GPU Based Algorithms for Terrain Texturing

Kris Nicholson

November 6, 2008

Department of Computer Science & Software Engineering

University of Canterbury, Christchurch, New Zealand

Supervisor: Dr. R Mukundan

Contents

Abstract

Three dimensional terrain rendering is used for various computer applications such as mapping software, flight and other simulations, and computer games. Rendering these terrains at a real time frame rate, with a continuously changing view point can be difficult since they contain a large number of polygons. Techniques have been developed to simplify the rendering task. Level of detail (LOD) algorithms can be applied to these terrains to reduce the detail when it is not required. Effective texturing and lighting is usually applied to keep realism while decreasing the polygon count. Using available graphics hardware for these techniques is very useful for increasing performance. Current programmable graphics hardware allows much greater performance for graphics applications than traditional algorithms running on the CPU. Previous research has shown various improvements in terrain rendering performance using GPU programming. In this report, we investigate GPU based procedural texturing and lighting techniques on three dimensional terrain using real terrain data. We implemented some of these techniques and comment on the visual result to help developers consider which techniques to use.

1 Introduction

Real time terrain rendering is an important field and is widely used in 3D games, flight simulators, synthetic vision systems and 3D mapping software. Due to the large scale of terrain, they require a large number of polygons to render. This is a problem if the terrain needs to be rendered in real time. Because of this, a number of Level of Detail (LOD) algorithms have been developed to reduce the number of rendered polygons based on their position relative to the viewer.

One way to increase the visible detail of terrain without increasing the number of polygons is using effective texture mapping and lighting. Texture mapping can show surface details that are not represented by polygons with the use of colour. Advanced lighting techniques can provide the illusion of extra geometry since shadows will be created as if the surface had a particular structure.

In the past few years, the perfomance of GPUs have increased very quickly, surpassing Moore's Law. Because of this, there is an increasing interest to use them in any way possible. This could be to offload processing from the CPU, or to process more complicated algorithms for more realistic effects.

The most common way for 3D terrain to be represented is in the form of a heightmap. A heightmap is simply a 2D grid of height values covering the area of the terrain. Using a brute force method we could render a terrain by simply assigning a vertex to every height value and constructing polygons from this.

A commonly used format for heightmaps is the Digital Elevation Model (DEM). DEMs are freely available from various sources including the United States Geological Survey[1] which provide DEMs for the majority of the US.

Texturing is usually able to be done quite efficiently and effectively if the model is well known. Artists can apply textures using masks and it is a simple task to blend textures together in the desired way using these masks. The situation is more complicated if we want to texture a terrain that is not known to us beforehand. This requires procedural texturing, where we must select and apply textures based on the features of the geometry alone.

In this report, we discuss some techniques regarding texturing and lighting computer generated 3D terrains using the Graphics Processing Unit(GPU). We will use procedural texturing to attempt to render a real terrain in a realistic way. In this way we hope to provide an overview of these techniques for graphics developers to consider when texturing their terrain.

This report continues with an investigation into a number of related methods that have been used for terrain rendering in section 2. We describe the various techniques we implemented in section 3. We then analyze these techniques in section 4 and finally discuss the results and indicate possible future work in section 5.

[1]United States Geological Survey - http://www.usgs.gov/

2 Background and Related Work

2.1 Shaders

Shaders are simply pieces of code that run on graphics hardware (the Graphics Processing Unit, or GPU). When shaders are used, they replace the fixed function pipeline which normally performs a range of calculations for normal rendering. There are currently three types of shader; vertex shaders, fragment (or pixel) shaders and geometry shaders. A GPU will contain a number of shader processing units to run these pieces of code in parallel (generally between 32 and 200 units for current technology). The parallel nature of the processing works without the need for constructs like mutexes and critical sections.

Originally, each shader type had it's own processing units, so there would be a number of vertex shader units and a number of fragment shader units, but more recently this has been abstracted to general purpose shader units which can be used for any type of shader. Recent graphics hardware can also be used for a more wide variety of processing other than graphics, such as physics.

2.1.1 Vertex Shaders

Vertex shaders are pieces of code that are run for every vertex of every polygon that is sent to the graphics pipeline. For the particular vertex, the shader is given the position of the vertex, the normal direction in world coordinates and the colour of the vertex as inputs.

The vertex shader is responsible for anything related to vertex and texture coordinates as the standard fixed function implementation is disabled when a vertex shader is used. This includes converting vertex coordinates from world coordinates to clip (or screen) coordinates, converting and resizing surface normals, standard per-vertex lighting, colour material computations and texture coordinate conversion or generation. Some of these steps can be skipped if they are unnecessary for the rendering task they are used for.

2.1.2 Fragment Shaders

Fragment shaders are pieces of code that are run (at least once) for every pixel that are part of polygons sent from the vertex shader. For the fragment, the shader is given the colour of the fragment (which can be bilinearly interpolated automatically between the colours of the vertices) and texture coordinates.

The fragment shader is responsible for colouring every pixel. This includes basic colours as well as any application of textures, texture environments and texture functions. Again this can be simplified if some features are unnecessary.

2.1.3 Geometry Shaders

Geometry shaders are a recent development and are only available on recent graphics hardware. They are executed after the vertex shader and operate on an entire primitive (such as three vertices for a triangle). They are able to create more primitives such as points, lines or triangles for every primitive they are passed.

Because of their focus on geometry, geometry shaders have not been used in this project.

2.1.4 Shader Languages

When shaders were first introduced, there were no high level languages available so assembly programming was required. This meant that different graphics cards (especially from different manufacturers) required the shaders to be programmed independently.

Now there are a number of different shader languages that can be used, the majority of which use a C-like syntax. High Level Shading Language (HLSL) is a Microsoft developed language to work with the DirectX graphics pipeline. Because we are using OpenGL, it is unsuitable for our project. C for graphics (Cg) is an Nvidia developed language that works under OpenGL or DirectX and can work on competitor graphics cards (such as ATi) also, but there can be problems with this. Finally, the OpenGL Shader Language (GLSL)[1] is, as the name suggests, a shader language for use with the OpenGL graphics pipeline. We have used GLSL for our project since it should be more cross-compatible than Cg.

2.2 Lighting

The OpenGL graphics pipeline uses the Phong-Blinn lighting model which is an approximation to Phong shading. This model uses a combination of three types of light. Ambient light represents the light that is present which is not lit directly by a light source. Diffuse light represents areas that are in line of sight with a light source. Specular light represents light that reflects from an object directly from the light source to the viewer. Both diffuse and specular light require the surface normal vector to determine which direction the surface is facing.

Because surface normals require some processing to calculate, often it is more efficient to store the normals in a texture map, called a normal map, and access these precalculated values at runtime. Normal maps are often used to provide the illusion of extra geometry by using a higher resolution normal map than the geometry resolution. Although this only affects the lighting, it is usually difficult to notice the lack of geometry. To use a normal map in this way, a per pixel lighting model is required, which is not supported by the standard OpenGL Phong-Blinn lighting model.

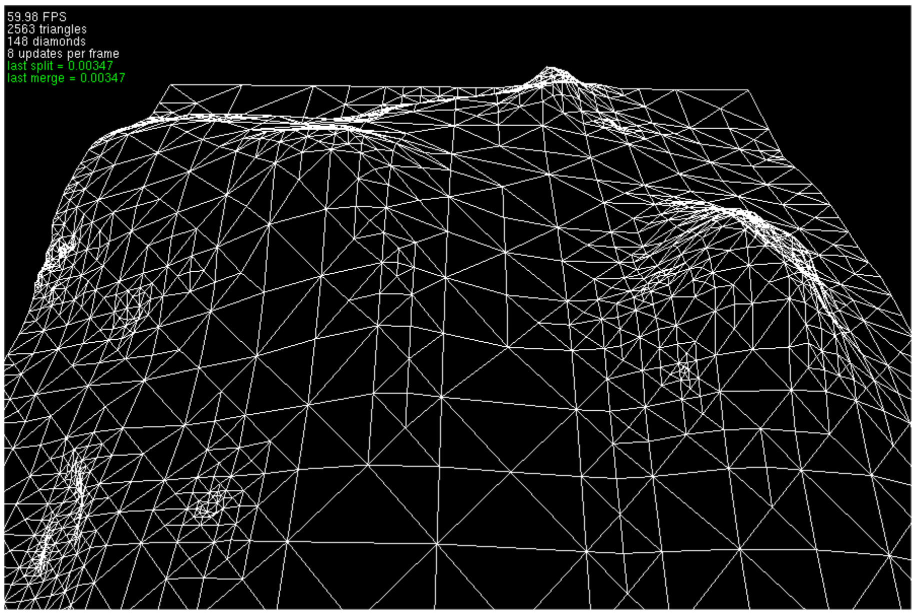
The lighting model does not take other objects in the scene into account. This means that shadows from these other objects will not be present. Shadows can be generated using projection techniques to project the model onto a 2D plane, which can then be drawn as a dark transparent object.

2.3 Terrain Rendering Algorithms

Although this paper focuses on the way a terrain is textured, the underlying algorithm that generates the geometry to render is important, especially with respect to generating texture coordinates which needs to be performed for every vertex.

2.3.1 The ROAM Algorithm

The Real-time Optimally Adapting Mesh (ROAM) algorithm [2] is a hierarchical terrain rendering algorithm that uses LOD to approximate a terrain structure



```
59.98 FPS
2563 triangles
148 diamonds
8 updates per frame
last split = 0.00347
last merge = 0.00347
```

Figure 1: A ROAM triangulation for a terrain. Notice how various levels of subdivision are applied based on the distance to the camera and the local curvature of the terrain.

based on its heightmap.

A ROAM terrain triangulation, as seen in Figure 1, consists of a number of right-angled isosceles triangles where every vertex coincides with a pixel on the heightmap, but the number of triangles will be much less than the brute force approach. The triangulation changes at runtime to provide a relatively accurate representation of the terrain through the split and merge operations as shown in Figure 2.

A split operation is performed to increase the number of triangles (and therefore, the accuracy) at a particular point in the terrain. A merge operation is performed to decrease the number of triangles at a particular point in the terrain. To decide when and where these operations need to be performed, error metrics are used. The primary error metrics used are the position and orientation of the camera and the the error in the height of the middle of the base of the triangle and the value that the heightmap would give for this position. So a larger number of triangles will be used for regions that are close to the camera and for regions that have a large amount of variance and detail.

2.3.2 GPU Based Algorithms

When early terrain rendering algorithms were developed, the primary focus was to control the number of polygons sent through the graphics pipeline as this was the major bottleneck. However, GPU capabilities have increased much more quickly than other areas of hardware. As GPUs have become able to handle more geometry, the focus has now turned to reducing the load on the CPU to perform other tasks such as AI [3, 4]. This is not to say that the brute

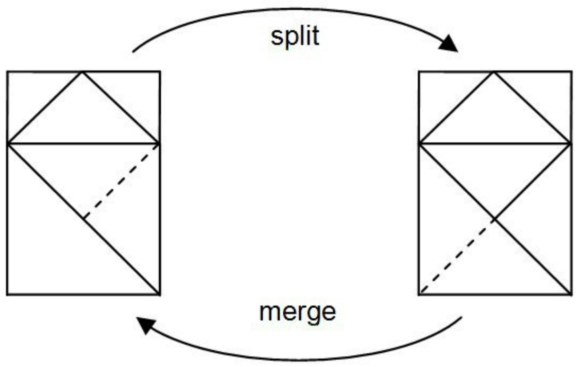


Figure 2: The ROAM split and merge operations. The dotted lines indicate a proposed split. After the split a forced split would be required to stop cracking between polygons.

force method is acceptable, but LOD algorithms need to be quick to compute for large numbers of polygons.

2.4 Previous Work on Texturing Terrain

2.4.1 Texture Blending

Blending textures to apply on a terrain was originally a fairly difficult problem. Bloom [5] reports on a way to blend these textures in the framebuffer. This requires the geometry to be drawn multiple times. Each time the geometry is rendered, one texture is applied with a varying amount of alpha blend applied. This means that when the terrain is drawn the second and subsequent times, parts of the original render (which is still in the framebuffer) will show through, producing an effective blend. Bloom argued that drawing the terrain multiple times is justified because of the increasing fill rate of newer graphics technology.

Jenks [6] describes a similar result using programmable shaders in the GPU. Emphasis is placed on using alpha maps to specify where textures should appear.

2.4.2 Procedural Texture Mapping in Frostbite

Andersson presents a paper describing the terrain rendering algorithms involved in their propietary Frostbite engine [7]. They mention how in their past engines they used a number of texture masks, painted by artists. Textures would then be blended based on these masks. For their new engine they support procedural texture mapping as well as mask based approaches when needed. These can be combined since the masks use a sparse quad-tree data structure, so they can essentially use partial masks for small areas of the terrain.

For the procedural texture mapping, three per-pixel parameters can be used to apply a particular texture. These are the height of the terrain at that point, the slope of the terrain and the normal direction. Normals are calculated in the shader using samples from the heightmap. These normals are also used for per pixel lighting, similar to what would be done with a normal map.

High frequency detail maps are used to increase the level of texture detail at close distances. These are blended with other texture maps, depending on the particular texture.

To make the transitions look more natural, fractal brownian motion noise is used to provide some randomness to the blending areas.

A number of techniques we apply are based on those found in Andersson's paper. In particular, slope based texturing.

2.4.3 Texture Synthesis and Aperiodic Texture Mapping

To texture the terrain, a texture will usually need to be repeated across the terrain a number of times since it is usually unreasonable to store a texture large enough to cover an entire terrain to an acceptable resolution. This is usually done by tiling the texture in a grid structure. The main issue with this is that tiling a texture like this will produce a noticeable repeating pattern.

Texture synthesis is a way to create a large texture based on a small sample texture. It is done in a random way so that it is much less noticeable than the traditional tiling method. It is now possible to perform texture synthesis at run-time with real-time frame rates, which should provide an effective, realistic looking texture to apply to a terrain while only using a small amount of texture memory.

A number of papers have looked into using texture synthesis [8, 9, 10]. A technique that many use involves the use of Wang tiles [11, 12] . Wang tiles are represented by a number of tiles with a range of coloured edges. These tiles must then be arranged into a grid in a way that the colours at each edge match. This can be applied to texturing a plane if we replace the coloured edge representation with textures, where the edge of each texture has a corresponding match.

These textures can either be created manually or they can be obtained automatically from one source image. This is done by performing a minimum error boundary cut in a diagonal pattern through the centre of the image.

Although this technique is effective, there is a particular problem with it. Any two tiles can be placed diagonal to each other, as long as another two tiles are picked that can match the edges of the first two. This can produce artifacts in the output. Lagae and Dutré [13] introduce a variant on Wang tiles that use coloured corners instead of coloured edges. This increases the complexity of tiling, but fixes the corner problem.

One drawback to texture synthesis is that it is dependant on the type of texture used. Some textures can be synthesised very effectively, but others are noticeably wrong. Often these textures consist of larger shapes that we as humans know should be whole, but the algorithm may split apart.

2.4.4 Satellite Imagery

Okamoto et al. investigate using satellite imagery to texture a terrain [14]. Because of the large amount of data required for the textures, the majority of the paper focuses on how to manage this data in a database management system, and the various levels of caching required to finally render the textures.

Li et al. also investigate using satellite imagery to texture a terrain[15], but rather than apply the textures directly, they use the satellite images as input for texture synthesis and generate a new texture for the terrain.

Satellite imagery would be an interesting way to texture a terrain accurately, but brings about its own challenges in terms of data management and texture detail. It also requires extra data on top of the heightmap, so we have not used it for this project.

2.4.5 Virtual Textures

Virtual textures [16] are a very new form of textures, which follow a similar structure to virtual memory. That is, the texture itself is stored in non-volatile memory such as a hard disk and only parts of the texture are stored in memory at any one time. The advantage to this method is that very large textures can be used, which is ideal for terrain rendering. One virtual texture can be used for the entire terrain, allowing artists complete control without worrying about texture size limits.

Although this method is being adopted in new game engines, its major advantage, creative control, does not help with our aim, which is to texture arbitrary heightmaps procedurally. It might possibly provide a performance improvement over other methods but we have not attempted to implement them.

2.4.6 Triplanar Texturing

Geiss and Thompson create a demo to showcase NVIDIA technology where complex 3D terrain structures called cascades are generated completely by the GPU using geometry shaders [17]. As part of this demo they need to texture the structure and do so with a technique they call triplanar texturing.

Triplanar texturing is based on planar projection texturing which is the most common method of texture coordinate generation for heightmap based terrains. Planar projection involves applying a texture to a surface as if it was a flat quadrilateral. Triplanar texturing is similar, except that we can choose the direction that the projection is applied based on the surface normal. Essentially we find which plane the surface matches the closest out of the x-z plane, the y-z plane and the x-y plane. They also apply a different texture for each planar projection.

2.4.7 Indirection Mapping

One technique is introduced by McGuire and Whitson [18] to attempt to solve the texture stretching problems that occur when using planar projection on arbitrary surfaces (See section 3.5 for more information).

Their method involves a preprocessing step which sets up a system of springs across the surface, which are then relaxed over a couple of steps to fit the surface. Using this model they can produce a texture map, called an indirection map, which modifies the texture coordinates variably based on the surface structure.

The result is a mapping that does not stretch along large slopes since the coordinates are more evenly spaced based on the surface area, rather than the planar projection area. However, this technique also distorts some flat areas of the surface, particularly around regions that are close to a largely sloped area. This is especially noticable in textures containing straight lines as part of their structure. For the tests performed in the paper, the preprocessing step took between 10 seconds and 2 minutes.

We have not used this technique in our project, mainly because of its complexity, but it could be a useful way to use to solve the texture stretching problem when texturing terrain.

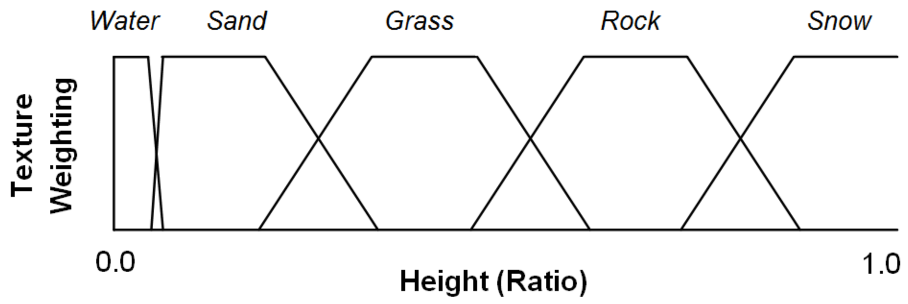


Figure 3: Graph showing texture intensity based on height

3 Implementation

We started with a previously written implementation of the ROAM terrain rendering algorithm using OpenGL. It also implemented procedural texturing, using a texture which is precalculated based on terrain height at load time, before rendering. It could also perform hardware multitexturing with a detail map if supported.

We added a new method of texturing the terrain using GLSL shaders. We developed one vertex shader and one fragment shader to shade the entire terrain. The shader contains a number of tests so that certain texturing methods can be toggled or modified at run time. Ideally the shader should actually be tailored to the specific methods that you want, but we opted towards interactivity and experimentation rather than performance.

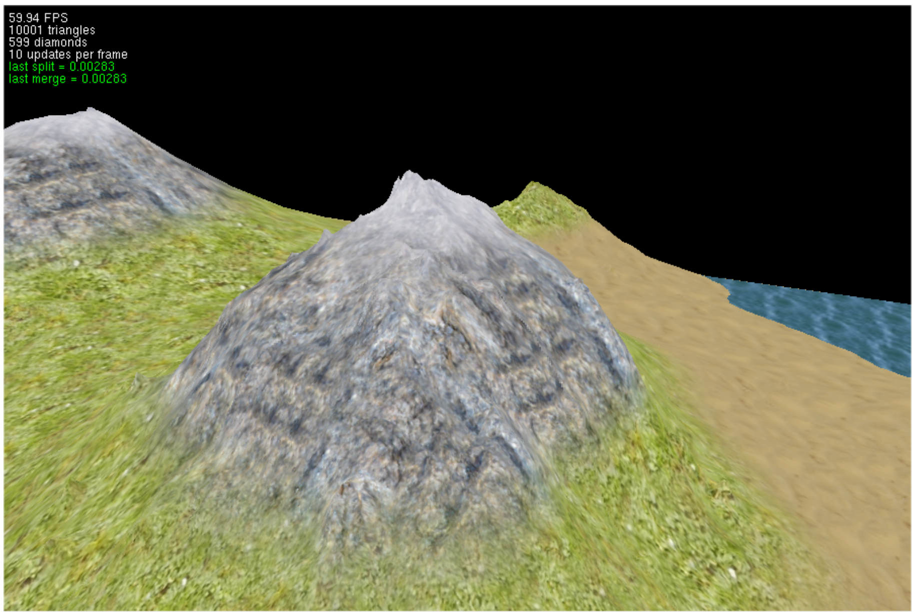
See Appendix A for the vertex shader code listing and Appendix B for the fragment shader code listing.

3.1 Height Based Texture Mapping

Height based texture mapping is a very simple method to texture a terrain in an interesting way that roughly estimates what a real terrain might look like. A texture is picked from a set depending on the height at that part of the terrain. In our case we use a water texture for the lowest points (sea-level), with sand, grass, rock and finally snow textures as the height increases.

However, if the method is kept this simple there will be harsh transitions between textures. To transition between these levels in a way that produces a more realistic look we need to blend between them. A simple blending method would use a linear interpolation between the two textures over a small region around the height level cutoff as in Figure 3. The result of this blending can be seen in Figure 4.

This level of blending can be achieved without the use of shaders, using a procedural texture. When the terrain is first loaded we create a texture with a size relative to the terrain. For each pixel in the new texture we find the corresponding point on the terrain heightmap and select the texture accordingly. If the height falls in a region close to the height level cutoff, we can blend the two textures using linear interpolation.



```
59.94 FPS
10001 triangles
599 diamonds
10 updates per frame
last split = 0.00283
last merge = 0.00283
```

Figure 4: A typical height based texture mapping application

Once this texture is created, it simply needs to be applied to the terrain using the x and z coordinates of the terrain for the texture coordinates s and t.

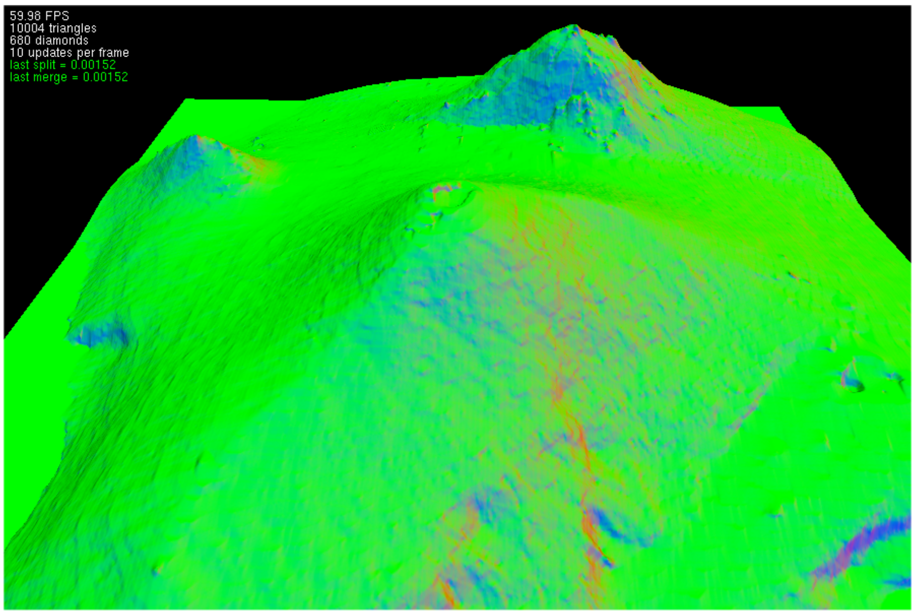
This method is effective, but there are some problems. The texture that is created must be applied over the entire terrain (or perhaps a subset of the terrain, depending on the implementation). This means that the texture resolution will be proportional to the size of the terrain. If it is not large enough, when applied it will appear pixellated (and/or blurry when using filtering). This texture could become very large and consume too much video memory (virtual textures might be used to help with this).

To overcome these problems we can use an approach using shaders. For this, we load the textures for each level when loading the terrain. The texture is then selected and blended at runtime. This can be done in the vertex shader or the fragment shader. Using the vertex shader, the height can be found from the height of the current vertex. If the fragment shader is to be used, the heightmap must also be loaded as a greyscale texture.

3.2 Detail Blending

Using standard texturing techniques, the terrain texture will always look low detail when the viewer is very close to the terrain. One way to help with this is with the use of a detail map.

A detail map is simply a texture, usually monochrome, which shows high detail including cracks and bumps. This is blended 50% with the colour texture maps, but with a very high repetition. When the viewer is close to the terrain, the colour map will be a fairly flat colour, but the detail map provides texture.



```
59.98 FPS
10004 triangles
680 diamonds
10 updates per frame
last split = 0.00152
last merge = 0.00152
```

Figure 5: Normal map data applied as a standard texture

3.3 Lighting

Lighting is a very important aspect to realistic looking terrains. It is related to our topic as the methods involved still require some use of textures. The default OpenGL lighting model may be adequate to provide simple lighting but shaders offer more interesting ways to light the terrain.

3.3.1 Normal Mapping

Normal mapping is a commonly used lighting technique in recent graphics applications normally used to provide surface detail. It requires per-pixel lighting control, which in OpenGL is only possible using the fragment shader.

A normal map is an image which represents the normal directions at every point on the surface. We can use these normals in the Phong shading model for every pixel to produce realistic lighting.

For our method of terrain rendering this technique is very easy to apply. Since we are using a heightmap, we can simply calculate the normal map from this. The normal x, y and z coordinates are converted to red, green and blue values respectively and stored in the texture. Figure 5 shows how this data is visualised when applied as a standard texture map. It is primarily green as the y coordinate of the normals are normally dominant. This makes sense as a completely flat surface would have maximum y and zero x and z components.

Passing the normal map through to the fragment shader we can then calculate the diffuse light value based on the corresponding normal.

Although the geometry can be quite low detail and can change because of the LOD algorithm used, the normal information will always be accurate and give the terrain a level of depth.

3.4 Slope Based Texture Mapping

Although height based texture mapping is reasonably effective, it is not the only parameter we can use to determine the type of texture that should be applied. Slope based texture mapping is based on the idea that large slopes usually indicate a more stable surface. It is unlikely that sand, grass or snow will be located on a cliff side. Therefore we can use another cutoff, a slope cutoff, to determine whether a different texture should be applied than what would normally be applied using height based texture mapping.

In our case we have a convenient measure of the slope, the normal, found from the normal map. However, we do not need the entire normal. All we need is a measure of the surface's deviation from a flat surface. It does not matter whether this slope is oriented towards the x or the z directions. In fact, the y component of the normal is all that is required. For a flat surface, the y component of the normal will be 1 and for a vertical slope it will be 0. Because heightmaps do not determine overhanging terrain (such as a cave or tunnel) we do not need to worry about negative values. From exploratory testing we found that a good value for the cutoff is around 0.75. Because this is calculated at runtime we can change this using key presses, which would make it very easy to find the optimal looking value.

As with height based texture mapping, this technique will produce harsh lines along the boundary condition. The solution is similar also, we just need to take an upper and lower boundary and blend between the height based result and the slope texture in this region. This works effectively most of the time. Very sharp differences in slope may still show a fairly sharp line, but given the terrain this may be appropriate.

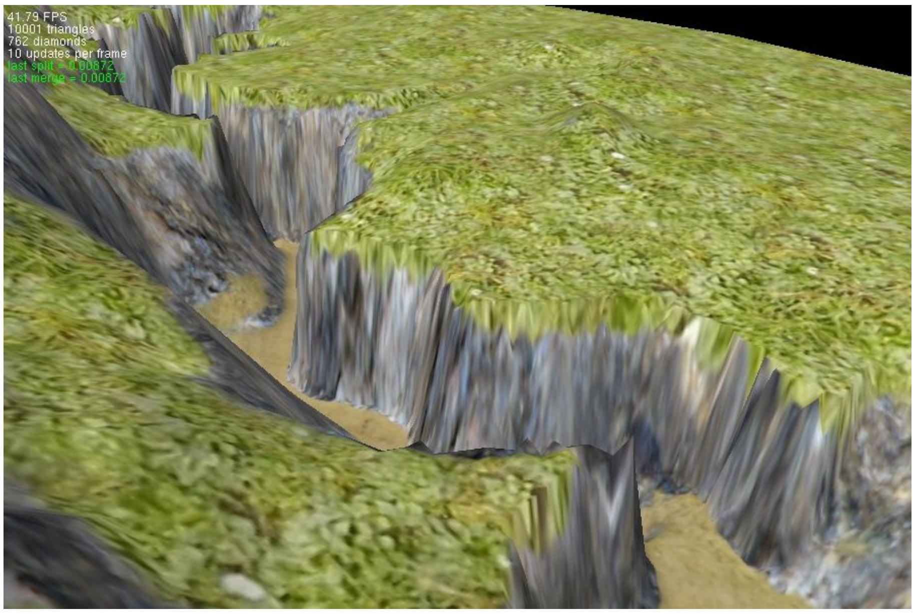
3.5 The Texture Stretching Problem

When texturing the terrain, texture coordinates are generated in a very simple way using a planar projective method which maps the x and z coordinates of the terrain geometry directly to the texture coordinates, s and t. This works well for fairly flat terrains and gentle slopes. However, any steep slopes will suffer from texture stretching which can be seen quite clearly in Figure 6.

3.5.1 Triplanar Texturing

As already mentioned, the simplest way to generate texture coordinates for a terrain is to use the geometry x and z coordinates. However for a steep cliff face, it would actually be more accurate to use the y and z or y and x coordinates, depending on the orientation of the cliff face. Triplanar texturing attempts to apply this logic.

The main algorithm to apply triplanar texturing is fairly simple. First, we check whether the slope is relatively large in the same way that we do with slope based texturing. These regions with high slope will be the only regions affected by the algorithm. We then check what the larger component of the normal is, out of x and z. If x is the larger component, we use the geometry z coordinate as the texture coordinate s, and the geometry y coordinate as the texture coordinate t. If z is the larger component, we use the geometry x coordinate as the texture coordinate s, and the geometry y coordinate as the texture coordinate t.



```
41.79 FPS
10001 triangles
762 diamonds
10 updates per frame
last split = 0.00872
last merge = 0.00872
```

Figure 6: Texture stretching on steep slopes

Using shaders, the main part of this algorithm is performed in the fragment shader, but the texture coordinates need to be calculated in the vertex shader. This means we need to pass three sets of texture coordinates through to the fragment shader, which can then pick which set it wants to use.

Unfortunately, if this algorithm is performed as above, distinct lines will appear since the texturing method changes suddenly. To avoid this we will also need to blend between the highly sloped and less sloped regions. This requires that we not only check which of the x and z coordinates are larger, but also take the values of each into account to weigh the colour value.

3.6 Water Animation

Although water rendering has improved greatly in graphics applications, we decided to implement a very simple animation which scrolls the water texture.

Because we want the animation to be continuous, we need to pass a uniform variable to the shader simply indicating the current frame number. This can then be multiplied by some factor and then used to modify the texture coordinates.

This technique is an example of how different textures can easily have extra functions applied to them to more realistically portray the material they represent.

3.7 Level of Detail Texturing

Level of detail (LOD) is a common method which is generally used to reduce the amount of data processed to improve performance when the change would be unnoticeable. In particular, geometry will usually have some form of LOD to reduce the number of polygons as the distance from the camera increases, which should increase the rendered frame rate as long as the metric used to calculate LOD is simple enough to calculate.

There are two forms of LOD, discrete and continuous. A discrete LOD system has specific levels of detail that the system will switch to. In the case of a polygon model, there will be a high detail model, but also a medium detail and a low detail version (and possibly more) of the same model. Using discrete LOD can cause noticeable ‘popping’, where a model will change suddenly and break realism.

A continuous LOD system will use some algorithm to change the level of detail incrementally. This reduces any popping effects, although a static object can look like it is constantly changing if the changes are not small enough. Continuous LOD will usually require more processing to perform and the artist also loses control of some details of the model at lower levels of detail.

In our case, we are not using LOD for performance reasons, but for visual reasons. When blending the textures together, each texture is repeated a certain number of times over the terrain. Using shaders, we can actually change this repetition value at runtime to get the effect we want. However, if texture repetitions are kept low the terrain will look very pixellated (or blurry if texture filtering is used), especially when the camera is close to the terrain. If the texture repetitions are high then the repetitions start to become visible and form a grid pattern which reduces the realism of the terrain which should have a random looking texture.

Because of this, we tried a level of detail method to change the value of these repetitions based on the distance to the camera. In this case continuous LOD would not be possible. It would create a strange warping effect with the texture. We use a discrete LOD system with 7 levels. Given a particular point, we calculate the distance to the camera. Given this distance, d, we calculate the level of detail, k, using the following equation:

$$k = floor\left(\frac{log\left(\frac{d}{7.5}\right)}{log\left(2\right)}\right)$$

The result will be any integer. The way this is implemented however, limits k to be between 1 and 7. The level will be 1 for any distance less than 15 and will increase by one each time the distance doubles.

Knowing the level of detail, we can calculate the number of texture repetitions, r, using this equation:

$$r = 10 \times \left(\frac{1}{2}\right)^{k-2}$$

So for each level of detail moving away from the camera the number of texture repetitions will halve.

This is fairly easy to do in the fragment shader, but we also experimented with performing this test in the vertex shader. Because the test will only be performed per-vertex here, it works fine for any polygon where the level of detail is constant. If the level of detail changes across one polygon, then the texture repetition value will change also, producing a strange stretching artifact. Because of this, we wanted to delegate these cases to the fragment shader while still using the vertex shader for the simple cases.

The test we used involved passing the level of detail value to the fragment shader. The OpenGL pipeline will then automatically bilinearly interpolate this value with respect to the geometry. If all three vertices of a triangle have the same value, the result will be this value. If one of the vertices are different, the result will be some ratio between them based on the location of the particular pixel. So we test whether the value passed into the fragment shader is an integer by subtracting the rounded value from itself and test if the magnitude is larger than 0. Unfortunately, as this value is a floating point number it will never be an exact integer, so we need to check that the magnitude is larger than some small value larger than 0. As it turns out, there is always a case like this in the centre of the two LOD regions, meaning there is a case this will fail. The result is that a small line will appear through the centre of the polygons that cross the LOD boundary.

As with the other methods this will create distinct lines between the levels, this time in a circular pattern around the camera. We use blending between the levels, which gives a fading effect between them.

4 Analysis

For our analysis we review each of the techniques described and compare the visual results. We also comment briefly on the performance of each method, however performance is not our primary concern here. Tests were run using a GeForce 8600GTS with 256MB of video memory.

4.1 Height Based Texture Mapping

Using shaders we no longer need a large texture to apply, since we can combine the smaller textures in real time. It also provides some extra benefits which may or may not be useful depending on the situation. In particular, because the blending is happening in real time, height levels can also be changed in real time. This could be used for a simple simulation to change the snow level. The textures could also be switched at runtime, perhaps to simulate seasons or weather effects.

Using the vertex shader is usually much faster than the fragment shader (depending on the polygon count and rendered resolution), but because it uses the geometry height coordinate rather than the actual height, it will be inaccurate. This inaccuracy is especially noticeable in the implementation of ROAM that we use since the terrain will change its level of detail as the camera moves. This can cause the texture to change as well, making the change even more noticeable. The low accuracy also has negative effects on large scale terrains where height differences are much smaller. Using the fragment shader also requires that the heightmap itself be passed to the shader as a texture, which is not normally necessary for ROAM.

4.2 Lighting

Figure 7 shows a terrain scene without lighting. After applying lighting using the normal map found from the height map, the same scene becomes a lot more realistic as can be seen in Figure 8.

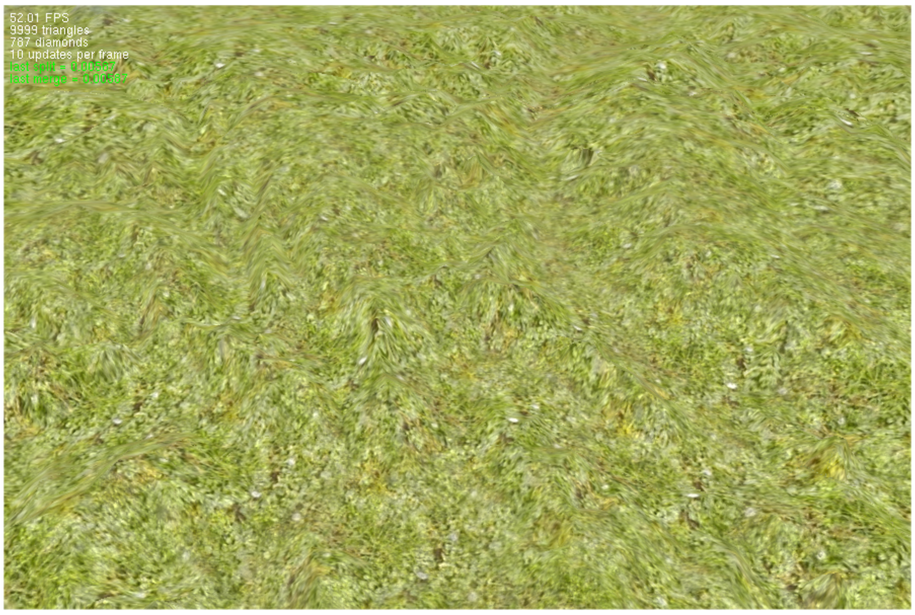
The normal map we use is at the same resolution as the heightmap. In fact this does not need to be the case, but higher resolution will provide better quality lighting. In some cases this may not be suitable to use a high detail normal map as it may use too much video memory. Our RGB normal map at 1024×1024 resolution (this depends on the heightmap, but all the heightmaps we used were at this resolution) will use 3MB of video memory without compression.

4.3 Slope Based Texture Mapping

The effect of slope based texturing can be seen by looking at the difference between Figure 9 and Figure 10.

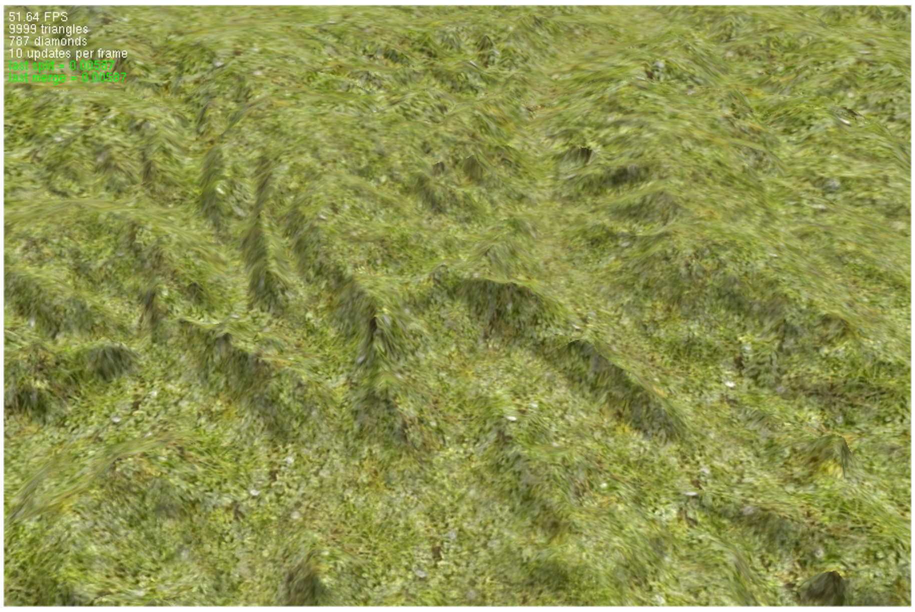
We can see that slope based texturing provides a very interesting and realistic looking effect on terrains, especially when the slope is large.

Because of the simplicity of the test, the performance difference is very negligible. But for our implementation, we require the normal map which uses extra texture memory. Since we are already using this for lighting however this is not much of a problem.



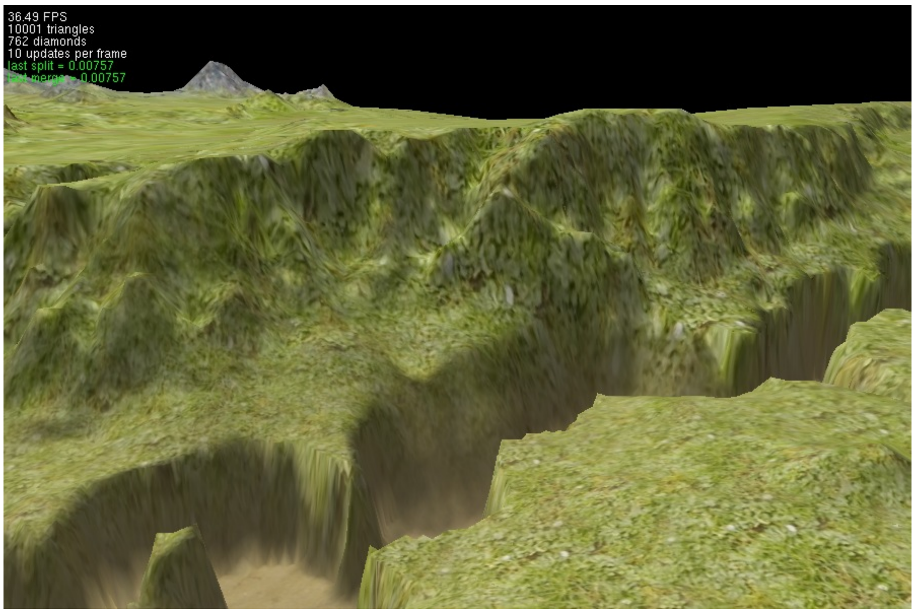
```
52.01 FPS
9999 triangles
787 diamonds
10 updates per frame
last split = 0.00587
last merge = 0.00587
```

Figure 7: A part of a rendered terrain before lighting.



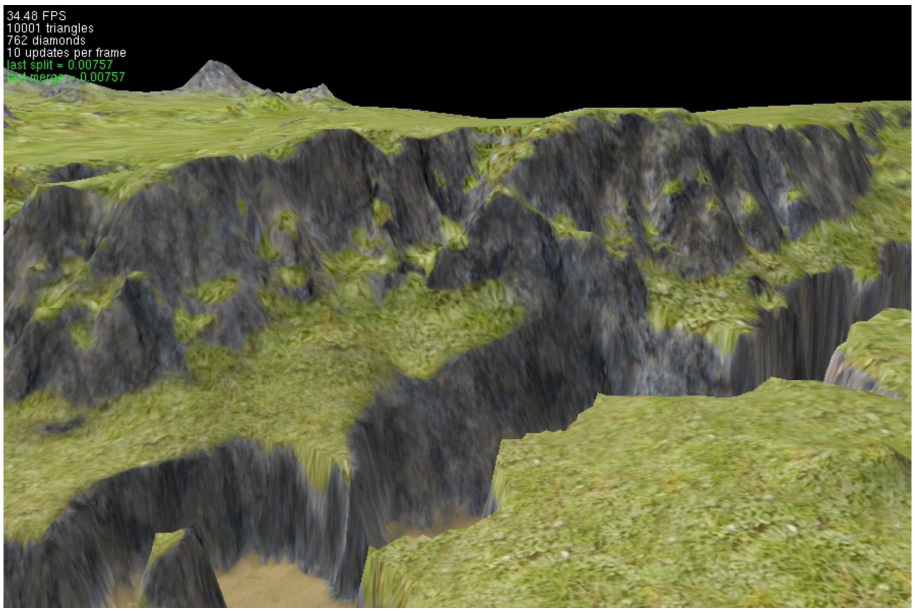
```
51.64 FPS
9999 triangles
787 diamonds
10 updates per frame
last split = 0.00587
last merge = 0.00587
```

Figure 8: The same part of a rendered terrain after lighting. Notice the extra details that were not visible without lighting.



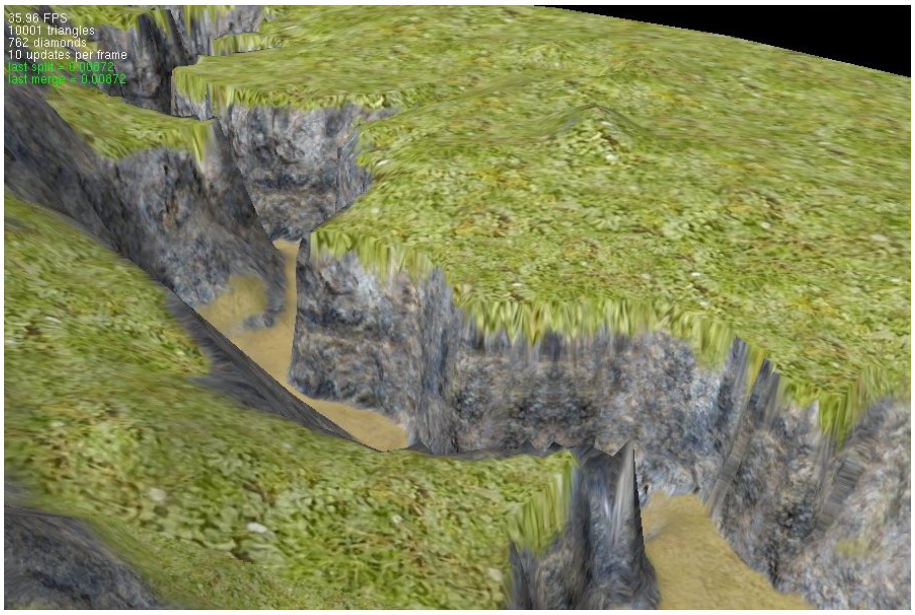
```
36.49 FPS
10001 triangles
762 diamonds
10 updates per frame
last split = 0.00757
last merge = 0.00757
```

Figure 9: Before slope based texturing.



```
34.48 FPS
10001 triangles
762 diamonds
10 updates per frame
last split = 0.00757
last merge = 0.00757
```

Figure 10: After slope based texturing.



```
35.96 FPS
10001 triangles
762 diamonds
10 updates per frame
last split = 0.00872
last merge = 0.00872
```

Figure 11: Using triplanar texturing to remove texture stretching artifacts.

4.4 Triplanar Texturing

In Figure 6, we saw what happens when we use simple planar projection to texture the terrain on large slopes. Using triplanar texturing, we get the result as in Figure 11. As we can see it improves the stretching situation considerably in these cases.

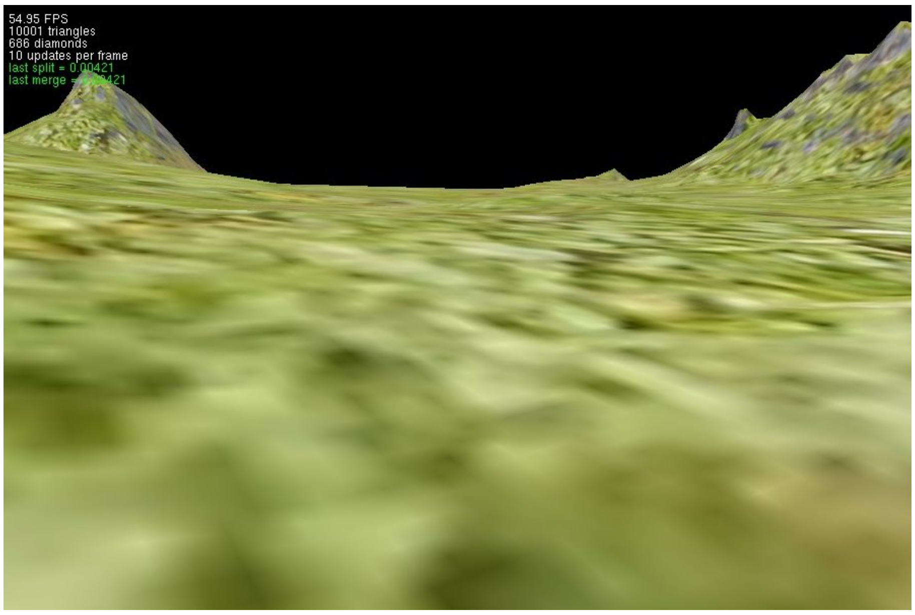
Because triplanar texturing will sometimes blend between planar projections, it requires up to three times more texture fetches. In our case this decreased our performance considerably, dropping the frame rate much more than the other methods we use.

4.5 Level of Detail Texturing

Figure 12 shows one of the problems we solve with level of detail texturing. When the viewer is very close to the terrain, the texture is low detail and looks blurry. After applying level of detail texturing, we get the result in Figure 13. Now that the texture repetitions are based on the distance from the camera to the terrain, the repetitions have been increased, giving adequate surface detail.

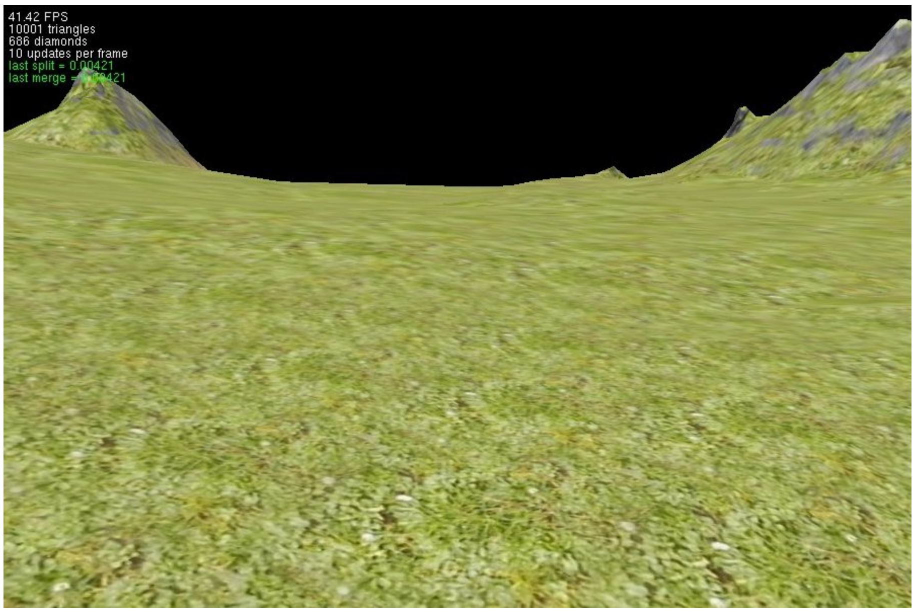
Figure 14 shows the other problem that level of detail will fix. At a large distance from the terrain, we can see a very obvious tiling pattern in the texture. Figure 15 shows the result after level of detail texturing. The texture repetition has been reduced for areas at a large distance.

However, there are some problems with our implementation of level of detail texturing. In particular, modifying texture repetitions only can cause some strange visual effects. Because we are using discrete LOD, we get a variant of the 'popping' problem. Although we blend between LOD regions there is a distinct, visible change between each level.



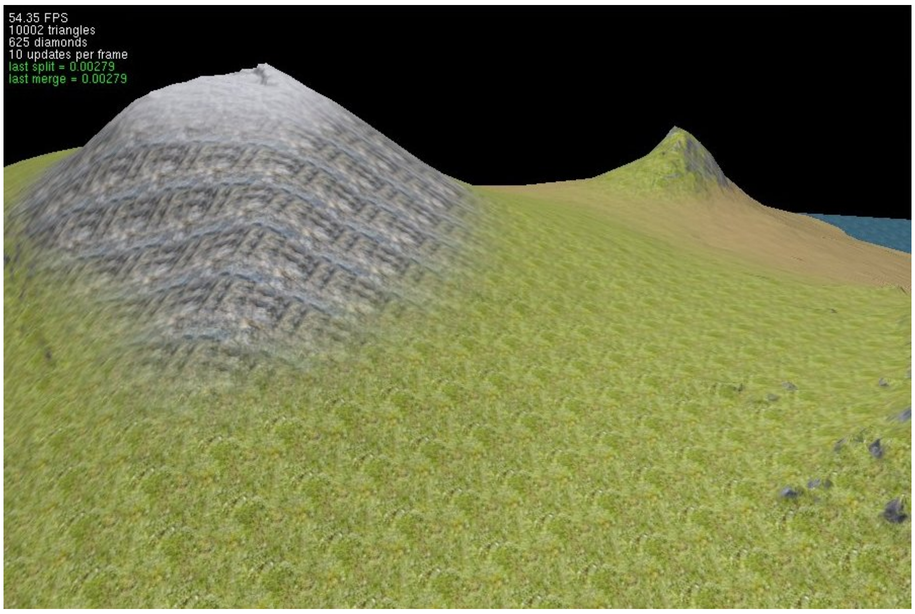
```
54.95 FPS
10001 triangles
686 diamonds
10 updates per frame
last split = 0.00421
last merge = 0.00421
```

Figure 12: Close to terrain with a medium texture repetition.



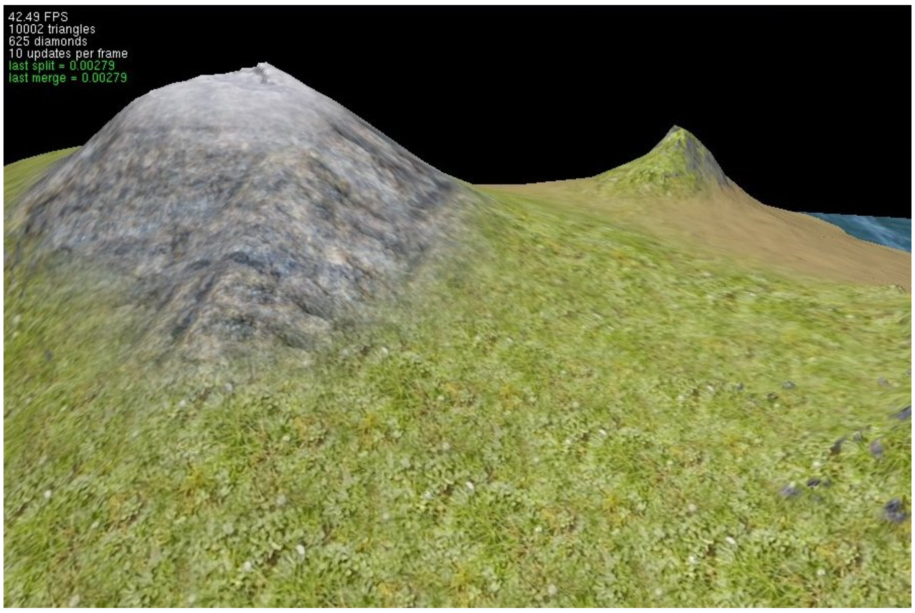
```
41.42 FPS
10001 triangles
686 diamonds
10 updates per frame
last split = 0.00421
last merge = 0.00421
```

Figure 13: Close to terrain using level of detail texturing.



```
54.35 FPS
10002 triangles
625 diamonds
10 updates per frame
last split = 0.00279
last merge = 0.00279
```

Figure 14: Distant from terrain with a medium texture repetition.



```
42.49 FPS
10002 triangles
625 diamonds
10 updates per frame
last split = 0.00279
last merge = 0.00279
```

Figure 15: Distant from terrain using level of detail texturing

Also, if you imagine falling towards the terrain, you would imagine that details on the terrain should get larger and larger. When the level of detail increases however, details on the terrain suddenly become smaller again. This can make it difficult to determine how close the terrain actually is.

Another side-effect to this is with the water animation. Since we add a constant to the texture coordinates each frame, the animation speed will be related to the texture repetitions. This means when blending occurs between levels of detail we are able to clearly see each level of detail since they are moving at different speeds. This gives a layering effect as the higher repetition level looks like it is below the lower repetition value. This is interesting, but does not look particularly realistic.

Some of these problems could be improved. To improve the 'popping' problem we could reduce the number of levels. Seven levels is far more than is necessary, three or four might have been enough.

Rather than changing repetition alone, we propose another method. A number of textures of the same resolution could be used, but with varying levels of detail. For near parts of the terrain, a texture with small surface details could be used, but for distant parts a texture that represents a larger area of ground could be used. If they are the same resolution, the close textures will need to be much denser than the distant ones.

5 Discussion and Conclusions

5.1 Discussion

After implementing these techniques there are a couple of points we will discuss. The first is that after this experimentation, which of the methods would we actually use for current technology to render a realistic looking terrain procedurally? The second point of discussion will be on shader development in general.

5.1.1 Techniques

If we were developing a terrain renderer using procedural texturing that was to be up to release standards, we would certainly use height based texture mapping. It is a simple test and gives interesting texture variation. It would be better if it could be tailored to the particular area it is meant to portray if this is possible. For example, it does not make much sense to use a snow texture for a mountain in Hawaii.

Per pixel lighting using a normal map definitely improves the surface detail of the terrain so we would use this also. Although perhaps the normals could be calculated in the shader. This may depend on the particular application however. Using extra texture memory may not be an issue.

Slope based texturing also seemed to work very effectively in the cases we saw. This is simple to implement so we would definitely use this technique.

Triplanar texturing is an interesting technique, but we would suggest only using it if the application requires it. Texture stretching is usually only an issue for large slopes such as a cliff face, so if these features are not present it may not be worth implementing an expensive feature like this. There is also the issue of slight blurring on other surfaces, but this may be possible to remove with more tweaking.

Level of detail texturing is also interesting, but there are more likely better solutions available. Perhaps a modified version of this implementation would work much better, as currently the blending between regions is too noticeable when the camera is moving. As technology improves further, it may be more effective to use very high resolution textures with texture filtering. The detail map technique would be another simple way of avoiding low detail textures, but the detail map should be specialised to the type of texture it is applied to, unlike in our case.

5.1.2 Shader Development

There is one important point we learned when developing with shaders. Because GPUs are in a rapid state of development, each generation is capable of more than the last. It is very difficult to know what a particular GPU will be capable of.

In our case, we developed our shaders on Nvidia GeForce 8000 series graphics cards. About half way through our development we found that the shaders no longer worked on older ATi graphics cards. Later, still we found it no longer worked on even GeForce 7000 series graphics cards. This meant we had inadvertently used features specific to the card we were developing on. We thought it might have been vertex textures, but this was not the only problem, so we gave up on backwards compatibility.

In a research setting, this is not a big problem since we are looking to the future. However, for a shader developer we strongly recommend that the shaders be developed on the lowest specification GPU that the shader is to be run on. We probably recommend that it be primarily developed on an ATi graphics card if cross compatibility is required as the shader compiler will throw errors more often while the Nvidia compiler will ignore some errors. The shaders should still be tested on both sets of hardware (and other manufacturers if supported) to ensure that it works as expected.

5.2 Future Work

Each of the techniques we presented in this paper could probably be improved or configured for a specific application. Other than that there are a few specific areas to focus on.

5.2.1 Texture Atlases

Texture Atlases are simply a collection of textures combined into one larger texture. Andersson described their use specifically for their sparse texture masks. In our project, texture atlases could have been used since our textures were quite small (256 pixels $\times$ 256 pixels).This would have the advantage of reducing the number of texture units required, allowing a greater number of total textures to be used simultaneously.

There are some problems with texture atlases related to bilinear texture filtering as one textures image can ‘bleed’ into another. This can be solved by leaving some space between the textures.

5.2.2 Texture Stretching Solution

Texture stretching is a fairly large problem for procedurally texturing a terrain from a heightmap. This is especially true when there is no chance for an artist to intervene and specify texture coordinates manually.

McGuire and Whitson’s indirection mapping technique is promising, but requires a significant preprocessing element and causes artifacts of its own, limiting its usefulness.

In this project we attempted to solve this problem also, firstly using triplanar texturing. This worked well for some highly sloped areas, but caused severe blurring on less sloped areas. We then attempted to find a new algorithm but were not able to find one in the time available.

5.2.3 Noise

Other papers have described how they apply noise in the texturing process to give a more natural look to the texture boundaries. This could have been applied in this project at any stage where blending is used. This includes height based texturing, slope based texturing, level of detail texturing and triplanar texturing.

5.2.4 Shadows from Occlusion

To improve the realism of the terrain lighting, shadows will need to be implemented. This involves checking to see if the part of the terrain is actually within line of sight with the light source. This could be done by drawing a ray from each pixel towards the light source and checking if the ray intersects any part of the terrain. This may have a simple solution since polygon intersection tests could be replaced by checking the heights along a 2D line across the heightmap.

5.2.5 Reduction of Blending

A common problem with the majority of the techniques presented in this paper is that of harsh boundaries. The basic solution to this problem is to blend between the boundaries to make the change less noticable. Blending, however, is really just another name for blurring. When textures are blended together, the result is an average (or combination) of the two. This reduces the sharpness of the overall image, especially when blending is used excessively. A more obvious example of this would be with our implementation of triplanar texturing. Little progress has been made in this project to minimise the amount of blending required, but in the case of triplanar texturing, any blending can affect large regions of the terrain.

It might be possible for textures to be combined in a way that preserves the original sharpness of both images. This might involve using boundary textures between each region which would be created by artists. Using these textures in combination with the original two, a texture synthesis technique could be applied to produce a minimum error boundary between each texture. This would produce a sharper image when combining textures.

5.2.6 Local Features

One problem with procedural texturing is that we are generally not taking the type of terrain into account when texturing. The classic case for this is lakes and rivers. Because they are above sea level, they will be textured with a ground texture rather than with water. Perhaps some form of local feature extraction could be performed to determine these cases (especially in the case of a lake). An artificial intelligence system could be used to train a classifier with features that indicate a lake or other terrain feature and texture these features differently.

5.2.7 Performance Analysis

A detailed performance analysis of some of these techniques would be particularly useful. Especially if they were performed using a GPU based terrain rendering method as there would be much more computation on the GPU and the bottleneck would be clear. Using ROAM there is a large amount of CPU usage still so the bottleneck might be there sometimes.

5.3 Conclusion

In this project we have investigated various methods for procedurally texturing and lighting terrain based on heightmaps using the GPU while using the ROAM terrain rendering algorithm.

We implemented a shader based implementation of height based texturing where different textures are applied and blended together depending on the height of the terrain. We then used the slope of the terrain to blend another texture in for large slopes or cliffsides. These methods are relatively simple and worked effectively.

We performed per pixel lighting using Phong shading with a normal map created from the terrain heightmap. This was a fairly simple to implement, yet effective lighting solution.

We investigated triplanar texturing as a solution to texture stretching on large slopes and cliffsides due to the simplified planar projection. It seems to be fairly effective, but could cause performance problems in some cases for little return.

We attempted to use a form of level of detail for texturing to set the texture resolution based on the distance from the camera to the point on the terrain. This worked well, but the discrete nature of this LOD created effects analagous to the 'popping' problem for geometry LOD.

There is more work to be done in further investigating and reporting on these techniques as well as finding new algorithms to solve some of the problems such as texture stretching.

References

[1] J. Kessenich, D. Baldwin, and R. Rost, “The OpenGL Shading Language,” 2006. http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf.

[2] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, and M. Mineev-Weinstein, “ROAMing terrain: Real-time Optimally Adapting Meshes,” *VIS*, vol. 00, p. 81, 1997.

[3] R. Pajarola and E. Gobbetti, “Survey of semi-regular multiresolution models for interactive terrain rendering,” *Vis. Comput.*, vol. 23, no. 8, pp. 583–605, 2007.

[4] F. Losasso and H. Hoppe, “Geometry clipmaps: terrain rendering using nested regular grids,” in *SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers*, (New York, NY, USA), pp. 769–776, ACM, 2004.

[5] C. Bloom, “Terrain texture compositing by blending in the frame-buffer,” 2000. http://www.cbloom.com/3d/techdocs/splatting.txt.

[6] T. Jenks, “Terrain texture blending on a programmable GPU,” 2005. http://www.jenkz.org/articles/terraintexture.htm.

[7] J. Andersson, “Terrain rendering in frostbite using procedural shader splatting,” in *SIGGRAPH ’07: ACM SIGGRAPH 2007 courses*, (San Diego, California), pp. 38–58, ACM, 2007.

[8] J. Stam, “Aperiodic texture mapping,” tech. rep., European Research Consortium for Informatics and Mathematics (ERCIM, 1997.

[9] M. F. Cohen, J. Shade, S. Hiller, and O. Deussen, “Wang tiles for image and texture generation,” *ACM Transactions on Graphics*, vol. 22, pp. 287–294, 2003.

[10] L.-Y. Wei, “Tile-based texture mapping on graphics hardware,” in *SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware*, (Grenoble, France), pp. 55–63, ACM, 2004.

[11] H. Wang, “Proving theorems by pattern recognition I,” *Commun. ACM*, vol. 3, no. 4, pp. 220–234, 1960.

[12] H. Wang, “Games, logic and computers,” *Scientific American*, vol. 213, no. 5, p. 98, 1965.

[13] A. Lagae and P. Dutré, “An alternative for wang tiles: colored edges versus colored corners,” *ACM Trans. Graph.*, vol. 25, no. 4, pp. 1442–1459, 2006.

[14] R. M. Okamoto, F. L. de Mello, and C. Esperança, “Texture management in view dependent application for large 3d terrain visualization,” in *SpringSim ’08: Proceedings of the 2008 Spring simulation multiconference*, (New York, NY, USA), pp. 641–647, ACM, 2008.

[15] Q. Li, C. Zhao, Q. Zhang, W. Yue, and G. Wang, “Region-based artificial terrain texture generation,” *Virtual Reality*, vol. 4563/2007, pp. 97–103, 2007.

[16] M. Mittring and C. GmbH, “Advanced virtual texture topics,” in *SIGGRAPH ’08: ACM SIGGRAPH 2008 classes*, (New York, NY, USA), pp. 23–51, ACM, 2008.

[17] R. Geiss and M. Thompson, “Cascades by NVIDIA,” in *Game Developers Conference*, 2007.

[18] M. McGuire and K. Whitson, “Indirection mapping for quasi-conformal relief texturing,” in *SI3D ’08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, (Redwood City, California), pp. 191–198, ACM, 2008.

Appendices

A GLSL Vertex Shader Code Listing

```
uniform float Theight;
uniform int Tlength, Twidth;
//uniform sampler2D normalMap;
uniform float texRep, detailRep;
uniform bool lighting;
uniform bool textureLOD;
//varying float h;
varying vec3 lightDir;
varying float distance;
varying float lodLevel;
varying float smoothLod;

varying float test;

float rep1;
float rep2;

void calculateTexLOD()
{
float currentCutoff = 15.0;
  float finalCutoff = 15.0;
int lod = 1;
int i = 6;
  for (int j = 1; j < 7; j++)
  {
   currentCutoff *= 2.0;
   lod = step(currentCutoff,distance);
     //finding the first cutoff that is larger than
    //the current distance
    if ((lod == 0) && (i==6))
   {
    i = j;
    finalCutoff = currentCutoff;
   }
  }
lodLevel = i;
smoothLod = smoothstep(finalCutoff/2.0,finalCutoff,distance);
//PROBLEM: if i changes across one polygon, texture repetition
//goes weird...need to somehow relegate this situation to the
//fragment shader...?
rep1 = 10*pow(0.5,i-2.0);
  rep2 = rep1/2.0;
test = i;
}
```

```
void main()
{
vec3 vVertex = vec3(gl_ModelViewMatrix * gl_Vertex);
distance = length(vVertex);

if (textureLOD) calculateTexLOD();
else
{
rep1 = 1;
rep2 = 1;
}

//Lighting
if (lighting)
lightDir = vec3(gl_LightSource[0].position.xyz - vVertex);

//Texture Coordinates
float x = gl_Vertex.x/Twidth;
float z = gl_Vertex.z/Tlength;
float y = gl_Vertex.y / Theight;
//h = y;

//height width ratio
float ratio = Theight/Twidth;
y *= ratio;

float texX, texZ;
float texX2, texZ2;
float texX3, texZ3;

float texX4, texZ4;

//normal projection. use x and z to sample texture
texX = x*texRep*rep1;
texZ = z*texRep*rep1;

texX4 = x*texRep*rep2;
texZ4 = z*texRep*rep2;

// projection2. x is largest normal component
// so use z and y to sample texture
texX2 = texZ;
texZ2 = y*texRep;

// projection3. z is largest normal component
// so use x and y to sample texture
texX3 = texX;
texZ3 = texZ2; //y

gl_TexCoord[0] = vec4(texX,texZ,0,0);  //first projection
gl_TexCoord[1] = vec4(texX2,texZ2,0,0);  //second projection
gl_TexCoord[2] = vec4(texX3,texZ3,0,0);  //third projection

//detail proj1
gl_TexCoord[3] = vec4(x*detailRep,z*detailRep,0,0);
//detail proj2
gl_TexCoord[4] = vec4(texX2/texRep*detailRep,texZ2/
texRep*detailRep,0,0);
//detail proj3
gl_TexCoord[5] = vec4(texX3/texRep*detailRep,texZ3/
texRep*detailRep,0,0);

gl_TexCoord[6] = vec4(x,z,0,0); //normal map

gl_TexCoord[7] = vec4(texX4,texZ4,0,0);

gl_Position = ftransform();
}
```

B GLSL Fragment Shader Code Listing

```
uniform sampler2D mytex[6];
uniform sampler2D normalMap;
uniform float levelRatio;
uniform float slopeCutoff;
uniform float triplanarCutoff;
uniform bool lighting;
uniform bool texturing;
uniform bool slopeTexturing;
uniform bool normalTexture;
uniform bool useDetailMap;
uniform bool triplanarTexturing;
uniform bool textureLOD;
uniform int frame;
uniform float texRep;
//varying float h;
varying vec3 lightDir;
varying float distance;
varying float smoothLod;

varying float test;

varying float lodLevel;

#define LODS 7

#define INSANE_DETAIL 7
```

```
#define EXTREME_DETAIL 6
#define REALLY_HIGH_DETAIL 5
#define HIGH_DETAIL 4
#define MEDIUM_HIGH_DETAIL 3
#define MEDIUM_DETAIL 2
#define LOW_DETAIL 1

#define WATER_SPEED 0.00048828125 // 1/2048

const int noLevs = 5;
float rep1 = texRep/10.0;
float rep2 = rep1/2.0;
float[7] lodCutoffs =
float[](0.0, 30.0, 60.0, 120.0, 240.0, 480.0, 960.0);

// Find the terrain level based on height
// Warning: multiple returns
int findTerrainLevel(float height)
{
if (height <= 0.005) return 0;
//multiple returns, loop may break early
for (int i = 1; i < noLevs; i++)
{
if (height <= levelRatio * i) return i;
}
return noLevs;
}

int getLevelOfDetail()
{
if (distance < lodCutoffs[1]) return INSANE_DETAIL;
else if (distance < lodCutoffs[2]) return EXTREME_DETAIL;
else if (distance < lodCutoffs[3]) return REALLY_HIGH_DETAIL;
else if (distance < lodCutoffs[4]) return HIGH_DETAIL;
else if (distance < lodCutoffs[5]) return MEDIUM_HIGH_DETAIL;
else if (distance < lodCutoffs[6]) return MEDIUM_DETAIL;
else return LOW_DETAIL;
}

int round(float num)
{
float result = floor(num);
float res1 = num - result;
if (res1 >= 0.5) result = ceil(num);
return result;
}

vec4 getLevelOfDetailColour(sampler2D tex)
{
vec4 colour1;
```

```
vec4 colour2;
float thisSmoothLod = smoothLod;
if (abs(lodLevel-round(lodLevel))>0.000001)
{
float cut1 = 30*pow(2.0,floor(lodLevel)-2.0);
float cut2 = cut1*2.0;
thisSmoothLod = smoothstep(cut1,cut2,distance);
float rep1 = 10*pow(0.5,floor(lodLevel)-2.0);
     float rep2 = rep1/2.0;
colour1 = texture2D(tex, gl_TexCoord[6].st*rep1*texRep);
colour2 = texture2D(tex, gl_TexCoord[6].st*rep2*texRep);
}
else
{
colour1 = texture2D(tex, gl_TexCoord[0].st);
colour2 = texture2D(tex, gl_TexCoord[7].st);
}

int i = lodLevel;

return mix(colour1,colour2,thisSmoothLod);
}

// Calculates lighting for this fragment given
// the normal and lightDirection (global)
// Returns the shade value (greyscale)
vec4 shadow(vec4 normalFromMap)
{
vec3 normal = gl_NormalMatrix * normalFromMap.xyz;
vec3 N = normalize(normal);
vec3 L = normalize(lightDir);
float lambertTerm = dot(N,L);
vec4 shade = vec4(0.4,0.4,0.4,1.0);
if (lambertTerm > 0.0) shade+= lambertTerm*vec4(0.6,0.6,0.6,1);
return shade;
}

// Applies a given colour to highly sloped areas and blends with
// current colour for
// "in between" slope values. Returns the resultant colour.
vec4 applySlopeTexturing(vec4 currentColour, vec4 slopeColour,
float slope, int level)
{
vec4 finalColour = currentColour;
//don't apply slope texturing for top level (assuming heavy snow)
// or bottom level
if ((level < noLevs-1) && (level > 0))
{
float difference = smoothstep(slopeCutoff/1.25,
slopeCutoff, slope);
finalColour = mix(slopeColour, currentColour, difference);
}
return finalColour;
}

// Changes the way textures are sampled based on normal
// direction.Normals with a large(ish) z component
// and/or x component will use a different combination
// of samples as calculated in the vertex shader.
// Sluggish. Requires many texture samples (up to 3
// times the amount)
vec4[noLevs] applyTriplanarTexturing(vec4 normal,
bool useDetMap, int lod)
{
float absX = abs(normal.x);
float absY = abs(normal.y);
float absZ = abs(normal.z);

vec4 textures[noLevs];
for (int i = 0; i < noLevs; i++)
{
textures[i] = vec4(0,0,0,0);
}
vec4 detail = vec4(1,1,1,1);
if (useDetMap) detail = vec4(0,0,0,0);

if ((absZ > triplanarCutoff) && (absZ > absX))
{
float difference;
difference = smoothstep(triplanarCutoff,0.8,absZ);
if (useDetMap) detail += texture2D(mytex[noLevs],
gl_TexCoord[5].st)*difference;
for (int i = 0; i < noLevs; i++)
{
textures[i] += texture2D(mytex[i],
gl_TexCoord[2].st*rep1)*difference;
}
}
else if (absX > triplanarCutoff)
{
float difference;
difference = smoothstep(triplanarCutoff,0.8,absX);
if (useDetMap) detail += texture2D(mytex[noLevs],
gl_TexCoord[4].st)*difference;
for (int i = 0; i < noLevs; i++)
{
textures[i] += texture2D(mytex[i],
gl_TexCoord[1].st*rep1)*difference;
}
}
```

```
  //alpha should be the same for all textures at this stage
float difference = 1.0-textures[0].a;
if (useDetMap) detail += texture2D(mytex[noLevs],
gl_TexCoord[3].st)*difference;
textures[0] += texture2D(mytex[0], gl_TexCoord[0].st+
WATER_SPEED*frame)*difference;
if (useDetMap) detail*=2.0;
for (int i = 1; i < 4; i++)
{
textures[i] += texture2D(mytex[i],
gl_TexCoord[0].st*rep1)*difference*detail;
}
textures[4] += texture2D(mytex[4],
gl_TexCoord[0].st*rep1)*difference;

return textures;
}

// Method: Calculate texture colours for every texture (!!)
// which includes any detail and any slope based texture
// sampling. Find the current height level and apply the
// appropriate colour/s, blending as necessary. Apply a
// slope based texture and finally calculate and apply
// lighting.
//
// Possible improvement: Only calculate texture samples
// that are required for the current height level.
void main()
{
float difference;
int lod = getLevelOfDetail();

bool useDetMap = ((useDetailMap) && (lod >= HIGH_DETAIL));

vec4 normals = texture2D(normalMap, gl_TexCoord[6].st);

float h = normals.a;
int level = findTerrainLevel(h);

normals = normals * 2.0 - 1.0; //changing [0, 1] to [-1, 1]

vec4 textures[noLevs];
vec4 detail = vec4(1,1,1,1);
if ((triplanarTexturing)&&(lod>=MEDIUM_DETAIL))
textures = applyTriplanarTexturing(normals,useDetMap, LODS-lod);
else
{
if (useDetMap)
detail = texture2D(mytex[noLevs], gl_TexCoord[3].st)*2.0;
//animated water
textures[0] = texture2D(mytex[0], gl_TexCoord[0].st+
WATER_SPEED*frame);
for (int i = 1; i < noLevs-1; i++)
{
textures[i] = getLevelOfDetailColour(mytex[i])*detail;
}
textures[noLevs-1]=getLevelOfDetailColour(mytex[noLevs-1]);
}

vec4 colour = vec4(1,1,1,1);
vec4 final_colour;

//apply the appropriate colour textures
//difference is the ratio of a point's height within its
//height level
if (texturing)
{
if (level == 0) colour = textures[0];
else if (level == 1) colour = textures[1];
else if (level < noLevs)
{
difference = smoothstep(levelRatio*(level-0.75),
levelRatio*(level-0.25), h);
colour = mix(textures[level-1], textures[level], difference);
}
else colour = textures[level-1];
}

if (slopeTexturing)
final_colour = applySlopeTexturing(colour,
textures[3],normals.y, level);
else
final_colour = colour;

if (lighting) final_colour *= shadow(normals);

if (normalTexture) final_colour = normals;

gl_FragColor = final_colour;
}
```