the difference if we apply the angle correction or if we change the tessellation factor.
In the figure 4.1 you can see the ocean rendering if we don't take care of about the angle of the camera. You can see that quads consist of two triangles the size of their edges projected to the screen are different. On the other hand if you see the figure 4.2 where the correction angle is applied then these quads seems to be the same size to the edges. In the figures 4.3 and 4.4 you can see the same effect with the terrain.
In the figures 4.5 and 4.7 you can see the ocean and terrain rendering with a minimum tessellation factor of 2 and a maximum tessellation factor of 64. In the figures 4.6 and 4.8 you can see the same image but with a higher minimum tessellation factor (4) and a lower maximum tessellation factor (16). It can be seen that in the second case the near detail is higher but the farther detail is lower, nevertheless the difference is more noticeable in the terrain rendering than in the ocean rendering because the highest geometry is more irregular.

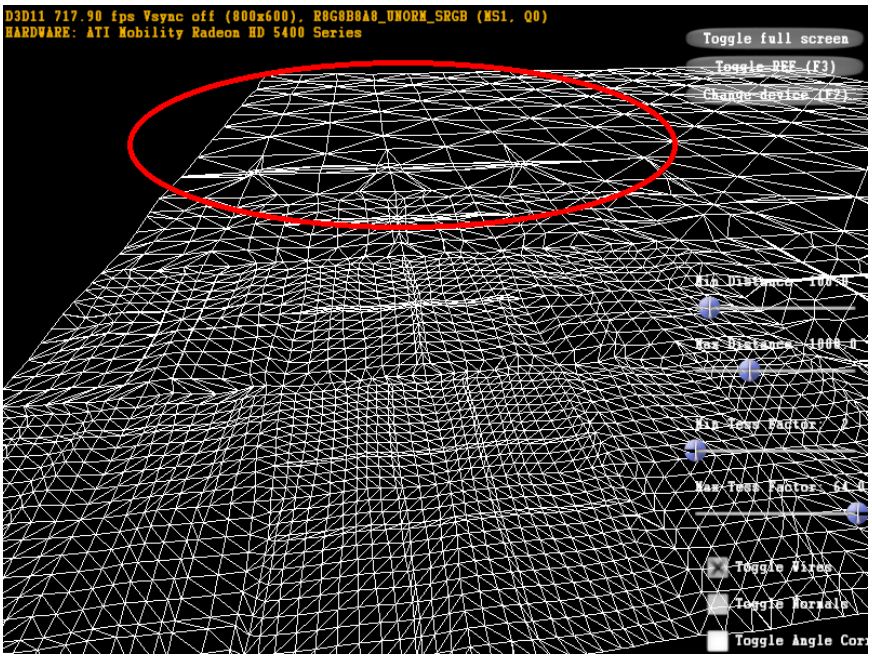


Figure 4.1: Ocean rendering without angle correction

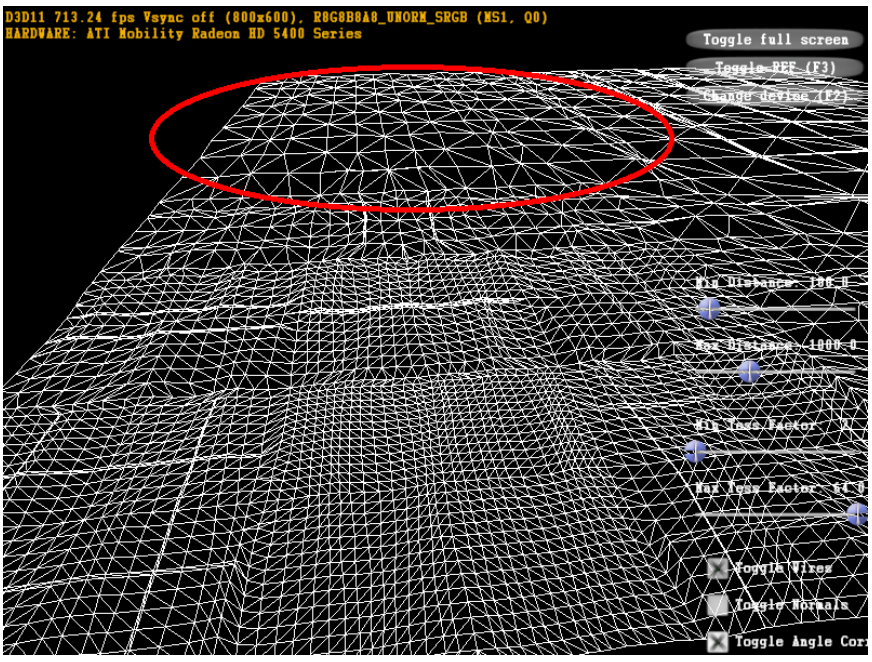


Figure 4.2: Ocean rendering with angle correction

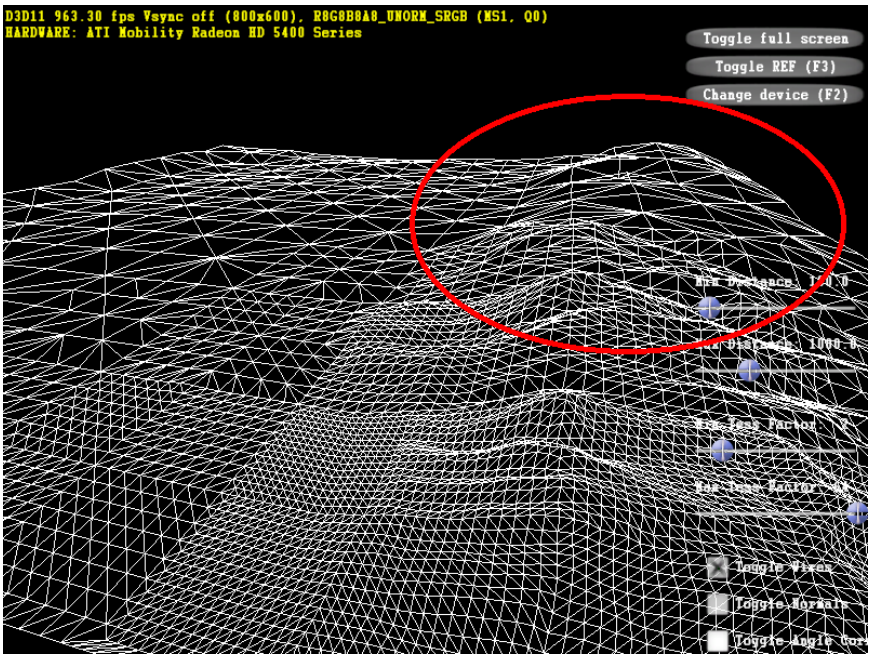


Figure 4.3: Terrain rendering without angle correction

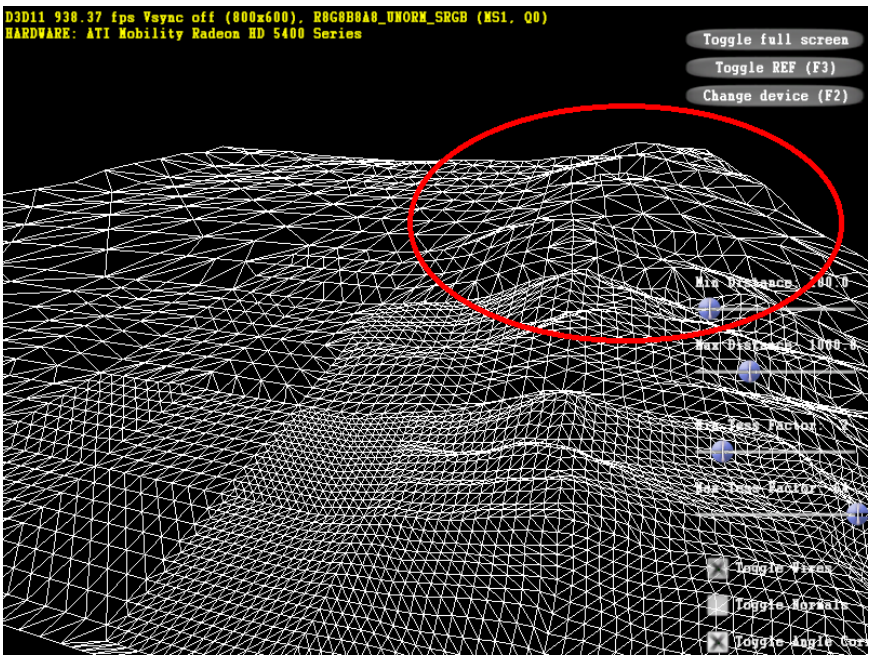


Figure 4.4: Terrain rendering with angle correction

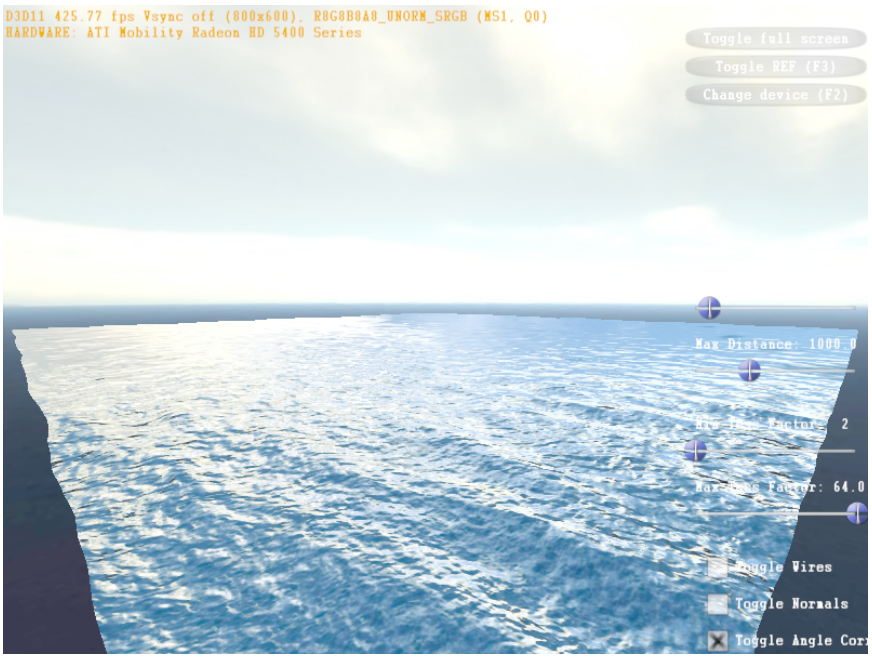


Figure 4.5: Ocean rendering: Min Tess Factor 2, Max Tess Factor 64

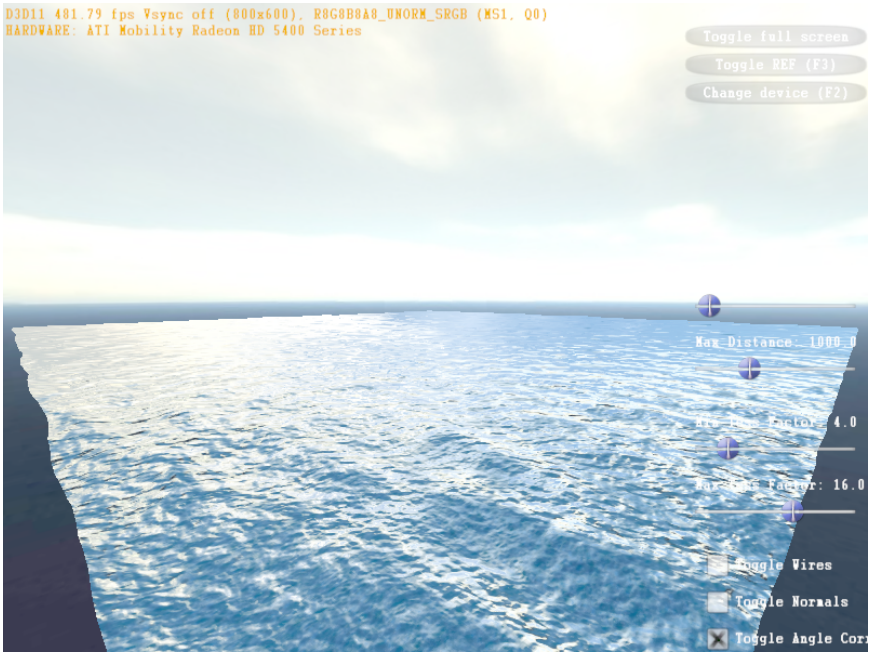


Figure 4.6: Ocean rendering: Min Tess Factor 4, Max Tess Factor 16

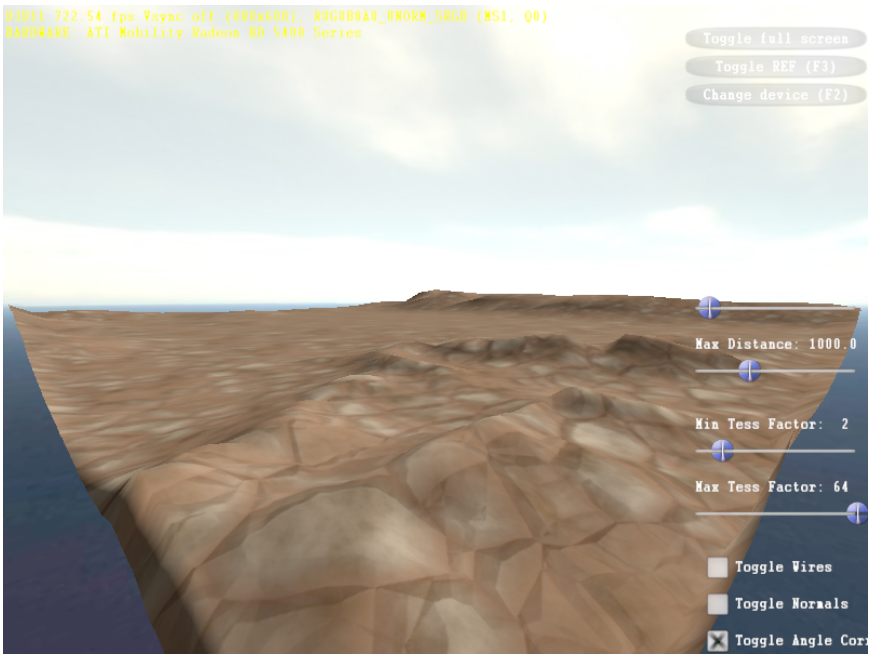


Figure 4.7: Terrain rendering: Min Tess Factor 2, Max Tess Factor 64

Figure 4.8: Terrain rendering: Min Tess Factor 4, Max Tess Factor 16

Chapter 5

Conclusions and future work

5.1 Conclusions

As we seen in this thesis, hardware tessellation it's a powerful tool that reduce the information transfer from the CPU to the GPU. It adds three new stages to the graphic pipeline and this allows a great flexibility to take advantage of hardware tessellation. We have seen the application in two fields, terrain and water rendering, but it can be used to others like other meshes.
The main point that we have to bear in mind is that we can use other techniques to calculate the tessellation factor but we always have to be coherent with tessellation factor and mipmap levels with all the patches to avoid the lines between them that we have seen.
In addition we have seen another important thing, it is better if we can use functions to represent the mesh like the ocean because then the resolution can be as high as the tessellation factor sets. If we use a heightmap like in the terrain then can be possible that we haven't enough information into the texture and we have to interpolate between texels. In addition we have seen that if the texture is too high then the performance falls.

5.2 Future work

In this thesis we have done a first rapprochement to hardware tessellation but this work can be improved or it can lead to further research. Here we give some ideas about future work but we have divided in two parts, the first one can be applied to two examples, the second one can be applied to the terrain.

5.2.1 Terrain and ocean rendering

Improvements for the terrain and ocean rendering:

- Tessellate tri-patches instead of quad-patches to see if it's more efficient or more realistic.
- Decide the tessellation factor depending on the size of the edges projected to the screen instead of the distance and the angle of the camera to see if it's more realistic or efficient.
- Define the geometry with vertex2 indeed vertex3 to avoid information because we don't use the y coordinate from the patch.
- Apply a view frustum culling.

5.2.2 Terrain rendering

Improvements for the terrain rendering:

- Put the terrain and normal texture in only one to be more efficient, three channels for the normals and one channel for the heights.
- Apply some visibility techniques to avoid rendering some parts of the terrain that are not visible because another higher mountain is in front of it.
- Decide the tessellation factor depending on the entropy of the heightmap points.

Bibliography

[1] Wendy Jones. *Beginning DirectX 10 Game Programming*. Course Technology Press, Boston, MA, United States, 2007.

[2] Microsoft, 2009. Windows DirectX Grapics Documentation (August 2009).

[3] László Szécsi and Khashayar Arman. Procedural ocean effects. In *ShaderX*^{6}*: Advanced Rendering Techniques*. Charles River Media, 2008.

[4] L. Szirmay-Kalos and T. Umenhoffer. Displacement mapping on the GPU - State of the Art. *Computer Graphics Forum*, 27(1), 2008.

Appendix A

Terrain fx file

```
//--------------------------------------------------------------------------------------
// File: terrain.fx
//
// This file shows an implementation of the DirectX 11
// Hardware Tessellator for rendering a terrain.
//
// Xavi Bonaventura Brugus
//--------------------------------------------------------------------------------------

// This allows us to compile the shader with a #define to
// choose the different partition modes for the hull shader.
// See the hull shader: [partitioning(TERRAIN_HS_PARTITION)]
#ifndef TERRAIN_HS_PARTITION
#define TERRAIN_HS_PARTITION "integer"
#endif

// The input patch size. It is 4 control points, the minimum
// to define a patch. This value should match the call to
// IASetPrimitiveTopology()
#define INPUT_PATCH_SIZE 4

// The output patch size. It is also 4 control points.
#define OUTPUT_PATCH_SIZE 4

//Solid terrain
RasterizerState noCullRasterizer
{
  CullMode = Front;
  FillMode = solid;
};

//Wireframe terrain
RasterizerState wiredRasterizer
```

```
{
  CullMode = Front;
  FillMode = wireframe;
};

//Background
RasterizerState defaultRasterizer
{
  CullMode = Back;
};

DepthStencilState defaultCompositor
{
};

BlendState defaultBlender
{
};

//------------------------------------------------------------------
// Constant Buffers
//------------------------------------------------------------------
cbuffer transform
{
  float4x4 viewProjMatrix;
  float4x4 orientProjMatrixInverse;
  float3 eyePosition;
}

Texture2D terrainHeightMap;
Texture2D terrainNormalMap;
Texture2D terrainTexture;

float sizeTerrain;
float heightMultiplier;
float minDistance;
float maxDistance;
float minTessExp;
float maxTessExp;
bool applyCorrection;
int sqrtNumPatch;
int textSize;

//Background
TextureCube envMap;

SamplerState linearSampler
{
  Filter = MIN_MAG_MIP_LINEAR;
  AddressU = Wrap;
```

```
  AddressV = Wrap;
};

//--------------------------------------------------------------------------------------
// Vertex shader section
//--------------------------------------------------------------------------------------
struct VS_CONTROL_POINT_INPUT
{
  float3 vPosition  : POSITION;
};

struct VS_CONTROL_POINT_OUTPUT
{
  float3 vPosition  : POSITION;
  float3 vVec0    : VECTOR0;
  float3 vVec1    : VECTOR1;
  float3 vVec2    : VECTOR2;
  float3 vVec3    : VECTOR3;
};

// This vertex shader passes the control points straight
// through to the hull shader getting the height from a
// texture. In addition it calculates four vectors to
// adjacent edges.
// The input to the vertex shader comes from the vertex
// buffer.
// The output from the vertex shader will go into the hull
// shader.

VS_CONTROL_POINT_OUTPUT vsTerrain( VS_CONTROL_POINT_INPUT
    Input )
{
  VS_CONTROL_POINT_OUTPUT Output;

  //We calculate every vector that goes from the vertex to the
  //adjacent vertexs
  //In the hull shader we will need this vectors to calculate
  //the angle of the camera in this point
  float increment = sizeTerrain / sqrtNumPatch;
  float2 coord = float2(Input.vPosition.x, Input.vPosition.z);
  float2 coord0 = float2(Input.vPosition.x, Input.vPosition.z
      + increment);
  float2 coord1 = float2(Input.vPosition.x + increment, Input.
      vPosition.z);
  float2 coord2 = float2(Input.vPosition.x, Input.vPosition.z
      - increment);
  float2 coord3 = float2(Input.vPosition.x - increment, Input.
      vPosition.z);
```

```
  float height = terrainHeightMap.SampleLevel(linearSampler,
      coord / sizeTerrain, 0).x * heightMultiplier;
  float height0 = terrainHeightMap.SampleLevel(linearSampler,
      coord0 / sizeTerrain, 0).x * heightMultiplier;
  float height1 = terrainHeightMap.SampleLevel(linearSampler,
      coord1 / sizeTerrain, 0).x * heightMultiplier;
  float height2 = terrainHeightMap.SampleLevel(linearSampler,
      coord2 / sizeTerrain, 0).x * heightMultiplier;
  float height3 = terrainHeightMap.SampleLevel(linearSampler,
      coord3 / sizeTerrain, 0).x * heightMultiplier;

  float3 pos = float3( coord.x, height, coord.y );
  float3 pos0 = float3( coord0.x, height0, coord0.y );
  float3 pos1 = float3( coord1.x, height1, coord1.y );
  float3 pos2 = float3( coord2.x, height2, coord2.y );
  float3 pos3 = float3( coord3.x, height3, coord3.y );

  Output.vVec0 = normalize(pos0 - pos);
  Output.vVec1 = normalize(pos1 - pos);
  Output.vVec2 = normalize(pos2 - pos);
  Output.vVec3 = normalize(pos3 - pos);

  Output.vPosition = pos;

  return Output;
}

//--------------------------------------------------------------
// Constant data function for the hsTerrain.  This is executed
// once per patch.
//--------------------------------------------------------------
struct HS_CONSTANT_DATA_OUTPUT
{
  float Edges[4]     : SV_TessFactor;
  float Inside[2]    : SV_InsideTessFactor;
  float mipLevel[4] : MIPLEVELVERTEXS;
};

struct HS_OUTPUT
{
  float3 vPosition  : POSITION;
};

// This constant hull shader is executed once per patch. It
// will be executed SQRT_NUMBER_OF_PATCHS *
// SQRT_NUMBER_OF_PATCHS times. We calculate a variable
// tessellation factor based on the angle and the distnace
// of the camera.
```

```
HS_CONSTANT_DATA_OUTPUT TerrainConstantHS( InputPatch<
    VS_CONTROL_POINT_OUTPUT, INPUT_PATCH_SIZE> ip, uint
    PatchID : SV_PrimitiveID )
{
  HS_CONSTANT_DATA_OUTPUT Output;

  //Calculating the central point of the patch and the central
  //point for every edge
  float3 middlePoint = (ip[0].vPosition + ip[1].vPosition + ip
      [2].vPosition + ip[3].vPosition) / 4;
  float3 middlePointEdge0 = (ip[0].vPosition + ip[1].vPosition
      ) / 2;
  float3 middlePointEdge1 = (ip[3].vPosition + ip[0].vPosition
      ) / 2;
  float3 middlePointEdge2 = (ip[2].vPosition + ip[3].vPosition
      ) / 2;
  float3 middlePointEdge3 = (ip[1].vPosition + ip[2].vPosition
      ) / 2;

  //Calculating a correction that will be applied to
  //tessellation factor depending on the angle of the camera
  //We calculate a correction for every central point of the
  //edges and the middle to avoid some lines between patches.
  float2 correctionInside = float2(1, 1);
  float4 correctionEdges = float4(1, 1, 1, 1);
  float4 correctionVertexs = float4(1, 1, 1, 1);
  if(applyCorrection)
  {
    float3 insideDirection0 = normalize(middlePointEdge0 -
        middlePointEdge2);
    float3 insideDirection1 = normalize(middlePointEdge1 -
        middlePointEdge3);
    float3 edgeDirection0 = normalize(ip[0].vPosition - ip[1].
        vPosition);
    float3 edgeDirection1 = normalize(ip[3].vPosition - ip[0].
        vPosition);
    float3 edgeDirection2 = normalize(ip[2].vPosition - ip[3].
        vPosition);
    float3 edgeDirection3 = normalize(ip[1].vPosition - ip[2].
        vPosition);

    float3 toCameraMiddle = normalize(eyePosition -
        middlePoint);
    float3 toCameraEdge0 = normalize(eyePosition -
        middlePointEdge0);
    float3 toCameraEdge1 = normalize(eyePosition -
        middlePointEdge1);
    float3 toCameraEdge2 = normalize(eyePosition -
        middlePointEdge2);
```

```
float3 toCameraEdge3 = normalize(eyePosition -
    middlePointEdge3);

float rank = 0.4;
float shift = 1 - rank / 2;
correctionInside.x = ( ( 1.57 - acos( abs( dot(
    toCameraMiddle, insideDirection0 ) ) ) ) / 1.57 ) *
    rank + shift;
correctionInside.y = ( ( 1.57 - acos( abs( dot(
    toCameraMiddle, insideDirection1 ) ) ) ) / 1.57 ) *
    rank + shift;

correctionEdges.x = ( ( 1.57 - acos( abs( dot(
    toCameraEdge0, edgeDirection0 ) ) ) ) / 1.57 ) * rank
    + shift;
correctionEdges.y = ( ( 1.57 - acos( abs( dot(
    toCameraEdge1, edgeDirection1 ) ) ) ) / 1.57 ) * rank
    + shift;
correctionEdges.z = ( ( 1.57 - acos( abs( dot(
    toCameraEdge2, edgeDirection2 ) ) ) ) / 1.57 ) * rank
    + shift;
correctionEdges.w = ( ( 1.57 - acos( abs( dot(
    toCameraEdge3, edgeDirection3 ) ) ) ) / 1.57 ) * rank
    + shift;

float3 toCameraVertex0 = normalize(eyePosition - ip[0].
    vPosition);
float3 toCameraVertex1 = normalize(eyePosition - ip[1].
    vPosition);
float3 toCameraVertex2 = normalize(eyePosition - ip[2].
    vPosition);
float3 toCameraVertex3 = normalize(eyePosition - ip[3].
    vPosition);

float angle0 = ( acos( abs( dot( toCameraVertex0, ip[0].
    vVec0 ) ) ) + acos( abs( dot( toCameraVertex0, ip[0].
    vVec1 ) ) ) + acos( abs( dot( toCameraVertex0, ip[0].
    vVec2 ) ) ) + acos( abs( dot( toCameraVertex0, ip[0].
    vVec3 ) ) ) ) / 4;
float angle1 = ( acos( abs( dot( toCameraVertex1, ip[1].
    vVec0 ) ) ) + acos( abs( dot( toCameraVertex1, ip[1].
    vVec1 ) ) ) + acos( abs( dot( toCameraVertex1, ip[1].
    vVec2 ) ) ) + acos( abs( dot( toCameraVertex1, ip[1].
    vVec3 ) ) ) ) / 4;
float angle2 = ( acos( abs( dot( toCameraVertex2, ip[2].
    vVec0 ) ) ) + acos( abs( dot( toCameraVertex2, ip[2].
    vVec1 ) ) ) + acos( abs( dot( toCameraVertex2, ip[2].
    vVec2 ) ) ) + acos( abs( dot( toCameraVertex2, ip[2].
    vVec3 ) ) ) ) / 4;
```

```
  float angle3 = ( acos( abs( dot( toCameraVertex3, ip[3].
      vVec0 ) ) ) + acos( abs( dot( toCameraVertex3, ip[3].
      vVec1 ) ) ) + acos( abs( dot( toCameraVertex3, ip[3].
      vVec2 ) ) ) + acos( abs( dot( toCameraVertex3, ip[3].
      vVec3 ) ) ) ) / 4;

  correctionVertexs.x = ( 1 - ( ( 1.57 - angle0 ) / 1.57 ) )
       * rank + shift;
  correctionVertexs.y = ( 1 - ( ( 1.57 - angle1 ) / 1.57 ) )
       * rank + shift;
  correctionVertexs.z = ( 1 - ( ( 1.57 - angle2 ) / 1.57 ) )
       * rank + shift;
  correctionVertexs.w = ( 1 - ( ( 1.57 - angle3 ) / 1.57 ) )
       * rank + shift;
}

//Calculating the distance for every patch vertex to the
//camera
float4 magnitudeVertexs;
magnitudeVertexs.x = clamp(distance(ip[0].vPosition,
    eyePosition), minDistance, maxDistance);
magnitudeVertexs.y = clamp(distance(ip[1].vPosition,
    eyePosition), minDistance, maxDistance);
magnitudeVertexs.z = clamp(distance(ip[2].vPosition,
    eyePosition), minDistance, maxDistance);
magnitudeVertexs.w = clamp(distance(ip[3].vPosition,
    eyePosition), minDistance, maxDistance);

//Calculating the minimum and maximum mip-map level to be
//used
float minMipmapLevel = log2(textSize) - log2( sqrtNumPatch *
     pow(2, maxTessExp) );
float maxMipmapLevel = log2(textSize) - log2( sqrtNumPatch *
     pow(2, minTessExp) );
if( minMipmapLevel < 0)
  minMipmapLevel = 0;

//Calculating the diference between the maximum and the
//minimum distance
float diffDistance = maxDistance - minDistance;

//Calculating a mip map level factor from 0 to 1 for every
//vertex.
//We apply the correction depending on the angle of the
//camera.
float4 factorVertexs = clamp( ( (magnitudeVertexs -
    minDistance) / diffDistance ) * correctionVertexs,
    float4(0, 0, 0, 0), float4(1, 1, 1, 1) );

//Linear interpolation between the minimum and the maximum
```

```
//mipmap level
Output.mipLevel[0] = lerp(minMipmapLevel, maxMipmapLevel,
    factorVertexs.x);
Output.mipLevel[1] = lerp(minMipmapLevel, maxMipmapLevel,
    factorVertexs.y);
Output.mipLevel[2] = lerp(minMipmapLevel, maxMipmapLevel,
    factorVertexs.z);
Output.mipLevel[3] = lerp(minMipmapLevel, maxMipmapLevel,
    factorVertexs.w);

//Calculating the distance from the central point of the
//patch to the camera and from every central point of every
//edge to the camera
float magnitude = clamp(distance(middlePoint, eyePosition),
    minDistance, maxDistance);
float4 magnitudeEdges;
magnitudeEdges.x = clamp(distance(middlePointEdge0,
    eyePosition), minDistance, maxDistance);
magnitudeEdges.y = clamp(distance(middlePointEdge1,
    eyePosition), minDistance, maxDistance);
magnitudeEdges.z = clamp(distance(middlePointEdge2,
    eyePosition), minDistance, maxDistance);
magnitudeEdges.w = clamp(distance(middlePointEdge3,
    eyePosition), minDistance, maxDistance);

//Calculating a tessellation factor from 0 to 1 for every
//edge and the middle.
//We apply the correction depending on the angle of the
//camera.
float2 factorInside = 1 - saturate( ( (magnitude -
    minDistance) / diffDistance ) * correctionInside );
float4 factorEdges = 1 - saturate( ( (magnitudeEdges -
    minDistance) / diffDistance ) * correctionEdges );

//Linear interpolation between the minimum and the maximum
//tessellation exponent. Then we round it and we raised 2
//to this value to get a power of two tessellation factor.
Output.Edges[0] = pow( 2, round(lerp(minTessExp, maxTessExp,
     factorEdges.x)) );
Output.Edges[1] = pow( 2, round(lerp(minTessExp, maxTessExp,
     factorEdges.y)) );
Output.Edges[2] = pow( 2, round(lerp(minTessExp, maxTessExp,
     factorEdges.z)) );
Output.Edges[3] = pow( 2, round(lerp(minTessExp, maxTessExp,
     factorEdges.w)) );

Output.Inside[0] = pow( 2, round(lerp(minTessExp, maxTessExp
    , factorInside.x)) );
Output.Inside[1] = pow( 2, round(lerp(minTessExp, maxTessExp
    , factorInside.y)) );
```

```
  return Output;
}

// The hull shader is called once per output control point,
// which is specified with outputcontrolpoints. In this case,
// we take the control points from the vertex shader and pass
// them directly off to the domain shader.
// The input to the hull shader comes from the vertex shader.
// The output from the hull shader will go to the domain
// shader.
// The tessellation factor, topology, and partition mode will
// go to the fixed function tessellator stage to calculate the
// UVW and domain points.

[domain("quad")]
[partitioning(TERRAIN_HS_PARTITION)]
[outputtopology("triangle_cw")]
[outputcontrolpoints(OUTPUT_PATCH_SIZE)]
[patchconstantfunc("TerrainConstantHS")]
HS_OUTPUT hsTerrain( InputPatch<VS_CONTROL_POINT_OUTPUT,
    INPUT_PATCH_SIZE> p, uint i : SV_OutputControlPointID,
    uint PatchID : SV_PrimitiveID )
{
  HS_OUTPUT Output;
  Output.vPosition = p[i].vPosition;
  return Output;
}

//--------------------------------------------------------------
// Domain shader section
//--------------------------------------------------------------
struct DS_OUTPUT
{
  float4 vPosition  : SV_POSITION;
  float3 vNormal    : NORMAL;
  float2 tex        : TEXCOORD0;
};

// The domain shader is run once per vertex and calculates the
// final vertex's position and attributes. It receives the UVW
// from the fixed function tessellator and the control point
// outputs from the hull shader. Since we are using the
// DirectX 11 Tessellation pipeline, it is the domain shader's
// responsibility to calculate the final SV_POSITION for each
// vertex.
// The input SV_DomainLocation to the domain shader comes from
// fixed function tessellator. And the OutputPatch comes from
// the hull shader. From these, we must calculate the final
// vertex position, color, texcoords, and other attributes.
```

```
// The output from the domain shader will be a vertex that
// will go to the video card's rasterization pipeline and get
// drawn to the screen.

[domain("quad")]
DS_OUTPUT dsTerrain( HS_CONSTANT_DATA_OUTPUT input, float2 UV
    : SV_DomainLocation, const OutputPatch<HS_OUTPUT,
    OUTPUT_PATCH_SIZE> patch )
{
  //All the initial patches have to be the same height
  //Linear interpolation between the position of the corners
  float WorldPosX = patch[0].vPosition.x + ((patch[2].
      vPosition.x - patch[0].vPosition.x) * UV.x);
  float WorldPosY;
  float WorldPosZ = patch[0].vPosition.z + ((patch[2].
      vPosition.z - patch[0].vPosition.z) * UV.y);
  float2 textCoord = float2(WorldPosX, WorldPosZ) /
      sizeTerrain;

  //Interpolation of mip-map level between four mip-map level
  //in the vertexs
  float mipLevel01 = (UV.y * input.mipLevel[1] + (1 - UV.y) *
      input.mipLevel[0]);
  float mipLevel32 = (UV.y * input.mipLevel[2] + (1 - UV.y) *
      input.mipLevel[3]);
  float mipLevel = (UV.x * mipLevel32 + (1 - UV.x) *
      mipLevel01);

  //Setting the terrain height from a height map texture and
  //scale by heightMultiplier
  WorldPosY = terrainHeightMap.SampleLevel(linearSampler,
      textCoord, mipLevel).x * heightMultiplier;
  //Getting the normals from a normal map texture
  float3 normal = terrainNormalMap.SampleLevel(linearSampler,
      textCoord, mipLevel).xyz;

  DS_OUTPUT Output;

  //Setting position, world position, normal and texture
  //coordinates
  Output.vPosition = mul( float4(float3(WorldPosX, WorldPosY,
      WorldPosZ),1), viewProjMatrix );
  //We change the range from [0,1] to [-1,1]
  Output.vNormal = normalize(normal * 2.0f - 1.0f);
  Output.tex = textCoord;

  return Output;
}

//--------------------------------------------------------------
```

```
// Shading pixel shader section
//--------------------------------------------------------------------

// The pixel shader works the same as it would in a normal
// graphics pipeline. In this case, we get the color from a
// texture and we perform a simple lighting.

float4 psTerrain(DS_OUTPUT input) : SV_TARGET
{
  //We have to normalize again because can be not normalized
  float3 N = normalize(input.vNormal);
  float3 L = normalize(float3(3, 2, 0));

  float4 color = terrainTexture.Sample(linearSampler, input.
      tex * 10);

  return color * saturate(dot(N, L)) + color * 0.1;
}

//--------------------------------------------------------------------
// Solid color shading pixel shader (used for wireframe
// overlay)
//--------------------------------------------------------------------
float4 psTerrainWired( DS_OUTPUT Input ) : SV_TARGET
{
  // Return a solid white color
  return float4( 1, 1, 1, 1 );
}

//--------------------------------------------------------------------
// Normal color shading pixel shader (used for normal overlay)
//--------------------------------------------------------------------
float4 psTerrainNormal(DS_OUTPUT input) : SV_TARGET
{
  //We have to normalize again because can be not normalized
  float3 N = normalize(input.vNormal);

  return float4(N, 1);
}

//Technique used to render the terrain
technique11 terrain
{
  pass P0
  {
    SetVertexShader ( CompileShader( vs_5_0, vsTerrain() ) );
    SetHullShader ( CompileShader( hs_5_0, hsTerrain() ) );
    SetDomainShader ( CompileShader( ds_5_0, dsTerrain() ) );
    SetRasterizerState( noCullRasterizer );
    SetPixelShader( CompileShader( ps_5_0, psTerrain() ) );
```

```
  }
}

//Technique used to render the terrain normals
technique11 terrain_normal
{
  pass P0
  {
    SetVertexShader ( CompileShader( vs_5_0, vsTerrain() ) );
    SetHullShader ( CompileShader( hs_5_0, hsTerrain() ) );
    SetDomainShader ( CompileShader( ds_5_0, dsTerrain() ) );
    SetRasterizerState( noCullRasterizer );
    SetPixelShader( CompileShader( ps_5_0, psTerrainNormal() )
        );
  }
}

//Technique used to render the wireframe terrain
technique11 wired_terrain
{
  pass P0
  {
    SetVertexShader ( CompileShader( vs_5_0, vsTerrain() ) );
    SetHullShader ( CompileShader( hs_5_0, hsTerrain() ) );
    SetDomainShader ( CompileShader( ds_5_0, dsTerrain() ) );
    SetRasterizerState( wiredRasterizer );
    SetPixelShader( CompileShader( ps_5_0, psTerrainWired() )
       );
  }
}

//Technique used to render the wireframe terrain normals
technique11 wired_terrain_normal
{
  pass P0
  {
    SetVertexShader ( CompileShader( vs_5_0, vsTerrain() ) );
    SetHullShader ( CompileShader( hs_5_0, hsTerrain() ) );
    SetDomainShader ( CompileShader( ds_5_0, dsTerrain() ) );
    SetRasterizerState( wiredRasterizer );
    SetPixelShader( CompileShader( ps_5_0, psTerrainNormal() )
        );
  }
}

//This last part of the fx file is to render the background
struct QuadInput
{
  float4  pos : POSITION;
  float2  tex : TEXCOORD0;
```

```
};

struct QuadOutput
{
  float4 pos     : SV_POSITION;
  float2 tex     : TEXCOORD0;
  float3 viewDir  : TEXCOORD1;
};

float4 psBackground(QuadOutput input) : SV_TARGET
{
  return envMap.Sample(linearSampler, input.viewDir);
};

QuadOutput vsQuad(QuadInput input)
{
  QuadOutput output = (QuadOutput)0;

  output.pos = input.pos;
  float4 hWorldPosMinusEye = mul(input.pos,
      orientProjMatrixInverse);
  hWorldPosMinusEye /= hWorldPosMinusEye.w;
  output.viewDir = hWorldPosMinusEye.xyz;
  output.pos.z = 0.99999;
  output.tex = input.tex;
  return output;
};

technique11 background
{
  pass P0
  {
    SetVertexShader ( CompileShader( vs_5_0, vsQuad() ) );
    SetHullShader( NULL );
    SetDomainShader( NULL );
    SetRasterizerState( defaultRasterizer );
    SetPixelShader( CompileShader( ps_5_0, psBackground() ) );
    SetDepthStencilState( defaultCompositor, 0);
    SetBlendState(defaultBlender, float4( 0.0f, 0.0f, 0.0f,
        0.0f ), 0xffffffff);
  }
}
```

Appendix B

Ocean fx file

```
//--------------------------------------------------------------------------------------
// File: ocean.fx
//
// This file shows an implementation of the DirectX 11
// Hardware Tessellator for rendering a ocean.
//
// Xavi Bonaventura Brugus
//--------------------------------------------------------------------------------------

// This allows us to compile the shader with a #define to
// choose the different partition modes for the hull shader.
// See the hull shader: [partitioning(OCEAN_HS_PARTITION)]
#ifndef OCEAN_HS_PARTITION
#define OCEAN_HS_PARTITION "fractional_even"
#endif

// The input patch size. It is 4 control points, the minimum
// to define a patch. This value should match the call to
// IASetPrimitiveTopology()
#define INPUT_PATCH_SIZE 4

// The output patch size. It is also 4 control points.
#define OUTPUT_PATCH_SIZE 4

#include "waves.h"
RasterizerState noCullRasterizer
{
  CullMode = Front;
  FillMode = solid;
};

RasterizerState wiredRasterizer
{
```

```
  CullMode = Front;
  FillMode = wireframe;
};

//Background
RasterizerState defaultRasterizer
{
  CullMode = Back;
};

DepthStencilState defaultCompositor
{
};

BlendState defaultBlender
{
};

//------------------------------------------------------------------
// Constant Buffers
//------------------------------------------------------------------
cbuffer transform
{
  float4x4 viewProjMatrix;
  float4x4 orientProjMatrixInverse;
  float3 eyePosition;
}

Texture2D oceanTexture;
Texture2D bumpTexture;

float sizeTerrain;
float time;

//Background
TextureCube envMap;

SamplerState linearSampler
{
  Filter = MIN_MAG_MIP_LINEAR;
  AddressU = Wrap;
  AddressV = Wrap;
};

bool applyCorrection;
float minDistance;
float maxDistance;
float minTessExp;
float maxTessExp;
```

```
//--------------------------------------------------------------------------------
// Vertex shader section
//--------------------------------------------------------------------------------
struct VS_CONTROL_POINT_INPUT
{
  float3 vPosition  : POSITION;
};

struct VS_CONTROL_POINT_OUTPUT
{
  float3 vPosition  : POSITION;
};

// This vertex shader passes the control points straight
// through to the hull shader.
// The input to the vertex shader comes from the vertex
// buffer.
// The output from the vertex shader will go into the hull
// shader.

VS_CONTROL_POINT_OUTPUT vsOcean( VS_CONTROL_POINT_INPUT Input
    )
{
  VS_CONTROL_POINT_OUTPUT Output;

  Output.vPosition = Input.vPosition;

  return Output;
}

//--------------------------------------------------------------------------------
// Constant data function for the hsOcean. This is executed
// once per patch.
//--------------------------------------------------------------------------------
struct HS_CONSTANT_DATA_OUTPUT
{
  float Edges[4]  : SV_TessFactor;
  float Inside[2] : SV_InsideTessFactor;
};

struct HS_OUTPUT
{
  float3 vPosition  : POSITION;
};

// This constant hull shader is executed once per patch. It
// will be executed SQRT_NUMBER_OF_PATCHS *
// SQRT_NUMBER_OF_PATCHS times. We calculate a variable
// tessellation factor based on the angle and the distnace
// of the camera.
```

```
HS_CONSTANT_DATA_OUTPUT OceanConstantHS( InputPatch<
    VS_CONTROL_POINT_OUTPUT, INPUT_PATCH_SIZE> ip, uint
    PatchID : SV_PrimitiveID )
{
  HS_CONSTANT_DATA_OUTPUT Output;

  //Calculating the central point of the patch and the central
  //point for every edge
  float3 middlePoint = (ip[0].vPosition + ip[1].vPosition + ip
      [2].vPosition + ip[3].vPosition) / 4;
  float3 middlePointEdge0 = (ip[0].vPosition + ip[1].vPosition
      ) / 2;
  float3 middlePointEdge1 = (ip[3].vPosition + ip[0].vPosition
      ) / 2;
  float3 middlePointEdge2 = (ip[2].vPosition + ip[3].vPosition
      ) / 2;
  float3 middlePointEdge3 = (ip[1].vPosition + ip[2].vPosition
      ) / 2;

  //Calculating a correction that will be applied to
  //tessellation factor depending on the angle of the camera
  //We calculate a correction for every central point of the
  //edges and the middle to avoid some lines between patches.
  float2 correctionInside = float2(1, 1);
  float4 correctionEdges = float4(1, 1, 1, 1);
  if(applyCorrection)
  {
    float3 insideDirection0 = normalize(middlePointEdge0 -
        middlePointEdge2);
    float3 insideDirection1 = normalize(middlePointEdge1 -
        middlePointEdge3);
    float3 edgeDirection0 = normalize(ip[0].vPosition - ip[1].
        vPosition);
    float3 edgeDirection1 = normalize(ip[3].vPosition - ip[0].
        vPosition);
    float3 edgeDirection2 = normalize(ip[2].vPosition - ip[3].
        vPosition);
    float3 edgeDirection3 = normalize(ip[1].vPosition - ip[2].
        vPosition);

    float3 toCameraMiddle = normalize(eyePosition -
        middlePoint);
    float3 toCameraEdge0 = normalize(eyePosition -
        middlePointEdge0);
    float3 toCameraEdge1 = normalize(eyePosition -
        middlePointEdge1);
    float3 toCameraEdge2 = normalize(eyePosition -
        middlePointEdge2);
```

```
    float3 toCameraEdge3 = normalize(eyePosition -
        middlePointEdge3);

    float rank = 0.4;
    float shift = 1 - rank / 2;
    correctionInside.x = ( ( 1.57 - acos( abs( dot(
        toCameraMiddle, insideDirection0 ) ) ) ) / 1.57 ) *
        rank + shift;
    correctionInside.y = ( ( 1.57 - acos( abs( dot(
        toCameraMiddle, insideDirection1 ) ) ) ) / 1.57 ) *
        rank + shift;

    correctionEdges.x = ( ( 1.57 - acos( abs( dot(
        toCameraEdge0, edgeDirection0 ) ) ) ) / 1.57 ) * rank
        + shift;
    correctionEdges.y = ( ( 1.57 - acos( abs( dot(
        toCameraEdge1, edgeDirection1 ) ) ) ) / 1.57 ) * rank
        + shift;
    correctionEdges.z = ( ( 1.57 - acos( abs( dot(
        toCameraEdge2, edgeDirection2 ) ) ) ) / 1.57 ) * rank
        + shift;
    correctionEdges.w = ( ( 1.57 - acos( abs( dot(
        toCameraEdge3, edgeDirection3 ) ) ) ) / 1.57 ) * rank
        + shift;
}

//Calculating the distance from the central point of the
//patch to the camera and from every central point of every
//edge to the camera
float magnitude = clamp(distance(middlePoint, eyePosition),
    minDistance, maxDistance);
float4 magnitudeEdges;
magnitudeEdges.x = clamp(distance(middlePointEdge0,
    eyePosition), minDistance, maxDistance);
magnitudeEdges.y = clamp(distance(middlePointEdge1,
    eyePosition), minDistance, maxDistance);
magnitudeEdges.z = clamp(distance(middlePointEdge2,
    eyePosition), minDistance, maxDistance);
magnitudeEdges.w = clamp(distance(middlePointEdge3,
    eyePosition), minDistance, maxDistance);

//Calculating the diference between the maximum and the
//minimum distance
float diffDistance = maxDistance - minDistance;

//Calculating a tessellation factor from 0 to 1 for every
//edge and the middle.
//We apply the correction depending on the angle of the
//camera.
```

```
  float2 factorInside = 1 - saturate( ( (magnitude -
      minDistance) / diffDistance ) * correctionInside );
  float4 factorEdges = 1 - saturate( ( (magnitudeEdges -
      minDistance) / diffDistance ) * correctionEdges );

  //Linear interpolation between the minimum and the maximum
  //tessellation exponent. Then we raised 2
  //to this value to get the tessellation factor.
  Output.Edges[0] = pow(2, lerp(minTessExp, maxTessExp,
      factorEdges.x));
  Output.Edges[1] = pow(2, lerp(minTessExp, maxTessExp,
      factorEdges.y));
  Output.Edges[2] = pow(2, lerp(minTessExp, maxTessExp,
      factorEdges.z));
  Output.Edges[3] = pow(2, lerp(minTessExp, maxTessExp,
      factorEdges.w));

  Output.Inside[0] = pow(2, lerp(minTessExp, maxTessExp,
      factorInside.x));
  Output.Inside[1] = pow(2, lerp(minTessExp, maxTessExp,
      factorInside.y));

  return Output;
}

// The hull shader is called once per output control point,
// which is specified with outputcontrolpoints. In this case,
// we take the control points from the vertex shader and pass
// them directly off to the domain shader.
// The input to the hull shader comes from the vertex shader.
// The output from the hull shader will go to the domain
// shader.
// The tessellation factor, topology, and partition mode will
// go to the fixed function tessellator stage to calculate the
// UVW and domain points.

[domain("quad")]
[partitioning(OCEAN_HS_PARTITION)]
[outputtopology("triangle_cw")]
[outputcontrolpoints(OUTPUT_PATCH_SIZE)]
[patchconstantfunc("OceanConstantHS")]
HS_OUTPUT hsOcean( InputPatch<VS_CONTROL_POINT_OUTPUT,
    INPUT_PATCH_SIZE> p, uint i : SV_OutputControlPointID,
    uint PatchID : SV_PrimitiveID )
{
  HS_OUTPUT Output;
  Output.vPosition = p[i].vPosition;
  return Output;
}
```

```
//--------------------------------------------------------------------------------
// Domain shader section
//--------------------------------------------------------------------------------
struct DS_OUTPUT
{
  float4 vPosition  : SV_POSITION;
  float3 vWorldPos  : WORLDPOS;
  float3 vNormal    : NORMAL;
  float3 vBinormal  : BINORMAL;
  float3 vTangent   : TANGENT;
};

// The domain shader is run once per vertex and calculates the
// final vertex's position and attributes. It receives the UVW
// from the fixed function tessellator and the control point
// outputs from the hull shader. Since we are using the
// DirectX 11 Tessellation pipeline, it is the domain shader's
// responsibility to calculate the final SV_POSITION for each
// vertex.
// The input SV_DomainLocation to the domain shader comes from
// fixed function tessellator. And the OutputPatch comes from
// the hull shader. From these, we must calculate the final
// vertex position, color, texcoords, and other attributes.
// The output from the domain shader will be a vertex that
// will go to the video card's rasterization pipeline and get
// drawn to the screen.

[domain("quad")]
DS_OUTPUT dsOcean( HS_CONSTANT_DATA_OUTPUT input, float2 UV :
    SV_DomainLocation, const OutputPatch<HS_OUTPUT,
    OUTPUT_PATCH_SIZE> patch )
{
  //Linear interpolation between the position of the corners
  float WorldPosX = patch[0].vPosition.x + ((patch[2].
      vPosition.x - patch[0].vPosition.x) * UV.x);
  float WorldPosZ = patch[0].vPosition.z + ((patch[2].
      vPosition.z - patch[0].vPosition.z) * UV.y);

  //Calculating the position and normal of every vertex
  //depending on the waves
  float3 du = float3(1, 0, 0);
  float3 dv = float3(0, 0, 1);
  float3 displacement = 0;

  for (int i=0; i<NWAVES; i++)
  {
    float pdotk = dot( float3(WorldPosX, 0, WorldPosZ), wave[i
        ].direction );
    float vel = sqrt( 1.5621f * wave[i].wavelength );
```

```
    float phase = 6.28f / wave[i].wavelength *( pdotk + vel *
        time );

    float2 offset;
    sincos(phase, offset.x, offset.y);
    offset.x *= -wave[i].amplitude;
    offset.y *= -wave[i].amplitude;

    displacement += float3(wave[i].direction.x * offset.x, -
        offset.y, wave[i].direction.z * offset.x);

    float3 da = float3(wave[i].direction.x * offset.y, offset.
        x, wave[i].direction.z * offset.y);
    da *= 6.28 / wave[i].wavelength;
    du += da * wave[i].direction.x;
    dv += da * wave[i].direction.z;

  }

  //Calculation of the final position
  float3 position = float3(WorldPosX, 0, WorldPosZ) +
      displacement;

  DS_OUTPUT Output;

  Output.vPosition = mul( float4(position,1), viewProjMatrix )
      ;
  Output.vWorldPos = position;
  Output.vBinormal = normalize(du);
  Output.vTangent = normalize(dv);
  Output.vNormal = normalize(cross(dv,du));

  return Output;
}

//--------------------------------------------------------------
// Shading pixel shader section
//--------------------------------------------------------------

// The pixel shader works the same as it would in a normal
// graphics pipeline. In this case, we apply a Fresnel
// reflection and we add some noise to the normal vector.

float4 psOcean(DS_OUTPUT input) : SV_TARGET
{
  float3 positionWorld = input.vWorldPos;
  float2 coord = float2(positionWorld.x, positionWorld.z) /
      sizeTerrain;

  //We add some noise to get a more realistic water
```

```
    float4 t0 = bumpTexture.Sample(linearSampler, float2(coord *
        100 * 0.02 + 2*0.07 * float2(1,0) * (time / 10) *
       float2(-1,-1)));
    float4 t1 = bumpTexture.Sample(linearSampler, float2(coord *
        100 * 0.03 + 2*0.057 * float2(0.86, 0.5) * (time / 10)
       * float2(1,-1)));
    float4 t2 = bumpTexture.Sample(linearSampler, float2(coord *
        100 * 0.05 + 2*0.047 * float2(0.86, -0.5) * (time / 10)
        * float2(-1,1)));
    float4 t3 = bumpTexture.Sample(linearSampler, float2(coord *
        100 * 0.07 + 2*0.037 * float2(0.7, 0.7) * (time / 10)))
       ;

    float3 N = (t0 + t1 + t2 + t3).xzy * 2.0 - 3.3;
    N.xz *= 2.0;

    N = normalize(N);

    float3x3 m;
    m[0] = input.vTangent;
    m[1] = input.vNormal;
    m[2] = input.vBinormal;

    float3 normalWorld = mul(N, m);
    normalWorld = normalize(normalWorld);
    //normalWorld = input.vNormal; //remove bumps

    float3 toCamera = normalize(eyePosition - positionWorld);

    float cosAngle = max(dot( toCamera, normalWorld ), 0);
    float F = 0.02 + 0.98 * pow( ( 1 - cosAngle ), 5 );

    float4 shallowColor = float4 (0.56, 0.62, 0.8 ,0);
    float4 deepColor = float4(0.2, 0.3, 0.5, 0);

    float3 reflDir = reflect(eyePosition - positionWorld, N);
    float4 envColor = envMap.Sample(linearSampler, reflDir);

    float4 waterColor = deepColor * cosAngle + shallowColor * (
       1 - cosAngle );
    waterColor *= oceanTexture.Sample(linearSampler, coord *
       100);
    float4 color = envColor * F + waterColor * ( 1 - F );

    return color;
}

//--------------------------------------------------------------
// Solid color shading pixel shader (used for wireframe
// overlay)
```

```
//--------------------------------------------------------------------------------------
float4 psOceanWired( DS_OUTPUT Input ) : SV_TARGET
{
  // Return a solid white color
  return float4( 1, 1, 1, 1 );
}

//--------------------------------------------------------------------------------------
// Normal color shading pixel shader (used for normal overlay)
//--------------------------------------------------------------------------------------
float4 psOceanNormal(DS_OUTPUT input) : SV_TARGET
{
  //We have to normalize again because can be not normalized
  float3 N = normalize(input.vNormal);
  return float4(N, 1);
}

//Technique used to render the ocean
technique11 ocean
{
  pass P0
  {
    SetVertexShader ( CompileShader( vs_5_0, vsOcean() ) );
    SetHullShader ( CompileShader( hs_5_0, hsOcean() ) );
    SetDomainShader ( CompileShader( ds_5_0, dsOcean() ) );
    SetRasterizerState( noCullRasterizer );
    SetPixelShader( CompileShader( ps_5_0, psOcean() ) );
  }
}

//Technique used to render the ocean normals
technique11 ocean_normal
{
  pass P0
  {
    SetVertexShader ( CompileShader( vs_5_0, vsOcean() ) );
    SetHullShader ( CompileShader( hs_5_0, hsOcean() ) );
    SetDomainShader ( CompileShader( ds_5_0, dsOcean() ) );
    SetRasterizerState( noCullRasterizer );
    SetPixelShader( CompileShader( ps_5_0, psOceanNormal() ) )
        ;
  }
}

//Technique used to render the wireframe ocean
technique11 wired_ocean
{
  pass P0
  {
    SetVertexShader ( CompileShader( vs_5_0, vsOcean() ) );
```

```
    SetHullShader ( CompileShader( hs_5_0, hsOcean() ) );
    SetDomainShader ( CompileShader( ds_5_0, dsOcean() ) );
    SetRasterizerState( wiredRasterizer );
    SetPixelShader( CompileShader( ps_5_0, psOceanWired() ) );
  }
}

//Technique used to render the wireframe ocean normals
technique11 wired_ocean_normal
{
  pass P0
  {
    SetVertexShader ( CompileShader( vs_5_0, vsOcean() ) );
    SetHullShader ( CompileShader( hs_5_0, hsOcean() ) );
    SetDomainShader ( CompileShader( ds_5_0, dsOcean() ) );
    SetRasterizerState( wiredRasterizer );
    SetPixelShader( CompileShader( ps_5_0, psOceanNormal() ) )
        ;
  }
}

//This last part of the fx file is to render the background
struct QuadInput
{
  float4  pos : POSITION;
  float2  tex : TEXCOORD0;
};

struct QuadOutput
{
  float4 pos     : SV_POSITION;
  float2 tex     : TEXCOORD0;
  float3 viewDir  : TEXCOORD1;
};

float4 psBackground(QuadOutput input) : SV_TARGET
{
  return envMap.Sample(linearSampler, input.viewDir);
};

QuadOutput vsQuad(QuadInput input)
{
  QuadOutput output = (QuadOutput)0;

  output.pos = input.pos;
  float4 hWorldPosMinusEye = mul(input.pos,
      orientProjMatrixInverse);
  hWorldPosMinusEye /= hWorldPosMinusEye.w;
  output.viewDir = hWorldPosMinusEye.xyz;
  output.pos.z = 0.99999;
```

```
  output.tex = input.tex;
  return output;
};

technique11 background
{
  pass P0
  {
    SetVertexShader ( CompileShader( vs_5_0, vsQuad() ) );
    SetHullShader( NULL );
    SetDomainShader( NULL );
    SetRasterizerState( defaultRasterizer );
    SetPixelShader( CompileShader( ps_5_0, psBackground() ) );
    SetDepthStencilState( defaultCompositor, 0);
    SetBlendState(defaultBlender, float4( 0.0f, 0.0f, 0.0f,
        0.0f ), 0xffffffff);
  }
}
```