Procedural Planetary Landscapes with Continuous Level of Detail

- Does the moon exist only when someone is looking at it? - Albert Einstein 1950

Thomas A. Grønneløv*   Axel E. Jensen†

Figure 1: Planetary landscape with continuous level of detail, rendered real-time

Abstract

We will present a real-time method for procedurally generating huge planetary landscapes with continuous level of detail. This approach enables us to produce interesting planets with a small or non existing pre-generated dataset, which in turn could be used to visualize an endless number of different planets.

Where previous work in landscape generation have generally been purely procedural or purely design, we have devised a method which allows for a seamless integration of design into the computer generated world.

Another novelty is the decoupling of the mesh optimization from the rendering. While a high frame-rate is a requirement for fast and smooth animation, the mesh optimization can run in the background at a slower pace. We have implemented a system with different update frequencies for rendering and mesh optimization, to let us prioritize the different tasks, and to distribute the workload on multiple processors.

A method to generate natural looking river systems in the procedural generated terrain is explored and implemented. While we found that actual real-time procedural river generation was very difficult, one could combine a fast preprocessing step, with correct river flow calculations, which could later be placed inside the terrain.

- Seamless blend of design and pure procedurally generated terraincolor
- Decoupled mesh optimization and rendering to better utilize multi core processors
- Correct river flow calculation in a procedural landscape.

*e-mail:Tag@Greenleaf.Dk

†e-mail: Axel@Eystein.Dk

Contents

1 Reading guide **1**

2 Glossary / notation **1**

3 Introduction **1**

4 Previous work **2**

5 Level of Detail **3**
- 5.1 Introduction to Level of Detail 3
- 5.2 Different categories of LOD 3
  - 5.2.1 Discrete LOD 3
  - 5.2.2 Continuous LOD 3
  - 5.2.3 Detail defined LOD 4
  - 5.2.4 View Dependent LOD 4
- 5.3 Motivations for using LOD, visual quality vs interactivity 4
- 5.4 An example 4

6 ROAM **5**
- 6.1 ROAM from 10 000 feet 5
  - 6.1.1 Basic ROAM geometry 5
  - 6.1.2 Altering the mesh 5
- 6.2 Prioritizing the triangles 7
- 6.3 Boundaries and ROAMing a sphere 7
- 6.4 A ROAMing example 7
  - 6.4.1 Prioritized queues 7
- 6.5 Memory usage 8
- 6.6 Rendering a ROAM 9
  - 6.6.1 Triangle ordering and format 9
  - 6.6.2 A winding way through the mesh 9
- 6.7 Decoupling rendering and optimization 10
- 6.8 Variations of ROAM 11
- 6.9 Ordinary ROAM error metric 11
  - 6.9.1 Alternative ROAM error metrics 11
  - 6.9.2 The uncertainty of lower subdivision levels 12

7 Bounding volumes **12**
- 7.1 Optimal bounding volume 12
- 7.2 Breaking the bounds, and expanding them 13

8 Z-buffer accuracy **13**
- 8.1 Squeezing the planes 14
- 8.2 Optimal clipping plane placement 15
- 8.3 Calculating the clipping points 15

9 Procedural generation of geometry and terrain **16**
- 9.1 Motivating use of procedural methods 16
- 9.2 An example of procedural generation 17
- 9.3 Advantages and disadvantages of procedural generation 17
- 9.4 Future perspective 17
- 9.5 Use of procedural generation in this paper 17
- 9.6 Fractal generation 17
  - 9.6.1 Midpoint displacement 18
  - 9.6.2 Fault line 19
  - 9.6.3 Multi fractals 20
  - 9.6.4 Ridged fractals 20
  - 9.6.5 Perlin Noise 22
  - 9.6.6 Coordinate system 23
  - 9.6.7 The noise conclusion 23
- 9.7 Not so random randomness 23
  - 9.7.1 Randomness from hashing 23

10 Are landscapes really fractal? **24**

11 Design and procedural generation **24**
- 11.1 The blending function 25
- 11.2 Midpoint displacement and design blending 26
- 11.3 2D area design 26
  - 11.3.1 Design space 27
  - 11.3.2 Sampling 27
  - 11.3.3 Blending 27
- 11.4 Design and visibility 28
- 11.5 Blending in the vertex shader 30
- 11.6 Distortion from projection 31
- 11.7 Interactive real-time design 31

12 Landscape features **32**
- 12.1 Dry land 32
- 12.2 Oceans and lakes 32
- 12.3 River systems 33
  - 12.3.1 1D path following profile 33
  - 12.3.2 Dropping water 33
- 12.4 Vegetation 33
- 12.5 Erosion 34

13 Calculating the natural flow of a river **34**
- 13.1 Moving point of interest 35
- 13.2 From 3D river to 2D design 35
- 13.3 Lakes 37
- 13.4 Erosion 37
- 13.5 Dry inseas and filling the ocean 37
- 13.6 Visualization of rivers and lakes 38
- 13.7 The river conclusion 38

14 Visualization **39**
- 14.1 Lighting 39
  - 14.1.1 Enter the shade 40
- 14.2 Terrain color 42
  - 14.2.1 Pixel shader to color the landscape 45
  - 14.2.2 Textures to color the landscape 46

15 View frustum culling **47**

16 Various error metrics **48**
- 16.1 Error measures related to geometry 49
  - 16.1.1 Surface layers 49
  - 16.1.2 Silhouettes 50
  - 16.1.3 Visual perception 50

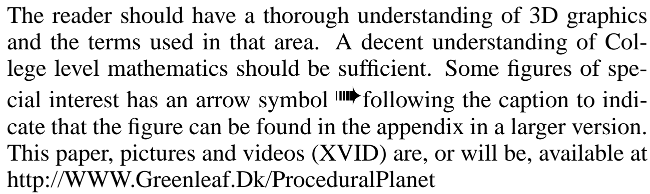
17 Test and analysis **51**
- 17.1 Frame coherence 51
- 17.2 Multiple threads 52
- 17.3 Fractal terrain generating algorithm 52
- 17.4 Effect of different number of triangles 52
- 17.5 Visible triangles 53

18 Conclusion **54**

19 Future work **54**

A Enlarged figures **60**

1 Reading guide



The reader should have a thorough understanding of 3D graphics and the terms used in that area. A decent understanding of College level mathematics should be sufficient. Some figures of special interest has an arrow symbol ➠following the caption to indicate that the figure can be found in the appendix in a larger version. This paper, pictures and videos (XVID) are, or will be, available at http://WWW.Greenleaf.Dk/ProceduralPlanet

In the introduction, we motivate the use of procedural generation on a large scale, but provide no specifics. This section can be skipped by readers who are familiar with the subject.

In section 4 we mention previous work which are in some way related to this project. As this project has two major topics, level of detail and procedural generation, this section is split into two corresponding subsections. For readers unfamiliar with either or both topics, we suggest to skim this section, or skip it and return to it later.

A general introduction to level of detail is given in section 5 followed by section 6, which contains an in depth description of the ROAM algorithm. The latter sections details are not essential for the understanding the rest of the paper.

In section 9 we introduce the concept of procedural generation of terrain, followed by a discussion on fractal terrain, section 10, and in section 12 and section 13 we describe some elements which needs to be taken into consideration when producing realistic looking terrain.

The use of design and problems arising thereby are found in section 11.

Lastly this paper presents in section 17 results concerning level of detail and procedural techniques.

2 Glossary / notation

We use few non standard notations, but a few may need explanation.

Decimal separator We use the SI decimal separators. The thousand separator is a space while the decimal separator is a point. One million and a half will therefore be 1 000 000.5.

Modulus Modulus is typed out as the word ”mod”. 10 mod 7 is 3.

ROAM While ROAM is the name of an algorithm, we also use it as a word for a mesh optimized by the algorithm.

LOD Meaning level of detail for an object or more generally for a scene. At times used for an algorithm altering an objects level of detail.

3 Introduction

Visualizing and generating large terrain, even planets, has many interesting uses.

One field would be in computer games. The emergence of games in which many players interact in large virtual worlds would benefit from the option of easily developing new worlds or planets. Recently a type of games known as MMPORPG[1] has become very popular, and contain thousands of players and computer characters, but the worlds are still limited in size - though huge. In other games players are traveling between solar systems containing many planets. Giving these planets more detail than just a textured sphere, so one can actually land on them and look around, would certainly add to the game experience. In computer games highly realistic terrains are often desirable but at other times unrestricted environments are even more important.

Another use would be for simulators, such as like flight simulators and combat simulators. Both would benefit from large terrains rendered at interactive rates and with some level of realism. At times simulations need to reflect a situation from real life and this could be one motivation for having a system in which procedural generated terrain can be blended with designed or sampled terrain or terrain elements.

When visualizing terrain or entire planets to present geological or geographical informations, data is sampled at a specific level. Using a procedural method to ”fill in the gaps” where data is missing as a complementary method to interpolation between samples, to add extra detail, could improve the visualization at times.

If you want to visualize a planet the size of Earth with a resolution of one square meter per sample, it will take more than 500 trillion samples for the height values of the surface. That would amount to 1 855 Terra bytes if the samples are stored as 32 bit data.

It is therefore not possible, with the computers of today, to give a high resolution representation of a landscape of that size, if you rely on brute force.

Should you wish to have one or more planets with a varying terrain, it would be absolutely impossible to manually design all the landscapes with the required level of detail, and currently we do not have detailed height samples from many other planets.

The solution, which exists, is to use level of detail algorithms and procedurally generated landscapes. Both have shown useful for both arbitrarily large worlds and for quite realistic looking sceneries. Examples from the gaming world would be the brand new Spore from Firaxis [Arts 2006] and older games like Elite [Elite 1984]. Neither one generates realistic landscapes, but they do provide an almost endless game world.

Combining level of detail algorithms and procedural generated terrain run-time is still a fairly new topic. It has interesting perspectives as this combination eventually could be used to visualize large number of planets at the same time, where certain features are predesigned and others are not. It overcomes the tedious work of having to design all the detail. It would be adventurous to explore new worlds visited for the first time. It makes it possible to maintain a large number of planets, as minimal data is stored for each planet and more detail for planets is generated when needed. The level of detail algorithm ensures that only what is visible, and of current interest, is defined in detail.

Level of detail algorithms optimizes the use of resource for rendering scenes. It increases the effectives of the rendering process and

[1] Massive Multi Player Online Role Playing Game

thereby increases the visual quality of the images produced with limited resource. It is accomplished by generating details for objects based on how much those details contribute to the viewing experience.

Many examples of breathtaking procedural landscapes exists. For example in [Musgrave 2006] or [Terragen 2006]. Procedural generated terrain can look very realistic or they can look like things never seen before. As mentioned above, procedural landscapes lowers the database size considerably. Taken to the extreme, an entire planet can be based on a single number.

At times more explicit control over the generated terrain is needed. whether it be the general appearance of the terrain or how a certain part of the terrain precisely looks. For that, an interaction between the methods which generates the terrain and a designer is needed. In developing computer games designers use some sort of ”level design” tool box. In simulators data samples from the Earth might be presented.

This interaction between designed or sampled terrain and procedurally generated terrain is a interesting topic. It is somewhat in conflict with the motivation for using procedural generation, to lower database size, but on the other hand purely designed terrain has considerably database size and would benefit from replacing parts of the designed terrain with procedural methods, whenever possible.

The task In this paper we will discuss the theory behind the level of detail and procedural generation of terrain.

Different methods to generate terrain procedurally are presented and select among those methods. When the methods are implemented as a computer program, the landscape should be generated procedurally as needed for rendering and thus give a subjectively realistic looking landscape with details where they are most appreciated.

Should a designer have specific demands for a specific part of the landscape, it should be possible for that designer to influence the procedurally generated landscape to a degree so that the design can be integrated seamlessly into the world.

4 Previous work

We deal with a mix of procedural generation and level of detail, so we relate to previous work in both fields, though much of that work is either entirely about level of detail or entirely about procedural generation.

Procedural content generation Procedural techniques has been used for some time now. At first they where used to produce relative simple effects. For instance in calculating colors and simple textures. Procedural techniques were catching on and around 1980s they where explored in more detail and began to be more widely used.

B. B. Mandelbrot is known for his work on fractal geometry and how it relates to nature. His work, to mention a few, from 1977 [Mandelbrot 1977] and later in 1983 [Mandelbrot 1983] has been an inspiration to many trying to create landscapes. But B. B. Mandelbrot realized that fractal landscapes where lacking erosion. Later, in 1991 B. B. Mandelbrot and F. K. Musgrave published their article, [Musgrave and Mandelbrot 1991], on synthetic landscapes and how it almost becomes an artistic process to create these landscapes.

In 1985 K. Perlin introduced 3D textures [Perlin 1985]. 3D textures, also known as solid texturing, were used to create textures like wood, marble and similar elements often seen in nature. 3D textures were also used as animated 2D textures.

Some years later K. Perlin and E. Hoffert extended 3D textures in what they call hypertextures. In their article, [Perlin and Hoffert 1989], they describe methods to model effect like fur, fire, erosion and the flow of water.

Similar techniques are used to model the more general class of gaseous form. Like fire, as mentioned above, but also smoke clouds as described by W. T. Reeves [Reeves 1983].

Also in 1989 F. K. Musgrave et al. published an article, [Musgrave et al. 1989], in which they describe a new way to generate fractal landscape which look like landscapes subject to erosion, a feature which fractal landscape at that time was criticized for not possessing. They also present a couple of erosion models to further improve the effect of erosion on the procedurally generated terrain.

In 1994 [Ebert 2003] was first published and is a collection of procedural techniques used for texturing, modeling and animating. It covers the theoretical background and also goes into depth on implementing techniques. It seems that procedural techniques are used in increasing broader spectrum.

One of the coauthors of the above, F. K. Musgrave, is also the originator of MojoWorld, [Musgrave 2006],which is capable of producing impressive worlds. These worlds are generated off line and view dependent level of detail is also prepared off line.

Other interesting uses of procedural methods is seen in [Greuter et al. 2003], where large and complex cities are generated, even real time. Furniture and other contents for the 3D world can be generated and placed procedural and placed according to 2D floor plans [Wehowsky 2001]. Combining all these techniques could produce interesting sceneries, both indoor and outside.

Another interesting development in procedural techniques is seen in the use of graphics hardware. Having programmable graphics card add a whole new perspective to how and where geometry should be generated. Instead of transferring large and complex geometric models to the graphic card, it is now possible to just have procedural methods programmed into the graphic card and saving band limit[2].

Level of detail As early as 1976 James Clarke [Clark 1976] described the advantages of using level of detail (LOD). He described how an object in a scene could appear with different levels of detail depending on various parameters, such as the objects distance to the viewer, orientation etc.

During the early 1980’s flight simulators where quick to appreciate and adopt LOD. Choosing between models of building of different LOD depending on the distance to the viewer made it possible to have more realistic scenes in flight simulators [Cosman and Schumacker 1981], [Yan 1985]. At that time, automatic methods for simplifying models where not present, so they where made by hand.

Using different LOD in terrain was also found useful as described in 1993 J. S. Falby et al. [J. S. Falby and Mackey 1993]. Realizing that simplifying terrain down to a level at which the throughput of the graphical pipeline would satisfy interactivity results in too simple terrains. In their paper they describe a method where parts of the terrain is represented at different LOD and paged according to the viewer.

[2]This requires geometry shaders

This type of discrete LOD representation has the problem of visual popping when switching between representation of different LOD. Also special care needs to be taken between adjoining blocks of different LODs.

In 1995 P. Lindstrom et al. uses a similar technique in [LIndstrom et al. 95], but decisions on which terrain representation to use is based on more factors, like information about the viewers orientation to the terrain, the roughness of the terrain etc. They are also looking at a solution to fix cracks between adjacent patches of different resolutions, albeit not the most pretty solution.

One year later P. Lindstrom et al. [Lindstrom et al. 1996] came up with a different approach. Instead of using the above mentioned technique of choosing between representations at given LOD, they present a method where the appropriate LOD is computed and the representation is generated real time. This method uses a combined method of simplification of existing model and repolygonalization step, further reducing polygon count.

In 1996 H. Hoppe describes in his article, [Hoppe 1996], what he calls progressiv meshes. A method to store and transfer meshes. Progressive meshes are able to gradually refine from a simpler model to a more complex model. One motivation is streaming of data. Its possible to render a simple model and as data arrives, progressively refine the model and render it.

In 1997 M. Duchaineau et al., [Duchaineau et al. 1997], described a view dependent method where a mesh is optimized by splitting and merging triangles. This method accommodates frame coherence and it use of priority queues for split or merge is flexible. P. Lindstrom et al. [Lindstrom et al. 1996] takes on a similar approach in their view dependent LOD solution.

Same year H. Hoppe describes [Hoppe 1997] a view dependant LOD method, but different from the above, his method is not restricted to regular meshes, but works well triangulated irregular networks, TIN, which eventual could mean a saving in triangles.

As graphic hardwares throughput has rapidly increased over the last decade, refinement of previous solutions [Levenberg 2002], [Lindstrom and Pascucci 2002] and development of new are emerging [Losasso and Hoppe 2004]. In F. Losasso and H. Hoppes article, [Losasso and Hoppe 2004], they describe how the graphics hardware is used to store and update a grid situated around the viewer representing the terrain.

5 Level of Detail

In the following, Level of Detail will be described. Advantages and disadvantages for different approaches will be described, motivating choices made later.

5.1 Introduction to Level of Detail

In computer graphics there is a need for showing complex models highly detailed. As the models level of detail grows, it will affect the rendering process negatively. Rendering detailed models real time comes at the cost of a lower frame rate, so some means of technique needs to be used.

Realizing that models can be represented with different levels of detail, as shown in Figure 2, depending on their distance to the viewer, with little or no effect to the visual quality of the scene, could lower the scenes polygon count and therefore rendering the scene would be faster.

Lowering the level of detail for models more distant (or in other ways of lesser importance for the scene) makes sense because the low detailed model would be represented by an area in the rendered image which relates to its size and distance. So a model close to the viewer has a higher level of detail and fills a larger area of the rendered image, and a distant model has lower detail and fill a smaller area. An example of this is illustrated in Figure 3.

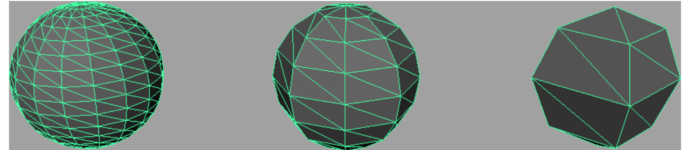


Figure 2: Showing same model with different Levels of Detail

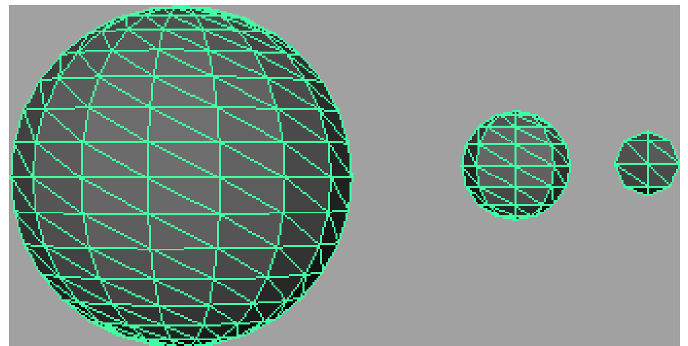


Figure 3: Showing models with different Levels of Detail at different distance

5.2 Different categories of LOD

There are different ways of calculating good level of detail for models. Each has its own pros and cons.

5.2.1 Discrete LOD

A model can be represented at different levels of detail. Having those different representations and choosing one of them depending on its position relative to the viewer is essentially what happens in discrete LOD. Making these models at different level of details can be done either manually or automatically. Making these models manually could be time consuming for designers, so today these models are mostly computer generated.

These models could be generated off-line and therefore their optimization would not be taking time away from the actual rendering. This pre-optimization could entail optimizing for different rendering hardware or organizing geometry in triangle strips. As these models are made without any knowledge of the scene or viewer, it is not possible to take into consideration whether parts of the model are hidden from the viewer and therefor could be of lower detail.

5.2.2 Continuous LOD

Instead of creating models with fixed LODs, continuous LOD uses a data structure for the model from which a representation at any LOD can be generated. This has the advantage over discrete LOD

that any model can be represented at an optimal LOD which utilizes resources more efficiently. Also, if a scene is changing over time, transition between one discrete model, in discrete LOD, to another may become apparent, which is not necessarily a problem for continuous LOD. However, even with continuous LOD it is possible to notice the transition in states, but they are more modest and techniques to minimize the effect exists.

Continuous LOD also supports the ability to continuously refine the scene over time by increasing LOD for the models. This again leads to the ability to stream models into the scene by starting with a coarse model and refining it as more information is received about the model.

When using continuous LOD run-time it might be costly to organize data in a way which is optimized for rendering.

If the detail level depends only on the distance between the observer and the geometry, then any region of the screen will contain the same number of polygons. only if the roughness is taken into consideration, the number can be different.

5.2.3 Detail defined LOD

Certain parts of a model can be more rough, for example in a landscape, a mountainous area is more varied or rough than a plain field. In order to keep this appearance in a simplified representation, i.e. a representation at lower LOD, it is necessary use more triangles on this rougher part of the scene. Some measure of the models local roughness is needed to be able to prioritize certain areas over others. In [Roettger et al. 1998] S. Röttger et al. describes a method to calculate what they call the surface roughness. Having this measurement makes it easy to prioritize rough areas to smooth areas, but it is recommend to use it in conjunction with some measurement of distance to the viewer or otherwise view dependent method section 5.2.4.

5.2.4 View Dependent LOD

Extending continuous LOD with information about the viewer allows a more specialized LOD method called view dependent LOD. Having information about the viewer gives the option of choosing which LOD certain models of the scene need to be. Also the models orientation and topology can be used to decide its LOD. For instance the part of the model facing the viewer should be more detailed than the part not visible. The models silhouettes LOD are also important. In this way a model has no longer a uniform LOD as for instance discrete LOD. One could even take into consideration the viewers point of interest in the scene and let periphery parts of the scene have lower LOD.

Figure 4 shows an example of continuous LOD. The observer was originally on the left hand side and was looking to the right.

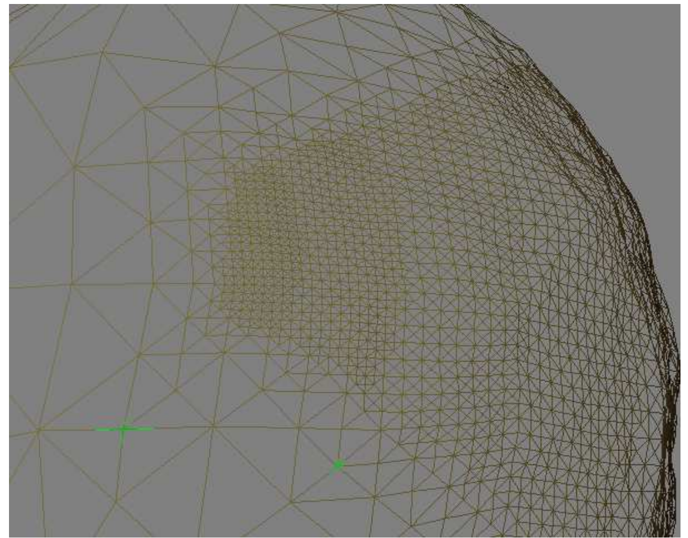


Figure 4: Showing View dependent LOD

5.3 Motivations for using LOD, visual quality vs interactivity

Choosing a LOD technique depends somewhat on the applications goal. If it is of great importance to render true to the original model in a visual optimal fashion or if a high frame rate in some interactive application is the main goal. Maybe there is a limited amount of resources at hand, as for instance polygons or no preprocessing is possible. In a interactive application one would usually try to get optimal visual result using some constraints which could a fixed amount of resources. For example having a fixed number of polygons for a scene, the goal is to use these polygons in a optimal fashion so that the models LODs result in a better visual result.

5.4 An example

Using a camera with a field of view vertically 45 degrees, horizontally 60 degrees and a resolution of 1024x768, it is possible to see the effect of a constant LOD.

Assuming the terrain is defined by a hight map with height values for each square meter then a mountain with a base of 20 by 20 square kilometers and a height of 10 kilometers would be represented by 400 millions height values or 800 millions triangles.

If one was to look at a similar pyramidal shaped landscape formation with base two by two square meters and height of 1 meters, then it would be represented by merely eight triangles. Illustrated in Figure 5

The two models has pronounced different details, but they will be projected to the screen with an area of same size. Table 1 shows that both fills 354x185 pixels on the screen. This means, the mountain has 3 000 visible triangles per pixel, while the pyramid has less than one triangle per pixel or about 30 000 pixels per triangle.

It is obviously a waste of resources visualizing the distant mountain with such a high Level of Detail. At the same time, it is clear, that the pyramid is represented at a too low level of Detail. There is no advantage representing the terrain by more than one triangle per pixel and its only an extra burden on the hardware.

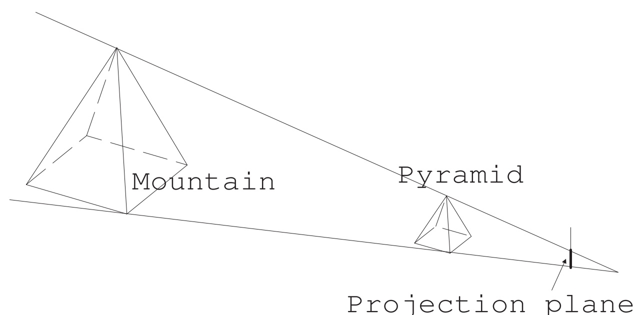


Figure 5: Showing the distant, large mountain and the nearer, smaller pyramid

$$hm = \frac{10km}{tan(45/2)100km} 768 \text{ pixels} = 185 \text{ pixels}$$
$$wm = \frac{20km}{tan(60/2)100km} 1024 \text{ pixels} = 354 \text{ pixels}$$
$$hp = \frac{1m}{tan(45/2)10m} 768 \text{ pixels} = 185 \text{ pixels}$$
$$wp = \frac{2m}{tan(60/2)10m} 1024 \text{ pixels} = 354 \text{ pixels}$$

Field of View : 60 degrees wide and 45 degrees heigh
Screen : 1024 by 768 pixels.
hm : height of mountain
wm : width of mountain
hp : height of pyramid
wp : width of pyramid

Table 1: A distant large object and a near small object projected to same screen size with constant Level of Detail

6 ROAM

We choose to use ROAM[Duchaineau et al. 1997] as our Level of Detail algorithm. There are many other algorithms which all serve the same purpose of generating or transforming a mesh with view dependent continuous level of detail section 5.2.4, but we selected ROAM because of its simplicity and because it is very well documented, easy to alter for various purposes and still performs quite well compared to newer algorithms. See [Bradley 2003] for comparisons between ROAM and other algorithms.

ROAM comes in a newer version 2.0 [Duchaineau 2006], where we use version 1.0, but 2.0 is still a work in progress and most of the major changes are related to moving triangles faster through the pipeline, at the cost of far greater complexity. ROAM 1.0 is considered to be fast enough for most purposes.

In section 6.4 there is an example of a planar mesh being subdivided to level 4. It may help the understanding of the following sections, to refer to that example.

6.1 ROAM from 10 000 feet

ROAM is an acronym which stands for Real-time Optimally Adapting Mesh. As the name states, the algorithm optimizes a mesh quickly enough to be real-time. By ”optimal” we mean to represent the more complex underlying surface with the least amount of error, given certain constraints on time, triangle usage or perhaps something else. The term ”error” could mean anything, but for our purpose we will refer to visual error. That in turn means that we are not dealing with the actual error, but rather the perceived error from some specific vantage point.

A main property of ROAM is the fact that if neither the terrain nor the observer changes, then the mesh also does not change. If the viewer or terrain only changes slightly, then generally so does the mesh. For an animated movement through a terrain this is an important property, since it takes advantage of the frame coherence which normally exists. Some have suggested that it does not make that big a difference, and have opted to not use it, as discussed in section 6.8, and to settle on a simpler version of the algorithm.

6.1.1 Basic ROAM geometry

The basic ROAM geometry consists of two right angled isosceles[3] triangles which are joined together along their common hypotenuse. From that rudimentary form a more complex mesh can be generated by subdivision. Such a base line aligned pair of triangles are called a diamond. The error of a triangle is the difference between it and the perfect subdivision. The error of a diamond is the maximum of its two triangles errors.

We will refer to the right angled corner as the top and the hypotenuse as the base edge. With the triangle oriented so that the base is down, the left and right edges are named left and right accordingly. The neighboring triangle along the left edge is the left neighbor and the triangle along the right edge is the right neighbor, and the neighbor along the base is the base neighbor, as seen on Figure 6

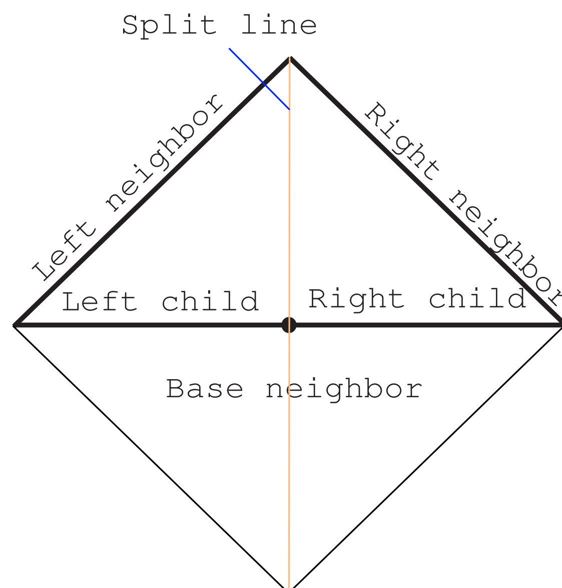


Figure 6: The standard ROAM triangle

6.1.2 Altering the mesh

If the mesh is not optimal, it will need to have triangles split or merged to better minimize the total error under the given constraints. On Figure 7 you see a mesh at six different levels of subdivision. The entire mesh is at a constant level of detail.

Splitting and merging given an upper and a lower error bound will be done using two prioritized queues of triangles and diamonds.

[3] Only the 2D-projections of the triangles are right angled isosceles. In 3D they are not - unless they happen to lie in the plane

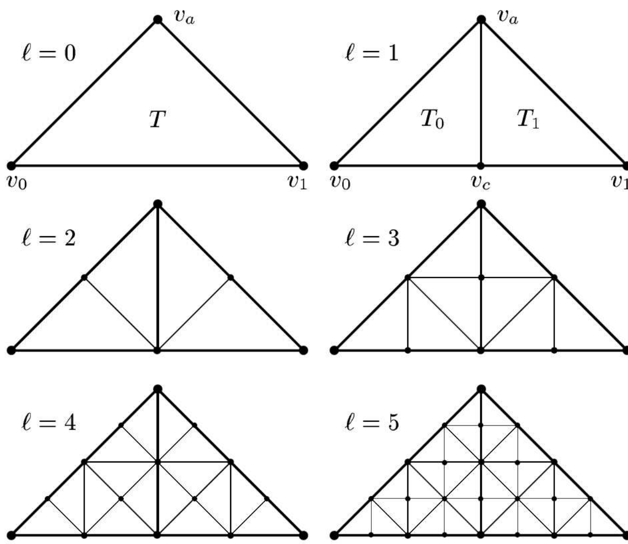


Figure 7: Different subdivision levels [Duchaineau et al. 1997]

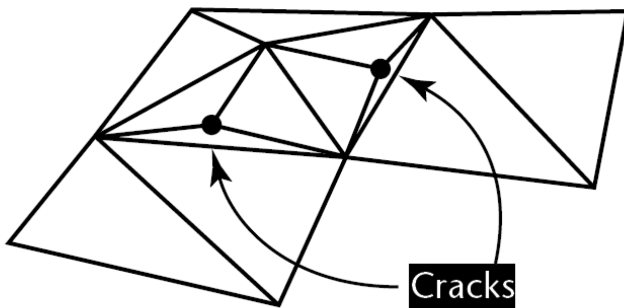


Figure 8: Cracks in a mesh which was not recursively split [Luebke 2003]

Splitting When a triangle is found to be too inaccurate, to represent the underlying detailed surface, within the given error bound, it will be split along the base edge by generating a new vertex at the center of the base edge, along the split line (Figure 6). This gives rise to a parent child relationship where the split triangle is the parent of the new two triangles. This is most commonly stored in a binary tree.

If triangles are naively split along the base, as described above, then you will end up with cracks in the mesh as seen in Figure 8. This means that in order to split a given triangle, you will need to split its base triangle as well. If the base triangle is not of the same subdivision level as the current triangle (if its base is not the current triangle), then it must in turn be split recursively. This recursion ends when two triangles have each other as base neighbors. An example of such a recursive split can be seen on Figure 9.

Splitting is done using a loop as seen in Listing 1. T is a triangle. Q is the priority queue of triangles ordered by the error. The queue is accessed through push and pop, which make ordered insertions and extractions.

When recursion ends A problem with recursion is when a mesh is optimized under some constraint on the maximal number of triangles. At some point a triangle is found to be too inaccurate, and should be split. At this point we could have 6 triangles left on the budget and we split the erroneous triangle. It just so happens that the split needs to split yet another triangle and then yet another and so on. At the end of the chain we arrive at a triangle which can be split and we do so, and now return back up the recursion, but before we reach the original triangle (the one most in need of being split), we run out of triangles. This means that we used all the triangles for splitting less important triangles. The guarantee of an optimal mesh under the given constraints is no longer a given.

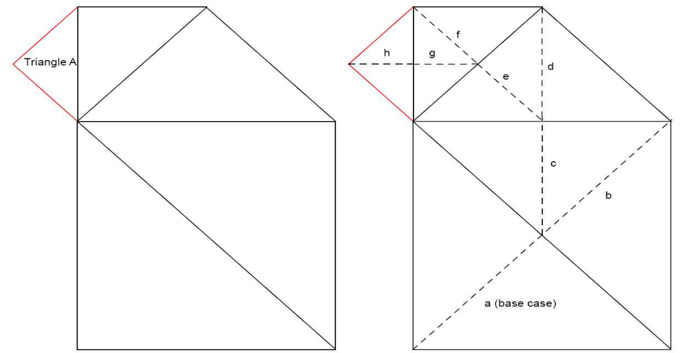


Figure 9: Recusive triangle splitting [Duchaineau et al. 1997]

Listing 1: Pseudo code for optimization by splitting

```
foreach T in Q                                      1
    T.Error ← CalculateError(T)                     2
T ← Q.Pop()                                         3
while (T.Error > UpperErrorBound)                   4
{                                                   5
    [T1,T2] ← Split(T)                              6
    T1.Error ← CalculateError(T1)                   7
    T2.Error ← CalculateError(T2)                   8
    Q.Push(T1)                                      9
    Q.Push(T2)                                      10
    T ← Q.Pop()                                     11
}                                                   12
```

Merging If a diamond section 6.1.1 is mergable, and is too detailed, it will be merged into the parent, and the mesh will be reduced by two triangles. It may sound strange that the mesh can be ”too detailed”, but the optimality is dependent of total error over all triangles *with* certain constraints, such as the number of triangles. Therefore the triangles might be put to better use elsewhere on the mesh.

A diamond is ”mergable” if the two triangles, which it consists of, are both split once, and if their children in turn are not split. On Figure 10 such a mergable diamond is shown on the right hand side, with a merged/unsplit triangle on the left hand side.

Merging is done using the following loop. D is a diamond. Q_D is the queue of mergeable diamonds ordered by the error of D, which is accessed through push and pop. Q_T is the queue of triangles. E(D) is the error of D and E(T) is the error of T.

1. For each D in Q_D calculate E(D)
2. D = Pop top from Q_D
3. While E(D) $<$ lowerErrorBound

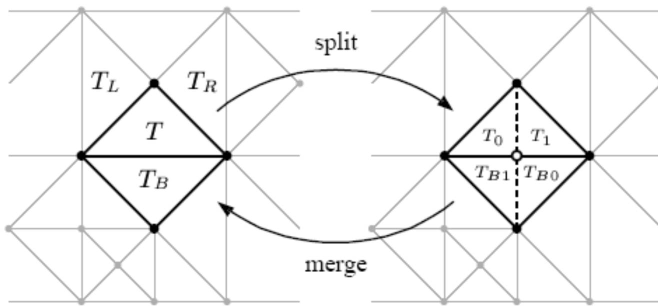


Figure 10: Splitting and merging with ROAM [Duchaineau et al. 1997]

4. Merge D into T_1 and T_2
5. Calculate E(T_1) and E(T_2)
6. Push T_1 and T_2 into Q_T
7. If T_1 or T_2 are now in mergeable diamonds, then push the diamonds into Q_D
8. D = Pop top from Q
9. End while

6.2 Prioritizing the triangles

For the splitting and merging to result in an optimal mesh, you need to always split the triangles with the largest error and merge the diamonds with the least error. For that to happen you will need to decide on how to represent the term "error" in the algorithm. The original ROAM description uses world space differences projected to screen space, but error could be anything. It could be the difference in color, the movement of a coast line etc. In section 6.9 we will go into detail with the ordinary geometry error metrics and in section 16 we deal with other priorities which we consider relevant for our application.

As stated previously we consider error to be visual error as seen from a specific vantage point.

6.3 Boundaries and ROAMing a sphere

An issue, which we have ignore till now, is the boundary conditions. What happens on the edge of the mesh? The edge is where a triangle shares one of its edges with no one. The original ROAM[Duchaineau et al. 1997] deals with this by always allowing splits and merges on the edge, as long as the diamond or triangle part of the split/merge allows it. We can however deal even more relaxed with it. We do not have an edge.

Our mesh might be perceived as a 2D map (which it is), but it is also the surface of a spherical shape. The mesh therefore wraps around and connects with itself, thus eliminating any edges. This does, however, mean that the base mesh can not be a single diamond. instead it must be a shape with volume. Any shape with volume will do. The smallest such shape is a tetrahedra which consists of four triangles. An easier shape to define would be a cube consisting of 12 triangles. The complexity of the base shape does not affect the complexity of the final mesh. A more complex base shape will simply be subdivided less in subsequent optimization steps.

6.4 A ROAMing example

The mesh (Figure 11) starts as two right angled isosceles triangles, which define a diamond. The diamond is splitable since neither one its triangles have been split already. For some reason we have a special interest in the area near the star, and want that part to be more detailed. Our priority, in the example, is to always generate finer detail around the star. We will not do any merges, but only splits, since the priority (the star) does not change between the different steps.

The optimization steps result in the construction of the binary tree in Figure 12).

1. The star is inside A, so it should be split. In order to split A we must split B as well. A is split into C and D while B is split into E and F.
2. Now the star is inside C which we will split. Any triangle on the edge can be split section 6.3 and C is now split into G and H
3. We now want to split H, which is *not* inside a splitable diamond. Therefore we must first split its base neighbor which is F. F can be split into I and J.
4. Now we can split H by also splitting its base neighbor I. H is split into K and L while I is split into M and N.
5. The star is inside L which must be split. L is not in a splitable diamond, so first its base neighbor G must be split. G is not in a splitable diamond either, so its base neighbor D must be split, which is can. D is split into O and P.
6. Now we can split G into Q and R while we split its base neighbor P into S and T.
7. The recursion has returned to splitting L which is now possible. L is split into U and V while its base neighbor Q is split into W and X
8. The optimization can continue till we have reached a maximal number of triangles, or till the mesh is sufficiently detailed around the star.

During the optimization, the binary tree on Figure 12 was built. Care must be taken to make a triangles left child its left node in the tree. This is important when reading out the triangles as one long winding strip and we will do in section 6.6.1.

6.4.1 Prioritized queues

The data structures, used to contain the prioritized queues of triangles and mergeable diamonds, need to be quite fast. It will be used extensively to extract from and insert into. The most common structures are doubly linked lists and binary heaps.

Doubly linked list This data structure is very easy to implement and understand. It is easy to only allocate the memory currently needed and allocate more as the list grows. For smaller datasets it will often be quite fast. Ordered insertion is O(n) and ordered extraction is O(1).

Binary heap A binary heap is a little more complex to implement and it usually has a larger overhead. You generally need to allocate all the memory, which will be needed in the future, from

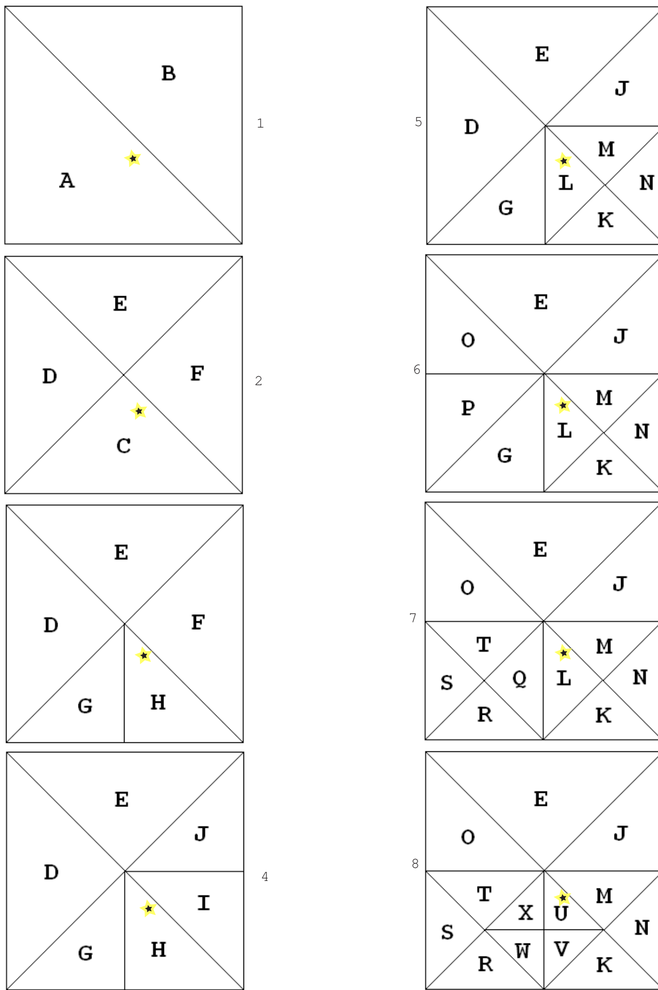


Figure 11: ROAM subdivision in 8 steps

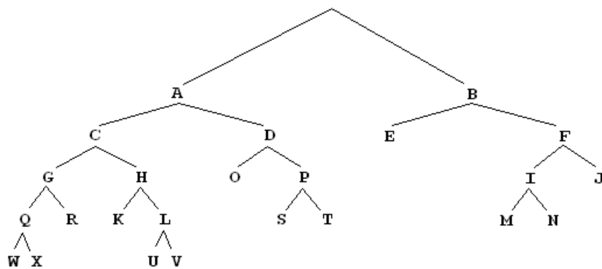


Figure 12: A binary triangle tree containing a ROAM

the start, since it will be quite time consuming to expand (reallocate) the structure later on. It is however a very quick structure for anything but tiny datasets. Ordered insertion is O(log n) and ordered extraction is O(log n) as well.

Comparison For smaller datasets and for simple tests, which has no great need for speed, the doubly linked list is the logic choice over the heap. However, for use in ROAM, which needs to be somewhat speedy, and especially for use in a ROAM of the size which we are intending to use it for, the heap is the better choice. Doubling the detail level of the mesh will only increase the workload by one[4] whereas the workload will have doubled for the list. This is a very dominant factor when we are talking about a mesh with several hundred thousand triangles.

6.5 Memory usage

It is generally preferable to preallocate memory if many objects will be created and destroyed runtime, and if this should be done as quickly as possible. For that reason, it is relevant to know how much memory should be set aside.

If you define an upper bound on the number of triangles, you can make the following assumptions about memory usage.

Vertices The relationship between the number of vertices and the number of triangles is linear.

Every subdivision will result in two triangles being split into four new triangles with a common top vertex. The common vertex is the newly generated vertex in the mesh.

If your base mesh is a cube consisting of 12 triangles and 8 vertices, then the subdivision will result in removing two triangles[5], creating four new child triangles and one new vertex. In other words, you add half a vertex for each new triangle.

The relationship between triangles and vertices is therefore always (1).

$$n_{vertices} \frac{n_{triangles}}{2} + 2 \quad (1)$$

A mesh of 64 000 triangles would need 32 002 vertices.

Triangles If triangles are stored in a binary tree, then even after a triangle is split, and is no longer visible, it still exists as a parent in the tree. This means that splitting a diamond results in creating four new triangles, hiding two and not deleting any.

If the mesh starts out as a cube with 12 triangles, of which none have parents, then it is not an ordinary binary tree with a single root, but rather a tree with 12 ”roots”. Every time the mesh is subdivided, you double the number of visible triangles.

For 16 000 visible triangles, the tree will need to hold 31,988 triangles as shown in (2).

$$n_{triangles} = 2 \cdot n_{visibleTriangles} - 12 \quad (2)$$

Diamonds Each splitting of a diamond results in a mergeable (Figure 10) diamond. At the same time zero, one or two previously mergeable diamonds should now be considered non mergeable (Figure 10).

How many diamonds are non mergeable depends on whether the triangles in the split diamond were part of mergeable diamonds before the split.

It is therefore not easily defined how many of the total number of diamonds, that are mergeable. What *can* be said is that the total number of mergeable diamonds is never greater than $\frac{1}{4}$ of the total

[4]It will generally be one iteration

[5]a triangle and its base must split together

number of diamonds since any triangle can at most be part of one diamond, and a mergeable diamond contains four triangles. Each subdivision of a diamond

Every subdivision of a diamond creates one new diamond, which is mergeable. This implies that the total number of diamonds, both mergeable and non mergeable ones will be given by (3). Given an upper limit to the number of triangles, we now have the upper limit of diamonds.

$$n_{diamond} = \frac{n_{visibleTriangles} - 12}{2} \quad (3)$$

6.6 Rendering a ROAM

To render the ROAM, you need to extract the visible triangles from the binary tree. By visible, we do not mean triangles which are within the view frustum, but rather triangles which have not yet been split into more detailed child triangles. The visible triangles are therefore the leafs of the tree as seen on Figure 12

There are generally two ways of doing this.

One method entails maintaining a list of visible triangles next to the binary tree, and letting the elements in this list point into the leafs of the list. Another method simply runs through the entire tree and outputs the leafs.

Though it is slightly faster to iterate through a simple list, it is only twice as fast as running through the tree. This is due to the property of a binary tree, whether it is balanced or not, which ensures that half the nodes in the tree will always be leafs while the other half is not. This means that for each leaf, the reader method will have to touch two nodes.

While being somewhat faster, the simple list suffers from a maintenance overhead. When triangles are altered, you have to update the tree *and* the list. This takes time and complicates the code. It also makes it more problematic to obtain the correct ordering of the triangles, as we describe in section 6.6.1.

We opt for the tree on its own, and do not want to implement the extra list.

6.6.1 Triangle ordering and format

An issue with ROAM, which has been considered its greatest weakness, is that it generally renders the triangles in an arbitrary order. This makes it impossible to use triangle strips, which would yield better performance on modern graphics cards. Triangle strips (Figure 13) are faster than a sequence of isolated or unordered triangles because it requires fewer vertex transformations. Each new triangle introduces one new vertex, while the other two vertices are recycled from the previous triangle. By recycling we mean to say that it is cached on the GPU, so that its screen transformed position can be reused as is.

To output a triangle list, you need the triangles to be output in a special order as seen on Figure 14. The last two vertices output will be the first two vertices of the next triangle. The winding order of the triangles also has to change between each triangle. The first triangle, of the strip, on Figure 13 has clockwise winding while the second has counter clockwise winding and so on. This is the way a graphics card expect the triangles, and if the format is not followed, it will result in errors.

The strength of triangle strips is the caching, but the cache is used for all indexed geometry and not just triangle strips. Most current graphics cards have a vertex cache of 21 vertices. This cache can be used when dealing with indexed geometry. With indexed geometry you provide a list of vertices and then you render separate triangles by indexing into the list. The triangles, shown on Figure 13 in the triangle strip, are made up of 5 unique vertices. This could be rendered as a list with the 5 vertices and then a sequence of indexed triangles such as [1,2,3],[2,4,3],[3,4,5]. This will result in the vertex cache being used. When vertex 2 has been transformed to screen space, it is stored in the cache. When the next triangle also indexes vertex 2, then it is fetched from cache, which is much faster than transforming it again.

Using indexed triangles can provide the same gain in speed, if the cache is utilized well. With a small cache of 21 transformed vertices, you will need to ensure that the vertices are somewhat ordered. If a vertex is being used too long after the first use, then it will have been evicted from the cache, and it will be transformed once more.

We have concocted a method which can generally read triangles from the ROAM in optimal order. The method can not make any guarantees as to winding order so it can not provide a triangle strip. It can however utilize the cache very well. The method ensures that each new triangle only brings one new vertex into the cache.

6.6.2 A winding way through the mesh

When moving through the tree, to output triangles for rendering, the left-child right-child order is reversed for each level in the tree. You will start the recursive read at the root by moving down the left branch, then the right and then the left and so on until you reach a leaf. When reaching a leaf you return up a level and now take the other branch down. This simple pattern is repeated till all leafs (the visible triangles) have been read out.

The quick reasoning for this patterns is that a triangle is connected to neighbors at the same level through its left or right side to the neighbors left or right side. When staying at the same level, we go through the sides. When a neighbor is at a lower level (it is smaller) then we connect to it through our left or right side into its bottom side. When a neighbor is at a higher level we connect through the bottom to its left or right side.

Looking at the tree in Figure 12 the read pattern will result in the following sequence: O,T,S,R,W,X,U,V,K,N,M,J,E. At the root we move to the left and reach A. From A we move right to D. From D we move left to O. O is a leaf and is output. We return up the tree to D where we *this time* move to the right, since we have already gone left at D, and reach P. From P we move right to T, since P is a right-first level. Remember that its parent D was a left-first level. T is a leaf and is output. From T we return up to P and now go left to S, which is a leaf and is output. Now we return all the way up to A where we this time move left. This goes on until the entire sequence has been read out. Figure 14 shows the ordering on the mesh, where it can clearly be seen that each triangle shares two vertices with its neighbor.

We will make quantitative tests of this method to show the cache utilization with the normal triangle ordering of always-left-first versus our alternative ordering.

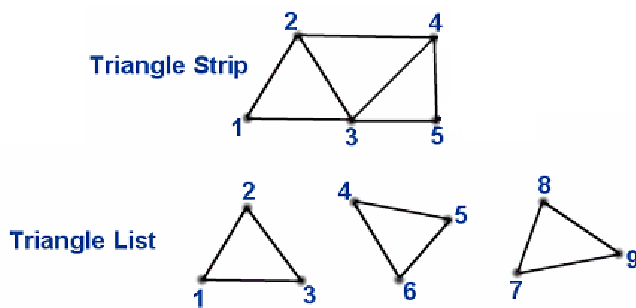


Figure 13: Triangles ordered as strip or list

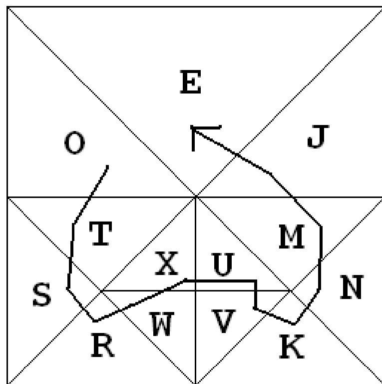


Figure 14: A winding, but ordered, way through the ROAM

6.7 Decoupling rendering and optimization

All previous implementations of ROAM, which are known to us, synchronize the update frequency with that of the rendering. The mesh is optimized, then rendered, then optimized again over and over. The optimization of the mesh generally takes much longer than actually rendering it. The optimization then becomes the limiting factor when trying to obtain a high frame rate.

We may need a frame rate of at least 30 FPS to give a smooth rendering experience, but does the mesh really have to be optimized at the same speed? If we fly close above the landscape with a speed of 100 m/s, then we have moved 3.33 meters in between frames. How much does the landscape have to be optimized to take that movement into account? The answer is ”it depends”.

There will be times where a movement of one meter will change the scenery from a cliff wall to a large view over the landscape. In that situation, it does matter quite a lot. Mostly, however, the scene will be the same and we need not update the mesh between each and every frame.

Looking forward What we need is a terrain which is optimized for more than one vantage point. It should be optimal as seen from every discrete step (frame) along the camera path. While it may be theoretically possible to render such a terrain, it will be somewhat slow, and we do not believe it is a path worth exploring further right now. What we suggest is to find an average of the cameras positions, and orientations, and optimize for that.

By average position, we mean to calculate where the camera will be, and how it will be oriented at $t+\frac{1}{2}\Delta t$ where Δt is the time in between updates. This can be done very easily by making a one step Euler integration over time as in (4), where $camera'(t)$ is the current linear and angular velocity of the camera. This method takes only the current camera properties into account, and assumes that the velocities of the position and the orientation, are constant throughout the time step. While other estimates could take multiple previous positions and orientations into account, and deal not only with the cameras first derivatives (velocity), but also with its second derivatives (acceleration) it will seldom be worth the trouble. It should not make much, if any, visible difference which method is used, since the camera will generally not move too abruptly, so the linear and rotational acceleration will be very low.

$$camera(t+\frac{1}{2}\Delta t) = camera(t) + \frac{1}{2}camera'(t)\Delta t \tag{4}$$

Relaxing it a bit It should be a good approximation to optimize for one half time step ahead, but we need to be a bit more relaxed on the error calculations when optimizing. If triangles are culled exactly when they are out of view, and thus are not subdivided into finer geometry, then that will be just fine for that specific location and orientation, but if we move ever so, the culled triangle will come into view and the scene will look rubbish. Since the scene is being used for more than a single frame, odds are that the triangle, which is culled from the strict optimization at $t+\frac{1}{2}\Delta t$, will at some point, in the interval $[t..t+\Delta t]$ be visible. There are two easy ways to relax. One is to widen the field of view somewhat. How much exactly, is to be find through trial and error. It will depend on how the camera is being moved and on the terrain. The other method is to move the camera a little up from the ground. With a terrain without overhangs, you will always see more of the terrain the higher you are, so more of the terrain will be considered important and get prioritized in the update.

There is always a downside The mentioned method has many valuable properties, but it does come at a cost. When one update must optimize for more than one (different) frame, no single frame will be optimal. This means that for any given frame, the mesh will be likely to have fine detail which is out of view, and the geometry which is in view will have less detail, because valuable triangles are wasted outside the field of view, and because the current frame can see geometry which was culled entirely in the central frame used for optimization.

Multi core systems Systems with more than one processor/core will benefit greatly from this separation of mesh optimization and rendering, since now one processor can render at full speed while the other optimizes at full speed. While the mesh optimization itself is hard to parallelize[6], the separated optimization and rendering is born for dual processing. Should it be, it can easily be reworked to run on even more processors. Three obvious tasks exist: optimization, copying the optimized mesh to the graphics card and rendering the mesh.

The conclusion This method has its down sides, but they are few. The visual degradation, that we should expect, can be entirely outweighed by the larger number of triangles that we are able to use now that we do not have to restrain the mesh optimization in order to get an acceptable frame rate.

[6] We know of no implementation which parallelizes ROAM

It is worth noting that the above mentioned method requires the mesh to be duplicated. One copy needs to be located on the graphics card while being rendered, and another is located in system memory while being optimized. One additional copy is needed since the optimized mesh will be copied from system memory to graphics memory while the mesh is being rendered. This means that two meshes should exist on the graphics card, so that one can be rendered while the other is being overwritten with the newly optimized version. When the rewrite/copy is complete, the renderer can switch to the newly optimized mesh and render that while the optimization and copy to graphics card is running on the other copy.

6.8 Variations of ROAM

Split only It has been suggested that the overhead of maintaining two separate priority queues, and transforming the mesh from frame to frame, is somewhat wasted effort[Polack 2003]. Some argue that the frame to frame coherence is in fact not large enough to warrant the more complex code, and perhaps it will provide better performance, and simpler code, to start from scratch with the base mesh every frame and then subdivide till the error bounds have been met. The basic argument is that the majority of geometry is generally near the observer, and if the observer moves, then the part of the scene which is close by, will often move out of view thereby *greatly* changing the priority of most of the geometry. It is clearly a much simpler approach, but whether it is a good idea depends largely on the way the terrain and observer behave.

Geo morphing When a triangle is suddenly split into two and the midpoint is offset, then this might cause what is known as "popping". From one frame to the next, the geometry pops up or down. Even pops which seem small in screen space can be very visible because of the way humans observe the world. Fast movement, ever so small, catches our eye. To reduce the popping problem, you can implement geo morphing[Snook 2003] which stretches out the pop over several frames. You subdivide as before, but at first the new vertex, at the midpoint, is positioned on the old baseline. Then over the next few frames it is moved up to its actual position. If the observer is not stationary, then such a slow movement of the terrain will not be noticeable since the entire scene will already be moving through the field of view. We will implement geo morphing and test it relative to no morphing and make conclusions about subjective performance in our scene. Geo morphing requires that one does not rebuild the entire bin tree of triangles every frame, since this will prevent morphing. There are no longer two states to blend between.

6.9 Ordinary ROAM error metric

The original ROAM [Duchaineau et al. 1997] uses the world space difference between the current triangle and *all* of its lower subdivision level triangles. That difference is then projected into screen space, for some camera location and orientation, where it is measured as pixel difference. Figure 15 shows such a projected error for a 1D example.

Obviously, the larger the area of the projected difference, the larger the error look on screen. However, defining the difference and then projecting it to screen is in no way a cheap operation, and it is something which has to be done every time the mesh is optimized.

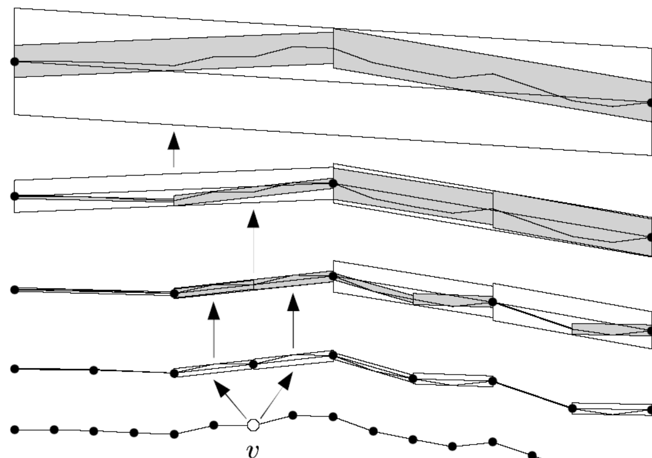


Figure 15: Projected ROAM error [Duchaineau et al. 1997]

6.9.1 Alternative ROAM error metrics

The original ROAM metric is "correct" but expensive. Perhaps the error can be calculated "well enough" in a much cheaper fashion. After all, this is all about making a visual impression which works, and not about making anything with greater accuracy than what will actually be noticeably.

Projected midpoint vector Instead of projecting the entire difference area, perhaps it will sufficient to project a vector from the midpoint of the current triangles base edge to the new vertex (Figure 6) on the two triangles of a lower subdivision level. Then the length of the vector, in screen space, would accurately represent the movement of the midpoint, though not the area of the movement.

Simple distance The general rule is that the perceived width and height of an object is halved whenever the distance to it is doubled. That in turn means that the area is one quarter that of before. The same thing can be said about the size of the error. The projected error vector, as mentioned before, will decrease in length as it moves farther away, and the projected error area will decrease even faster. Not all triangles have the same world space error, but when we consider the perceived error, it is a result of distance and actual error. The distance term will generally dominate the actual error, so one might argue that the distance from observer to triangle could be used as a rough, but very fast, error metric. Either as $\frac{1}{distance}$ or $\frac{1}{distance^2}$.

Distance and view vector Errors result in movement away or towards the center of the planet. That means that when the view vector is at a right angle to the vector from the planet center to triangle (the radius vector), then the error will be far more visible then when the view vector is parallel to the radius vector. This observation could be combined with the simple distance metric to form a perhaps superior metric, which would be defined as $\frac{1-dot(radiusVector,viewVector)}{distance}$.

It is impossible to know which of these metrics will perform best. It depends on various factors and can really only be determined through testing.

6.9.2 The uncertainty of lower subdivision levels

A problem, not mentioned before, is that while ROAM is originally designed to deal with preexisting geometry, we will generate the geometry on the fly. ROAM is designed to work in a bottom up fashion with full knowledge of the details, which are then simplified. By generating details, as we do, we are working top down and therefore do not know the details before we actually calculate them, and we can not go all the way down, due to computational cost. The whole idea with our CLOD scheme is to not calculate geometry which is not shown, and now we need to calculate the finer details for triangles which we might not even want to split into a finer mesh. This is a problem.

Obviously we can not deal with the difference between the current subdivision level and all the lower levels. As argued in section 3 all the detail of the world, even at a fairly rough scale, will be unmanageable.

The solution could be to only deal with the first lower subdivision. A triangle of level 5 would contain the calculated position of the level 6 vertex. That way the triangle can quickly enough decide on its error. It is not truly accurate since the offset to level 7,8,9 are considered to be zero, which they might not be.

As described in section 9.6.1 the possible offset between each subdivision level decreases the deeper you go. An often used factor of decrease is two, which means that the maximal possible error introduced by a finer subdivision level will always be half of the maximal possible error on the current level. Knowing this factor, you can establish an upper bound on the error, and use that for prioritizing. Again it is not correct per say, but it does allow us to choose the best candidate for subdivision given the limited information that we have.

Upper error bound If we for a triangle know the position of the base edges midpoint on the subdivision one level finer than the current, and we know how the maximal possible error is scaled, then the upper bound will be given by (5).

$$errorBound = \sum_{i=0}^{i=\infty} \frac{maxOffset}{scaleFactor^i} \tag{5}$$

Which reduces to (6) for a scaling factor of 2.

$$errorBound = 2maxOffset \tag{6}$$

We can therefore with a certain degree of accuracy consider the world error of a triangle as the distance between its current midpoint on the base edge and the next finer subdivision levels vertex plus the upper error bound times two. It is given that the error is not larger. It can be smaller, but not larger.

Another perspective on the error/possible offset of children is given in section 7.2. The method mentioned there can calculate the upper error bound for a given scaling without the need to actually calculate any deeper subdivision levels.

7 Bounding volumes

There are many situations in which one needs to test if a given triangle is colliding with some other geometry. When testing for visibility, we need to determine if the triangle is entirely outside the view frustum, or if it is inside. If we need to place structures on the ground surface, we need to refine the terrain at that specific location, and if visibility testing to a specific area is done with line of site, we need to see if a line intersects with a triangle.

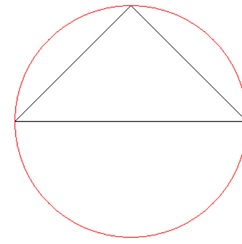


Figure 16: A minimal bounding sphere for a right angled isosceles triangle

A bounding volume can be used when performing test for clipping and intersections. One generally tests if the volume is contained wholly inside or outside another volume. This other volume could be a view frustum, and it could be something else like another 3D object with which the first object should generate a CSG[7] model.

7.1 Optimal bounding volume

An optimal bounding volume has a very tight fit around the bounded object, and allows for very simple clip testing. Generally speaking, the more tight fit, the less simple the clip testing is. An exception to this is a sphere, where a perfect fit is obtainable from a bounding sphere, and the calculations are very easy. When dealing with a triangle, one can take advantage of the fact that a triangle is always contained in a plane, which simplifies things, but still all three corners need to be tested for clipping. Another volume could be a small box which contains the triangle. If it is axis aligned, then it can fit quite tight. Boxes still require a few too many computations when dealing with volume-volume intersections. For that purpose the simplest and fastest method is to use bounding spheres, where one can always say that if a spheres center is more than its radius outside another volume, then the contained object is certainly outside.

For a right angled isosceles triangle a valid bounding sphere would be centered on the midpoint of the hypotenuses, and have a radius of half the hypotenuses. The three corners of the triangle would all touch the sphere, which can not be any smaller. It is evident that the bounding sphere is not a tight fit, but it is simple, and besides that, the bounding volume needs to contain not only the triangle, but also all its descendents.

For one triangle this is a suitable bounding sphere, but if the triangle is to be split, then its children might expand outside the original bounding sphere, which is a problem. A bounding sphere must be guaranteed to contain the entire primitive and, when dealing with primitives which can be refined, to contain all child primitives. As can clearly be seen, the bounding sphere will be broken if the triangle is subdivided in such a way that its new geometry is offset by more than the spheres radius. Even when the offset is not that large, subsequent subdivisions can still move the child primitives outside the original sphere as seen on Figure 17

[7] Constructive Solid Geometry

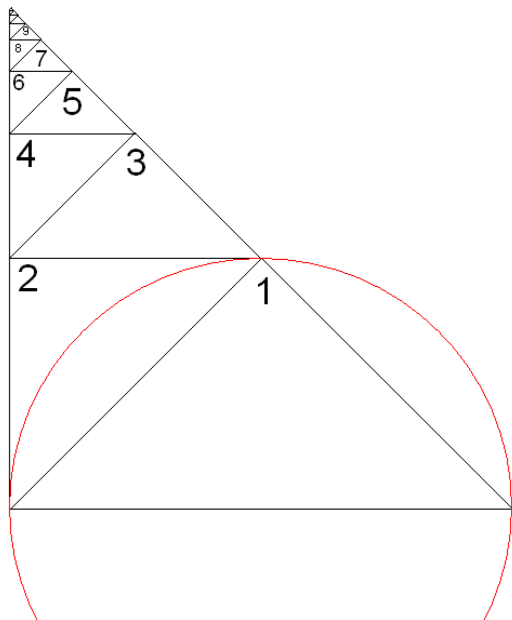


Figure 17: The bounds being broken by subdivision

7.2 Breaking the bounds, and expanding them

The result of children breaking the boundaries (as children often do ;-)), is that tests performed on triangles based on the bounding spheres, will fail. This can for example make triangles, which are in plain view, appear as if they are outside the view frustum. We need to expand the bounding sphere based on how much we allow the child objects to be displaced. The more each subdivision can be offset, the more the bounding sphere must be expanded. As can visually be seen from Figure 17, the vertical distance from the base line and to the top of the finest triangle is approximately the double radius. The illustration shows subdivision where the offset each step is half the length of the original line. In (7) the relationship between vertical and initial radius is shown, where L is the initial line length and δ is the scaling in each step. A δ of $\frac{1}{2}$ would imply that each offset is one half the length of the divided line segment. The equation follows naturally from the observation that if a line of length L is displaced $\frac{1}{2}$L to form two new line segments of length $\frac{1}{2}$L, which are again displaced their half length, then the worst case scenario, where every displacement is in the exact same direction as the previous, will move half the length of the previous move. This is the summation in (7), and that defines the upper bounds (the convergence) for the expansion. On Figure 17 the subdivisions are not all moving the same direction, but every second subdivision is. This means that twice as many steps are needed to move a certain distance away from the initial triangle, but for an endless summation it amounts to the same upper limit.

A sum in the form of (7) can be transformed from an endless summation into (8), which is undefined for $\delta \geq 1$, since it does not converge. We can obviously not have an upper limit on the offset if we keep expanding the line segments, rather than making them smaller. This shows that for $\delta = \frac{1}{2}$ we get an offset r of L, where a $\delta = \frac{1}{3}$ would give us $\frac{1}{2}L$. The higher the division factor each step, the less the radius expands.

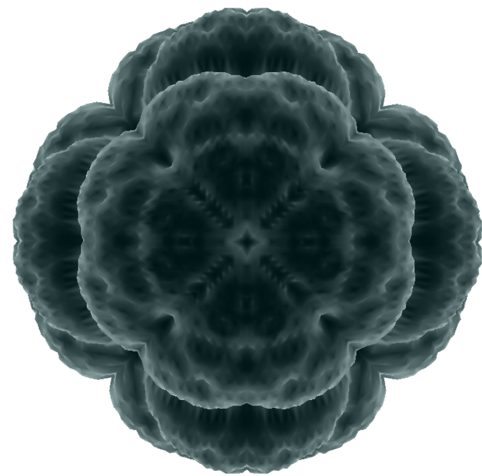


Figure 18: The bounds being broken by subdivision

$$r = \sum_{i=1}^{\infty} L\delta^i \tag{7}$$

$$r = \frac{L}{\frac{1}{\delta} - 1} \tag{8}$$

For a scaling factor of $\frac{1}{3}$ we had an offset r of $\frac{1}{2}$L which is the radius that we used originally. This implies that for scaling factors of $\frac{1}{3}$ or less, the naive bounding sphere will work. For scaling factors above $\frac{1}{3}$ it will not.

The resulting value r, is the radius of a bounding sphere centered on the initial baseline, if it is at least as long as $\frac{1}{2}$L. If r is smaller, then the bounding spheres radius is $\frac{1}{2}$L, so that bounding sphere radius is $max(\frac{1}{2}L, r)$

An example of a cube, consisting of 12 triangles, which are subdivided a number of times with a scaling of $\frac{1}{5}$, is shown in Figure 18. It is apparent that the cube expands most in the direction normal to the surfaces of the initial form, and then in the next subdivision it expands again most in the normal direction, but not as much as in the first step. This has the effect of making round expansions which again have round expansions on the, which again has... The initial cube is contained within the current object, and can not be seen any more. The cube is in a sense a slightly altered 3D-version of the Koch snowflake.

$$r = \sqrt{2(r_0 + \frac{r_0}{\delta - 1})^2} \tag{9}$$

8 Z-buffer accuracy

The meshes that we render are huge. An Earth-sized planet presents a terrain spanning 12 742 kilometers from end to end. From nearest point to farthest point there is more that 6 000 kilometers. It is not so much a problem of many triangles, since we will have optimized the mesh to make the best of some limited number of triangles, as a problem of scale. The depth buffer on the graphics card will

need to have high precision to allow for such a large scene, and still be able to correctly order pixels from the ocean floor and the surface of the shallow ocean near the shore. If the buffer is too inaccurate, then sometimes the ocean floor will be rendered on top of the ocean surface. If the camera is moving, then this will give a flickering shore line where sometimes water is rendered on top, and sometimes it is rendered as if below the ocean floor.

We find a 16 bit depth buffer to have too low accuracy, and one of at least 24 bit should be used. The Direct 3D API supports buffers of up to 32 bit, but as of 2006 most, if not all, graphics cards does not support this.

Assuming that we have a near and a far clipping plane separated by 7 000 000 meters, the space in between is large enough to hold one half Earth-sized planet[8]. The best accuracy of a 16 bit z-buffer, as calculated by (10), will then be 106.81 meters, which is certainly a problem for shallow water, and can be a problem in the landscape when landscape features are close together. For a 24 bit buffer the accuracy will be 0.42 meter, which is far better, and will generally be good enough (as we are looking down from orbit), if the near and the far clipping planes are placed as close together as possible. The problem can get worse if the planets are really huge, or if we need to render a planet and a moon at the same time. With the huge planet, the 24 bit will still often be enough. If not, then a ROAM, which is spawned from 12 parent triangles, can be split into separate objects for rendering, which we sort manually and render back to front. For the situation with a moon and a planet, we also sort by distance and render back to front and each time fit the clip planes nicely around the currently rendered object.

$$accuracy = \frac{far - near}{2^{bitDepth} - 1} \qquad (10)$$

8.1 Squeezing the planes

As (10) shows, the closer the clipping planes are, the better the accuracy. A person without much experience in 3D programming will often be tempted to place the near clipping plane just in front of the camera, and the far way out in the distance. This is easy way to ensure that the observed geometry will actually be within the clipping planes and be visible, and might even work, but it is in general a bad idea since the z-buffer accuracy drops drastically. What we need to do is to move the near clipping plane out as far as possible and to move the far in close. If optimally positioned, the geometry will fit exactly in between and we get full use of the z-buffers range from 0 to 1.

Figure 19 shows the optimal placement of the near and the far clipping planes for arbitrary geometry. The far clipping plane is moved in so close that it touches the farthest point on the geometry, while the near clipping plane is pushed out so far that it touches the nearest point of the geometry. Moving the planes any closer together will clip the model, which is generally not desirable.

While that clip plane placement if optimal for the geometry, viewed in that direction, it is not optimal if the geometry is not entirely inside the view field as is the case in Figure 20. In that situation we no longer consider the nearest or farthest points, but rather the nearest and farthest point which are visible. That means that we can, in that situation, move the near clipping plane out to the nearest point inside the view field. That point happens to be where the right side of the view frustum (the right clipping plane) intersects with the geometry.

[8] No more than half a sphere be seen from any point at any one time

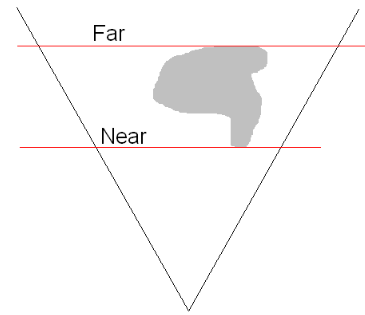


Figure 19: Naive placement of near and far clipping planes for arbitrary geometry

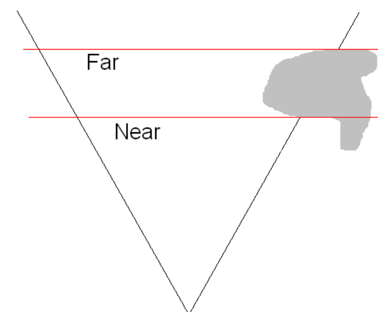


Figure 20: Almost optimal placement of near and far clipping planes for arbitrary geometry, taking into account that not all the geometry is inside the view field

The situation in Figure 20 is still not optimal since much of the geometry is hidden behind itself, as seen from that specific view point. This is known as ”self shadowing” which is a term mainly used when dealing with lighting. There is no need for the far clipping plane to be out as far as it is, since it is taking geometry into account which is actually not visible, even though it is inside the view field. Dealing with arbitrary geometry it is not a simple task to figure out where the self shadowing starts and ends.

We are however not dealing with arbitrary geometry, but rather with a spherical planet with tiny perturbations. For a sphere, it is not that difficult to calculate the point where self shadowing starts and ends. That point, or rather that circle, is the horizon. The horizon is where the line from the observer to the surface of the sphere, is coincides with the tangent at that point.

Figure 21 shows the optimal clip plane placement for a sphere. It takes into account that some of the sphere is behind the horizon, and it takes into account that some of the sphere is outside the view field.

How good is good enough? An average height person standing at the beach with the feet at the water-line will generally see the horizon 5 kilometers out. When rendering from that perspective, a 24-bit z-buffers accuracy of 42 cm (for a 7 million meters span) is

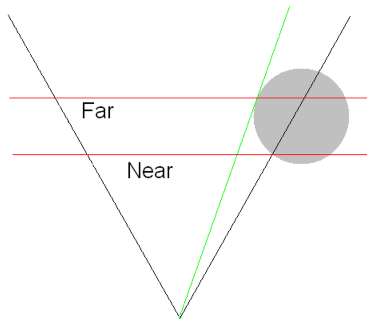


Figure 21: Optimal placement of near and far clipping planes for a sphere which is partially outside the view field. The green line is the line of site for the horizon point

not enough to avoid rendering errors, since the water surface might be less than 42 cm above the ground level, but if the clipping planes are placed optimally, the span between them need not be more than 5000 meters (actually 5 000 m minus 1.7 m, which is the persons eye height). That will then give an accuracy of 0.29 mm, which certainly *is* enough.

8.2 Optimal clipping plane placement

How do we ensure that the clipping planes are only far enough apart to contain the visible geometry, and are always that far apart?

Figure 22 shows a view frustum with a field of view at 90 degrees to easier illustrate the point. The clipping planes are based on the optimal clipping points A' and B', which are A and B projected onto the view vector.

A is given by the intersection of the view frustum and the sphere defining the planet. The illustration is in 2D, but, as Figure 23 shows, the point of intersection in 2D and 3D is the same. The point A is the intersection that we get from intersecting a 2D line and a circle, while Q (or rather the line between the Q points) is the intersection in 3D between a plane and a sphere.

B is the horizon point, which is defined as the point at which the vector from the eye point becomes the tangent of the circle. That is the point at which the horizon starts and from which the terrain farther out is not visible. On Figure 22 only the green segment of the circle is visible, even though the red segment is also inside the view frustum.

3D as 2D The 2D version of the clipping can be thought of as the projection of the 3D world onto a horizontal and a vertical plane through the view frustum. Define the horizontal plane as going through the eye point and having a normal vector pointing "up" in the view - the cameras up vector. Define the vertical plane as going through the eye point and having a normal vector pointing "right".

Now project the sphere onto the two planes and it becomes a two 2D representations. In Figure 22 the field of view was defined to be 90 degrees. In the two 2D spaces, the field of view is respectively the field of view and the field of view times the aspect ratio - if the view field is not square.

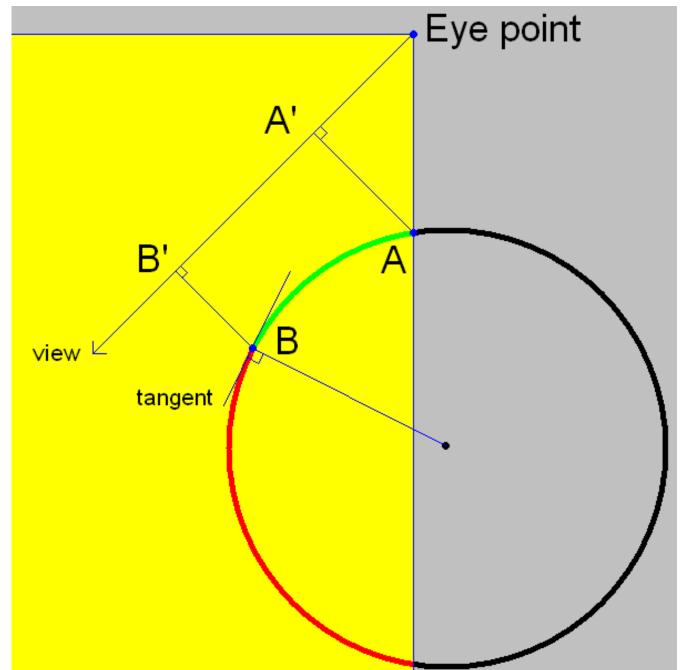


Figure 22: The optimal near and far clipping points for a view frustum observing a sphere. The near clipping point A' is A projected onto the view vector and A is the intersection of the view frustum and the sphere. The far clipping point B' is B projected onto the view vector and B is the point where the vector from the eye is the spheres tangent

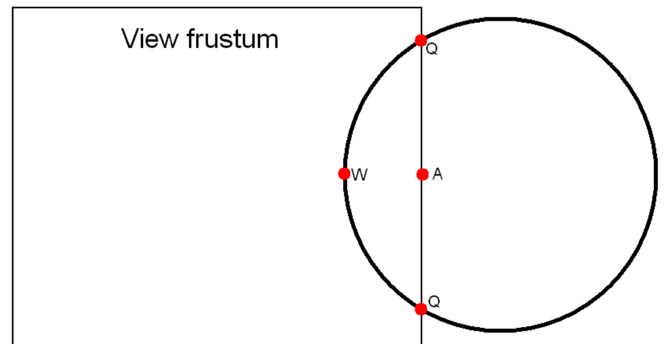


Figure 23: The intersection points of a view frustum and a sphere

8.3 Calculating the clipping points

The calculations will be explained in 2D and use the symbols from Figure 22. We need this in 3D, but as explained previously, 2D can show what is going on more clearly. The only extra complication is that the calculated A and B points, and their A' and B', might not be the same for both 2D projections (projections were explained in previous section). If one A' is closer than the other, then we need to use the closest one, and if one B' is farther away than the other, then we need to use the farthest one. This is because the clipping planes are just that... planes. They can not be bent to take into account that the planet might be closer in one projection than in another.

The same procedure could be used for arbitrary geometry, but would require every vertex to be projected, while a sphere is very

simple by only requiring one projection of the center point.

9 Procedural generation of geometry and terrain

Procedural generated geometry is a way to describe some instance without describing it explicitly. It could be some mathematical function, when used, which would return a description of, say, a sphere which in turn could be used to build a polygonal model of the sphere.

The polygonal model is produced when needed run time and not modeled before and stored. It is therefore only the procedural method which takes up space and not a model. This comes in handy when a model would be of considerably size. On the other hand, generating the procedural model when needed, takes time and burdens the hardware.

An example of procedural generation is seen in the game Spore from Firaxis [Arts 2006]. In this game procedural generation is used for many different purposes. They are generating the planets and their terrain procedurally. For different reasons. They have ambitions to have many different planets, and storing geometric representation alone would take up considerably amount of storage. They also want the game world to be new and exciting, so they rely on procedural methods to accomplish that. They use procedural methods for building small creatures, and by analyzing the model of the creature, they are able to animate them by calculating realistic movement.

9.1 Motivating use of procedural methods

When using procedural methods you will be given various parameters which can be used to influence the outcome of the method. It is not necessary for the user of the method to know how these parameters are used in the method, but rather the effect on the terrain. Parameters could for example be roughness of mountains, cursiveness of the shoreline etc. These parameters is also the only way to influence the terrain generated. This might in some cases not satisfy the users needs and in section 11 we discus this conflict.

During the last decades rendering hardwares capabilities has rapidly improved. In computer games we see a transition from simple restricted indoor scenes to complex and large outdoors scenes. It is possible to render these complex scene with modern hardware but it is a tremendous work to design all these scenes. Trying to shift the workload from designers to computers is one of the goals of procedural generation.

When using a procedural method for creating terrain for a outdoor scene, a designer need not to specify the terrain in every detail. Instead the designer uses the procedural methods parameters to define how the terrain should look like on a higher level of abstraction. So the designer could define how much of the terrain should be some mountains or to what extend the terrain should consist of fields, and the procedural method would generate the geometry, textures and what else would be needed to satisfy the designers wishes. But also in a random fashion, so that the terrain is varied even though the designer range of choices is limited.

In theory it is possible to achieve infinite fine resolution for the given geometry produced by the procedural method. A designer would have a hard time competing with such a method.

Procedural generating geometry might be useful in conjunction with LOD also. Using a procedural method to generate geometry when needed, based on the LOD-scheme, and not generating any more geometry than needed. This has the advantage of minimizing the size of the data for geometry.

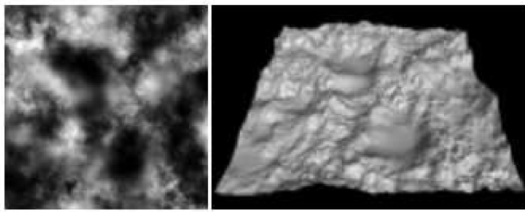


Figure 24: Height map and resulting mesh

9.2 An example of procedural generation

Procedural methods are implemented in different ways depending on what to generate and also depending on performance considerations. Performance comes into question when procedural methods are used in real time applications and then it has to perform fast as described later section 9.6. For real time applications it is possible to generate data in advance. If the procedural method is slow this could be the solution. On the other hand, datasets could be too large to be stored or to be processed and then a fast procedural method which generate data needed on the fly.

An example of how to generate terrain procedurally is described in the following. One way to represent terrain is using a height map. This is typically stored as a 2D gray scale image, where the intensity of each pixel represents the height of the terrain at the given pixels x and y coordinate. The height map can be transformed into a 3D mesh representing the geometry of the terrain, as seen in Figure 24, by interpreting the x,y pixel spacing in the height map as position in the horizontal plane, while the value stored in a pixel can be seen as the landscapes height above the horizontal plane.

The procedural method should then generate this height map. A simple way to do this is simply to find a random value for each pixel in the height map. This approach wouldn't produce a terrain that looks natural. Instead the method needs to be extended. When calculating a pixels value, its neighboring pixels values could be taken into account. So, if the neighbors pixel values are at a certain level, the given pixels value should have a value near this level, with the exception of an occasional steep cliff side. More things could be taken into consideration and the procedural method will generate more realistic landscapes. In section 9.6 through section 9.6.7 more details on the subject will be given.

9.3 Advantages and disadvantages of procedural generation

Summarizing the advantages from using procedural methods would be: It is possible to achieve high resolution for the procedural geometry at lower database size. Predesigned or samples from real world often has a fixed resolution and growing data size as resolution increases.

Procedural generation shifts the workload from designers to computers and thereby the production time in some cases. But still has the potential of being controlled by a designer at some higher level.

Procedural geometry can be more varied and therefor more interesting environment to explore.

Procedural generation has its limitations and disadvantages, which needs to be considered as well. Procedurally generating geometry run time will burden the hardware. Procedural methods can be CPU intensive and demands use of memory as well. Pregenerated geometry could be optimized for hardware in ways not possible for procedural generated geometry.

Procedural generated geometry in most cases needs some sort of LOD-algorithm which complexes the system more. It needs a LOD-algorithm because the geometry generated, in theory, is represented at infinite high resolution and therefore would occupy infinite data space unless some LOD-algorithm or bounds on the geometrys resolution.

9.4 Future perspective

How procedural generation will be used and to which extent in the future depends on many things. One aspect is the possibility for a designer to interact with the procedural method. The designer would be interested in being allowed to manipulate the result to a extend where the designer is deciding on every little detail and spanning to no control at all. Procedural methods should blend easily with predesigned geometry like a static mesh of a building in a procedural terrain or even predesigned area of the terrain like a city placed at a given point in the terrain. Procedural methods needs to be optimized for hardware and maybe the future will show uses of GPU-programming and multicore programming [Ebert 2003].

9.5 Use of procedural generation in this paper

We are using a procedural method for generating landscape for a planet. This landscape contains different elements which will be described in section 12. We primarily want to procedurally generate landscape for testing how it cooperates with the chosen LOD-algorithm. This could mean how we need to limit the procedural methods load on hardware, so it runs at a acceptable frame rate. We also use the procedural method to generate a natural looking landscape. This could mean a landscape which has elements seen on the Earth or the Moon for example. Landscape features are described in more detail in section 12, but in brief, we want to generate height values representing a varied terrain containing features such as mountainous areas, flatland and coastal lines between mainland and the ocean. Generally speaking, we need a procedural method, which for some 2D coordinate returns a hight value. Much like described in the above example section 9.2, but not storing the height values in a height map, but generating them when needed.

Instead of generating textures for the terrain we use the GPUs pixel shader and involve a procedural method for calculating colors. This could involve considerations like the terrains altitude, steepness, latitude etc.

9.6 Fractal generation

To generate terrain, clouds , soil type, vegetation etc. you need a noise function.

A large number of noise functions exist. All of which have their own strengths and weaknesses and are optimal for their own purposes. We have some rather strict demands for the noise function in order for it to work well with Level of Detail and a large scale as an entire planet.

A definition of noise is "randomness: the quality of lacking any predictable order or plan". Other definitions deal with noise as some unwanted corruption of the clear data. We will use the first definition, and consider noise as random data which exist on all scales. You can not zoom far enough into a noise function to make

it smooth, which you can with all usual functions. Noise and fractals are interchangeable in that way, and for our purpose.

A suitable noise function will satisfy most or all of the following demands.

Fast It must generate its values quickly. With a large mesh with many changes, as will be the case with a fast moving camera above a large planet, the noise function will be called often, and must not drag the application to a halt.

Work well with LOD Some noise functions are not well suited for generating local noise, as is needed for the local changes that are the trademark of a LOD-algorithm. By local we mean to generate noise values for a specific area without having to deal with the rest of the noise sample space.

Generate data usable for terrains The noise values should be readily usable as a height or opacity map. If the data contains too much high frequency and needs to be smoothed first, then that will add to the processing time.

9.6.1 Midpoint displacement

Midpoint displacement goes by other names, such as diamond square and plasma. For our use, the other names, and their slightly different approaches, are not optimal however, and we will ignore them from now on. We will consider the algorithm in its pure form, where you have a line, defined by its two endpoints, and displace that lines midpoint.

The general rule of thumb is that the magnitude of the detail decreases when the scale of the detail decreases. This means that two points separated horizontally by 100 kilometers can very well have a height difference of 1 kilometer, while two points separated horizontally by one meter will almost certainly not be offset by that distance. Exceptions to this are the almost vertical cliff sides, which does exist in nature, although they are rare. We deal with this in section 9.6.7.

To abide by the rule of diminishing magnitude of offsets, you scale the random offset in such a way that the scale is reduced when the subdivision level is increased as in (11). Others [Polack 2003] have defined the scaling function differently, but they generally are not dealing with Level of Detail, so we need it to slightly differently. In (11) the calculated offset is based on a random number between +1 and -1. That number is scaled by δ so that a δ of two halves the offset for each subsequent subdivision level, while a larger δ flattens out the values faster and a smaller δ keeps the large scale offsets for smaller detail levels. A value of less than one makes the offset grow, rather then diminish, which is generally not desirable. In other words, the terrain is rougher the lower the value of δ. The pseudo code in Listing 2 shows how this can be implemented.

$$offset = maxOffset * Rand * \delta^{-subdivisionLevel} \quad (11)$$

maxOffset, the maximal value for a level one subdivision
rand, a pseudo random function which returns a value between +1 and -1.
δ, the roughness

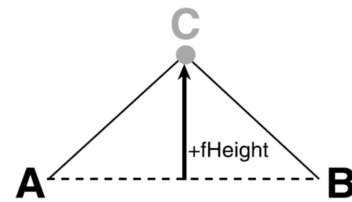


Figure 25: One level of midpoint displacement[Polack 2003]

Listing 2: Pseudo code for Midpoint Displacment

```
let L be a list of line segments,                      1
    defining the terrain, ordered by
    error.
                                                       2
l← L_Head                                              3
while l_Error > maximal accepted error                 4
{                                                      5
    p← l_Midpoint                                      6
    offset←maxOffset * random(−1..+1) *                7
        δ^(l_SubdivisionLevel)
    p←p+offset                                         8
    l_1 = new lineSegment(l_Start, p)                  9
    l_2 = new lineSegment(p, l_End)                    10
    insert L_1 and L_2 into L                          11
    l← L_Head                                          12
}                                                      13
```

Figure 26 to Figure 29 shows a "terrain" being generated by a few subdivisions of the originally horizontal line (level zero). At the first step, the lines midpoint is displaced, as seen in Figure 25. A random displacement is generated and the line now consists of two segments. The next step offsets the two midpoints on the new lines, and so on till level 6 where the terrain consists of 64 line segments ($2^6 = 64$).

Special considerations for a sphere Given the two points, in 3D, A and B, the midpoint M is not located at $\frac{A+B}{2}$, and is the case for a flat world. Rather it is located at $\frac{A+B}{2}(\frac{|A|+|B|}{2}/|\frac{A+B}{2}|)$. This is essentially a matter of scaling the vector from the planets center, which is located at Origo, to the linear midpoint, so that is gets a length which is the average of the radius of the two endpoints. If A is 6000 km from the center and B is 7000 km from the center, then M is 6500 km from the center, and is located on the line going from the center through the linear midpoint, as seen on Figure 30

The midpoint displacement conclusion This approach is ideally suited for ROAM section 6, where detail is added to the mesh by generating a new vertex at the midpoint of an existing line, and moving that vertex into position, which is *exactly* how Midpoint Displacement itself adds detail. It seems clear that this algorithm can very quickly add detail. It also seems clear that it is quite easy to use it with Level of Detail. If more detail is wanted for a certain area, then the lines, in that area, are simply subdivided more than the lines outside the area. When a line is subdivided to provide more detail, the endpoints are not changes, which is another required[9] attribute of Level of Detail. A problem that Midpoint

[9] Required by us

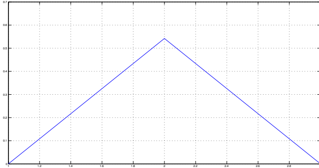


Figure 26: Midpoint Displacement at level one

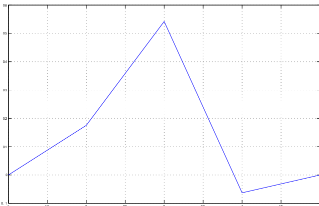


Figure 27: Midpoint Displacement at level two

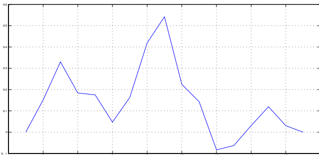


Figure 28: Midpoint Displacement at level four

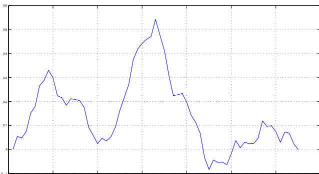


Figure 29: Midpoint Displacement at level six

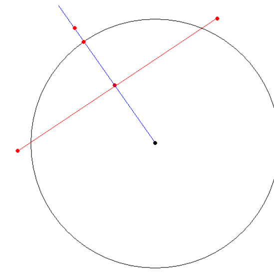


Figure 30: A different midpoint for spheres

Displacement shares with all but one algorithm[10], is its inability to fit to a vertical line. The steeper the slope, the more subdivisions is needed. The method can equally well subdivide other "random" per vertex values, such as cloud cover, vegetation, roughness.

9.6.2 Fault line

The Fault Line algorithm is inspired by plate tectonics and landslides which is responsible for the large scale terrain features on Earth.

When two plates collide, what usually happens is that one goes under the other. This is a fault line and along the line the terrain abruptly rises up or drops down.

The fault line effect can be seen on smaller levels when parts of a hill or mountain breaks loose and slides downwards as a whole unbroken piece.

The algorithm is very simple to understand. You start with a terrain of uniform height. Next you draw a random (straight) line through the landscape. The part of the landscape on one side of the line is lowered by some amount and the other side is raised by an equal amount. This is repeated a number of times after which you have a landscape as seen on Listing 3. Figure 31, Figure 32 and Figure 33 shows a terrain generated using this method.

A total number of 1024 faults have been used on all the figures. The first is the result of the raw algorithm, whereas the later two are the result of the raw algorithm combined with some smoothing. It is clear that Figure 31 shows a very jagged terrain with flat areas and intense high frequency[11]. Figure 31 does in no way look like a cross section of a naturally occurring landscape.

Figure 31 is better, but still in no way naturally looking. Figure 33 is smoothed heavily and does look better.

One characteristics of a landscape, generated using fault lines, is that it has flat peaks and valleys due to the combination of vertical and horizontal lines before the smoothing. That can actually be a desired attribute, since such plateaus do occur naturally. The smoothing can be thought of as erosion flattening out the landscape

[10] Fault Line can do this

[11] The high frequency is evident from the many vertical sides

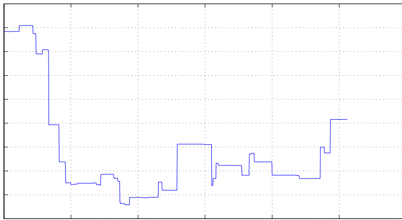


Figure 31: Fault line with 1024 faults

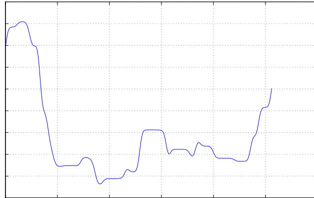


Figure 32: Fault line with 1024 faults and average smoothing of size 100

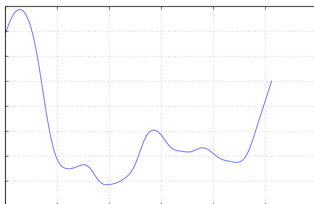


Figure 33: Fault line with 1024 faults and average smoothing of size 1000

Listing 3: Pseudo code for fault line algorithm

```
let M be a mesh defining a level terrain
let v be a vertex in M
for iter=0 to numIterations
{
    L← new randomLine
    foreach vertex v in M
    {
        if v is below L
            v.height←v.height+offset
        else
            v.height←v.height−offset
    }
}
```

The fault line conclusion The fault line algorithm has a background in nature, which can in some cases make it more realistic looking. Generally however, it needs a lot of faults to provide the finer detail, and even then, it needs smoothing. It might be useful, with a very low number of fault lines, in a multi fractal section 9.6.3, where it can provide faults on the large scale, and leave the finer details for some other function. An even worse problem, that the required number of fault lines, is that it is unsuitable for use with level of Detail. When more detail is needed in a specific region, you will need more fault lines, and you will need fault lines which pass through that region. There is no apparent easy way to generate just the right fault lines, and most likely none for real-time applications. This algorithm represents everything as vertical or horizontal lines. Any line which is sloped, will therefore be a jagged version of the true form, and the larger the step-size, the more jagged it is. Unlike other methods it *can* represent the ultimate in high frequency: a vertical line.

9.6.3 Multi fractals

Multi fractals as defined in [Ebert 2003] are fractals resulting from the multiplication of two other fractals. We will however use the term more broadly and consider a combination of two fractals, by other means than just multiplication, a multi fractal as well.

A multiplication allows one function to scale the other, which in turn can give a more complex function. A multi fractal is not a noise function on its own. It is merely a joining of other functions in such a way as to provide a richer result. Simple addition would not result in more richness, but simply a terrain with all the vertical scales doubled on average.

Let $F_1, and F_2$ be fractal functions. Interpreting each one as height values will give a mountain terrain. The combined function of $F_1 F_2$ will give a terrain where the functions are sometimes scaled towards zero and sometimes towards higher values. Where one function gives a landscape with peaks and valleys, two functions multiplied gives a landscape with peaks and valleys and sometimes rougher and sometimes softer terrain. Figure 34 and Figure 35 are such two fractal functions and Figure 36 is the combined multi fractal. On the left side of F_3 you see a smoother terrain, due to the multiplication with the low values of F_2 around that area.

Apart from simple multiplication, one function could provide input of any kind to the other function. By multiplication we have $F_3 = F_1 * F_2$ but as an alternative we could have $F_3 = F_1(F_2)$ where F_2 provides input for F_1. As an example F_2 could be used to define the roughness of the terrain generated by F_1, if ridges should occur section 9.6.4, vegetation or something else. This will be described further in section 14.2

Multi fractals are clearly good for giving some more complexity to a terrain. They are easy to use together with all other functions, since they simply scale two or more functions with each other. They do however require a little more computation, since at least two ordinary fractals are combined.

9.6.4 Ridged fractals

The inventor of MojoWorld, F. Kenton Musgrave came up with a very simple and yet very powerful method of making a smooth noise function look more interesting. The idea is to generate sharp ridges in the terrain by taking the terrains absolute value and inverting it, as seen in Figure 37, Figure 38 and Figure 39. $ridgedTerrain(x) = -|smoothTerrain(x)|$. Where the smooth terrain gently moves from some positive value to a negative value, the absolute value of the terrain will move towards zero and then abruptly change direction and move up towards positive values again. This results in sharply defined valleys. By then inverting this function, you change the valleys into mountain ridges, and you now have a smooth landscape as before, but with a number of sharp ridges. A ridged terrain from MojoWorld can be seen on Figure 41.

Ridged Midpoint displacement There is a problem with this simple method however. It is easy to understand and implement for

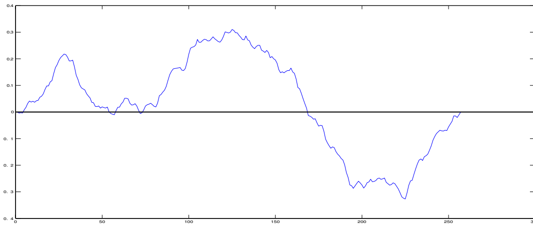


Figure 34: One fractal function generated using midpoint displacement, F_1

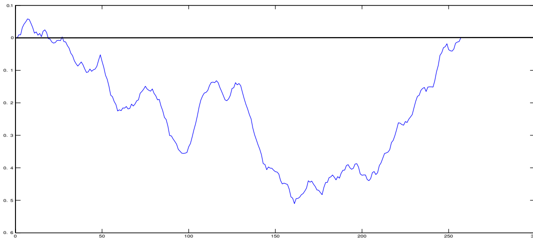


Figure 35: One fractal function generated using midpoint displacement, F_2

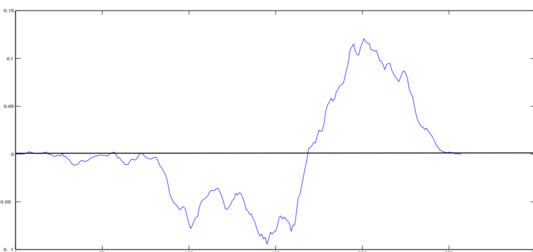


Figure 36: One multi fractal function generated by multiplying F_1 and F_2

a pure mathematical landscape function, but for midpoint displacement, it is somewhat more complicated. Consider an edge from a to b where both a and b are positive values. The midpoint m in between might be at a negative value. If it is negative, then it should be made positive to ensure that the function consists of its absolute values. This moves m up closer to a and b, but without generating the zero point crossings, that you would ordinarily get. Further refinement which generates midpoints between a and m and between m and b, will be unlikely to go below zero, and the result is a function which is flattened, but does not have ridges, as can be seen in Figure 40.

We suggest a simple solution to this problem (which will show its ugly head again when blending designed and procedurally generated terrain section 11.2), which is to keep two versions of the height values. One is the midpoint generated terrain and the other is the transformed (ridged) data values. The midpoint displacement will then work in the midpoint data, but the displayed terrain is that midpoint data passed through the ridge function $f(x) = -|f(x)|$. This will allow us to use midpoint displacement and still generate ridges. Figure 42 shows the result of separating the midpoint dis-

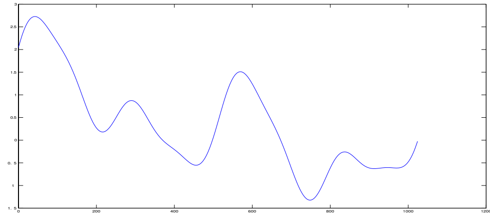


Figure 37: Ordinary Perlin noise

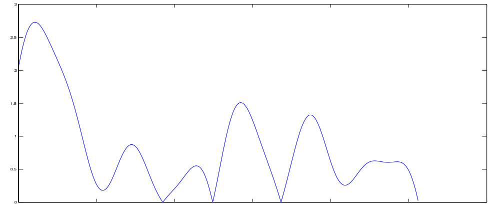


Figure 38: The absolute value of Perlin noise. The valleys are now sharply defined and no longer rounded

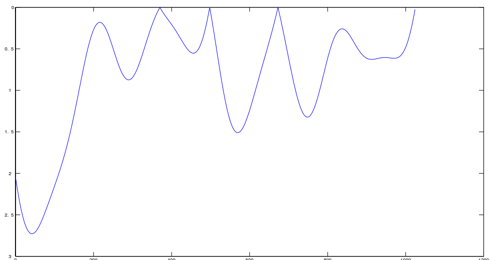


Figure 39: The inverted perling noises absolute value, which makes the valleys into sharp ridges

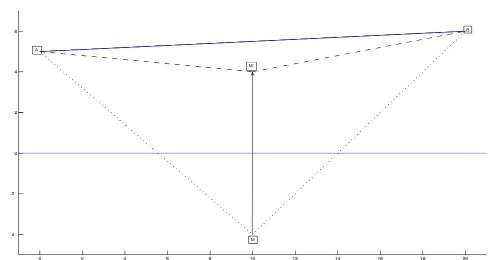


Figure 40: Midpoint displacement not behaving well when implemented as a ridged function. The midpoint M between A and B is moved up to a positive value, without generating any ridges. The function is more flat than before.

placement terrain height data and the ridge method.

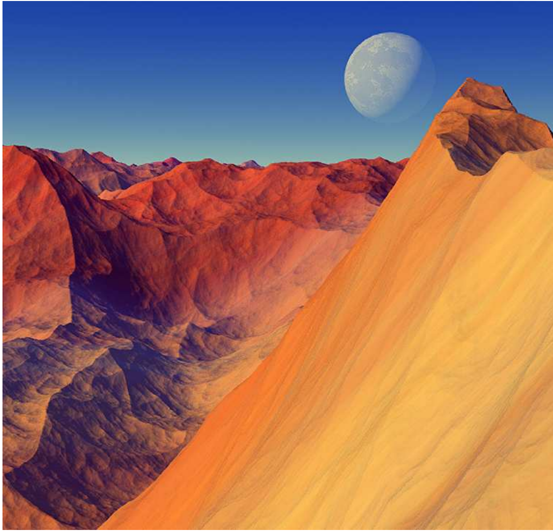


Figure 41: A scene showing ridged terrain from MojoWorld

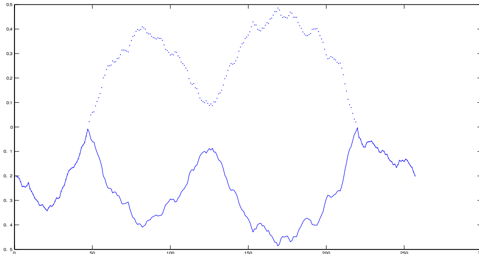


Figure 42: Midpoint displacement working correctly together with ridges. The dotted line represents the original data generated by midpoint displacement, while the solid line is the ridged version.

9.6.5 Perlin Noise

If you an unfamiliar with Perlin noise, see [Perlin 2006] and [Elias 2006]. It would seem strange to most people if "The Mother of Noise Functions" Perlin's noise function is not mentioned. Now it has been mentioned :-). We will not be using it for terrain generation. Mainly because it is rather expensive. To obtain the fine details of less than one meter, and still supply variations on the large scale of a planet, you would need at least ten octaves. That is ten summations of interpolated values, which is not something that comes for free.

You can settle for fewer octaves on the large scale, where the finer details will not be visible anyway, but when moving closer to the surface, you will need all the detail. Tricks can be used to not calculate all the interpolations when subdividing, but rather reuse the data from the level above, but by doing so, the algorithm turns into a more complicated midpoint displacement, without really giving anything in return for our trouble. Another trick could be to use a few number of Perlin Noise octaves and then use midpoint displacement, or some other method, to obtain the finer details.

Interpolation The value of a point between noise values is obtained through interpolation. Any kind of interpolation could be used, but generally it should be fast, require few sample points and still be somewhat smooth. Cosine interpolation would be a usable choice, but linear interpolation can be sufficient and generally much faster. Figure 43 and Figure 44 show the same noise but with linear and cosine interpolation. As is evident, though the images are small, the two methods does not look that different, though the image with linear interpolation has an almost invisible horizontal discontinuity along the center line.

Figure 43: Perlin noise used to generate an image. Linear interpolation was used

Figure 44: Perlin noise used to generate an image. Cosine interpolation was used

This is a subject on its own and we refer to [Perlin 2006] and [Elias

2006] for further details.

Normal vectors The normal vectors for a Perlin noise function is obtained by partial differentiation of the interpolant function. We have selected to use cosine interpolation

Another problem is that when using Perlins noise function with more dimensions, it becomes increasingly expensive. The function is for a n dimensional space $O(2^n)$. It takes 2^n noise calculations and $2^n - 1$ interpolations.

9.6.6 Coordinate system

For a planar world you need to calculate noise in two dimensions. A suitable coordinate system is easily found. For a 3D space, you need three dimensions, and that is straight forward. A 2D world which is wrapped around in the third dimension till it connects with itself is not so straight forward.

You could chose to still consider it a limited planar world, and use 2D coordinates. When you move along one dimension, the position increases until you meet the edge and start over. This is easy, but has the problem that you get a discontinuity at the edge.

Instead, you could use polar coordinates, and we do on Earth, and you would not get the edge discontinuities, since there is no edge as such. Polar coordinates present another problem however, at the poles you will see pinching, which looks like a texture edge, if a texture is used, is squeezed together into a point. This is because a change in longitude will mean less and less to the position as the longitude approaches the poles. At the poles, you have all longitudes present at once. The pinching makes polar coordinates less than optimal as well.

The third method is to consider the world a surface in 3D, which is actually is. This way there is no discontinuities, since the surface is no longer a 2D plane bent over the third dimension, but rather a 3D surface in a 3D space. It doesn't have the deficiencies of the other two methods, it is more "correct" and it can make some 3D calculations, in error estimation, easier to perform. It has one set back though - is more costly to calculate noise in 3D than in 2D. This is the coordinate system we chose to use.

Calculating the noise in 3D for a spherical planet reduces to calculating the noise on the surface of the sphere. We define a solid noise function [Ebert 2003] and calculate its values where it intersects the sphere.

9.6.7 The noise conclusion

We will be using midpoint Displacement , combined with Fault Line, as a Multi Fractal. Fault Line on its own can not provide the finer details that we desire, and Midpoint Displacement can look a little "unnatural" because the entire mesh is somewhat homogeneous without the few sharp defining edges that you see in nature.

As stated previously, Fault Line and Level of Detail are not well suited for each other, but if you consider a few fault lines, which are all pre calculated by some pseudo random algorithm, or defined by a designer, and join them in a multi fractal with another method which is LOD capable, then the faults can bring some extra dynamics into the terrain. It will almost exclusively be visible on the large scale, but it comes at a low cost.

Where midpoint Displacement can not represent vertical lines, and performs badly when dealing with very steep slopes, the Fault Line algorithm is well suited for just that. The combination will be an algorithm which can handle slopes well, but can also easily represent sudden changes in terrain; that being height or some other attributes.

9.7 Not so random randomness

During the description of midpoint displacement section 9.6 we merely stated that we used some (scaled) random number displace with. A true random number is not usable however. We need the exact same terrain to be generated each time LOD decides that it should come into existence. It is somewhat unrealistic if you go left around a mountain and see a valley, but when you return to your starting point and this time go right around the mountain, you see a large plain or something else, which is not the valley, you saw the last time. To avoid this effect, we will need to generate random *looking* values, which are actually very deterministic.

The above mentioned randomness problem is not relevant for *most* users procedural terrain generation techniques, since they are generally used in a top down fashion, where you generate the entire dataset (terrain, color patters or something else) once and for all. Then perhaps LOD is used to simplify the data, but they do not generate, destroy and regenerate data for a given region, as we do it.

9.7.1 Randomness from hashing

To obtain the deterministic pseudo random number used in displacement, we need a function H(x,y,z) which guarantees that $H(x_1,y_1,z_1) = H(x_2,y_2,z_2)$ if $x_1 = x_2$, $y_1 = y_2$ and $z_1 = z_2$. Further more it should generate a random sequence for $[H(x_1,y_1,z_1),H(x_2,y_2,z_2),...,H(x_n,y_n,z_n)]$ even though $x_{1..n},y_{1..n},z_{1..n}$ are strongly correlated.

Any good hash function will satisfy the above requirements. An extra requirement we must introduce is that the function should have a small memory footprint and should be able to execute quite quickly. Further more, it should be seedable, so that the same function can be initialized with a different seed and then generate a different hash value for the same input.

Hashing with lookup tables One of the easiest, and yet powerful, methods of hashing values is to use pre generated permutation tables and do a sequence of permutations and lookups. A problem with this method is that the hash is cyclic, so that $H(a) = H(b+\delta)$ though $a \neq b+\delta$ for some δ. δ is the functions period, and after δ sequential hashes, the function repeats. The result of repeating hash values is that the terrain tiles and has visible patterns. The goal is then to maximize δ without incurring to high a penalty in computational cost and memory usage.

The permutation scheme is illustrated in Listing 4. The permutation table contains random numbers generated with an ordinary pseudo random number generator. It can be random numbers or a sequence of numbers in random order. When using a table P_1 to index into another table P_2, then P_1 should contain values from 0 to $length(P_2) - 1$ in a random order. If a table size of 8 bit (256 entries) is used and the resulting hash should be a 32 bit value, then four tables of 256 could be used and the results can later be shifted and or'ed together like in (12), where H_9^n denotes the n'th 8 bit hash function. Should the a 32 bit floating point in the interval -1..+1, then a simple scaling can be used $(\frac{H_{32}(x)}{(2^{32}-1} - \frac{1}{2}) * 2$

$$H_{32}(x) = H_8^1(x) << 24|H_8^2(x) << 16|H_8^3(x) << 8|H_8^4(x) \quad (12)$$

The function is deliberately constructed to have a very short period, in order to demonstrate the problem. In the code, it is obvious that $P[xmod5] = P[(x+5n)mod5]$ for any n, which gives the function a $\delta = 5$ for x. The same is the case for y and z. Longer periods can easily be generated by making P larger, but that takes up more memory which in turn can push the table out of the cache and make the function much slower. If the space, we want to hash coordinates in, has a size of 20 000 by 20 000 by 20 000 kilometers and we need different hash values for positions one meter apart, then the table should have a length of at least 20 000 000. Certainly more than we can fit into the cache.

Listing 4: Hashing with permutation tables

```
P is a table of 5 random numbers                1
function hash H(x,y,z)                           2
a←P[x mod 5]                                     3
b←P[ (a+y) mod 5]                                4
hash←P[ (b+z) mod 5]                             5
```

Long-period hash functions The paper [Lagae and Dutré 2006] describes a very simple, and intuitively easily understandable, method to generate much longer periods.

The principal idea is that by adding together two hash functions with period δ_1 and δ_2 you will have a resulting function with period $\delta_1 * \delta_2$

It is easily verified with a simple example.

Given two "hash functions" H_1 and H_2 where $H_1(0..1) = [1,0]$ and $H_2(0..2) = [0,2,1]$, you have $\delta_1 = 2$ and $\delta_2 = 3$. A sequence of numbers n=0..20 will hash into the following table

	0	1	2	3	4	5	6	7	8
$H(n)$	1	0	1	0	1	0	1	0	1
$H(n)$	0	2	1	0	2	1	0	2	1
$(H_1(n)+H_2(n))\%3$	1	2	2	0	0	1	1	2	2

It is evident that H_1 and H_1 have extremely short periods of 2 and 3, while the period of H_3 is 6 - the multiple of the other two. For this to work optimally it is required that δ_1 and δ_2 are prime to each other, and that they are close to each other as well. If they are factors of each other, then the combined period will not be maximized, but rather be the maximum of δ_1 and δ_2. If they are not close to each other, then the period can be n times the longest period, where n is less than the shortest period. As an example take two functions with period 150 and 100. They are prime to each other, but the combined hash has a period of only 300, after which the sequence is repeated, since 150*2=100*3=300.

In [Lagae and Dutré 2006] it is mentioned that the period lengths should be relative prime and close, but the actual obtained combined period length is not defined. It is clear, however, that $lcm(\delta_1, \delta_2)$ defines that period length, since this is the period after which both periods are restarting. By keeping the period lengths close (relative to their lengths), you ensure that $lcm(\delta_1, \delta_2) = \delta_1\delta_2$. The easiest way to pick good lengths, is to simply select neighbor primes as period lengths. This is done at initialization, so there is no real-time overhead from calculating primes, if that was even considered expensive.

The conclusion is that two short-period hash functions can be combined into a long-period hash function, with a period of $\delta_1\delta_2$, by adding them together, *if and only if* δ_1 and δ_2 are prime to each other and close.

10 Are landscapes really fractal?

One of the properties of fractals is "self similarity". This means that the fractal looks the same at all scales. Given an image of a fractal, you can not determine at what scale it is seen. This is said to be true for landscapes as well. A mountain has smaller mountains on it, which have still smaller mountains and so on. While this may seem true at first, you should soon realize that a mountain on the large scale is very different from a small segment of the same mountain. A real mountain needs to have a certain base in order to grow to a certain altitude. For smaller rocks, that base does not need to be the same size. A large mountain can only have overhangs up to a certain size before they break off, while smaller rock formations are more solid. The roughness is not the same on all scales either. The same observations goes for the other "fractals" in nature. they are *not* the same on all scales.

An example of where one could get into trouble is the midpoint displacement algorithm section 9.6.1. It common to use a scaling of $\frac{1}{2}$ for each subdivision. If we offset by a maximum of M at the first displacement, then we offset by up to $\frac{1}{2}$M the next step and then 0.25M and so on. If we have a planet where we want the maximal offset to be 10km, then that is M=10 000m. If the line segments are 10 000km long at the first step of the algorithm, then we have M=5.000m when the lines are 5.000km. This way our M will be 1m when the line segments are 1km. This translates into a very flat terrain. Over a distance of one kilometer it will not move more than 1 m up or down. Nowhere on Earth, except perhaps on Artica, will you ever see terrain like this.

The alternative could be to scale by something other than $\frac{1}{2}$, but that would not be right either. On the small scale $\frac{1}{2}$ could be just right. If the terrain can be displaced 10m over 100m, then 5m over 50m still sounds reasonable. It becomes 1cm over 10cm, which now perhaps again is a little too flat.

The point is that since the terrain is not the same on all scales, then neither should the fractal function be. We have introduced various methods to give the landscape some more dynamics section 9.6, but neither deal with the differences based on scale.

The result of this is that a fractal method need something extra to make it generate good landscapes. That something extra could be a more complex scaling or perhaps utilization of the fault line method or pregenerated mountain ranges.

11 Design and procedural generation

When using a procedural method to generate terrain, you are at the same time aiding the designer by letting a computer do some of the time consuming and boring work, but you are also taking away some control from the designer on how the final result should look. Solving this conflict is not easy, but we discuss a possible solution to part of the conflict in the following.

Ideally, a designer would wish for the maximum range of possible control over the generation of terrain.

The designer could wish to explicit define how every little detail in a terrain should look. The reasons for this are many. Maybe the designer wants to model a precise copy of a scene from real life. Maybe the terrain should have a overall layout that accommodate the game play in the best way. Placing static meshes, like buildings, in the terrain might lead the designer to want to take control over the terrain at the given position.

On the other end of the scale the designer would not wish to get involved in the generation of terrain. The resources to be used on designing the terrain could be used elsewhere. Or maybe the procedural generation satisfies the needs for terrain. Letting the computer generate terrain procedurally might even produce surprising results which designer wouldn't think of on their own. Thinking outside box might produce new ideas leading to development of new solutions.

Somewhere in between the designer might want to control the general look of the generated terrain. For this, the parameters for the procedural method might be suitable as mentioned earlier in section 9. These controls could even combined in a way which gives the designer a intuitive tool to manipulate the procedural generation, not by modifying the parameters used in the procedural method, but at a higher abstraction layer. Predesigned worlds is an old topic which has been widely used to make computer games and 3D-representations of a real environment, which has been sampled. Purely procedurally generated worlds have been around for quite some time as well. Having predesigned worlds with a limited resolution, and then filling in fractal noise for that extra detail has also been done before. What is more of a new subject is how to blend design and procedural generation seamlessly.

It is often desirable to be able to generate a procedural world, which can be huge and still possess rich details, and still be able to define specific landscape features at certain areas, and to do that with absolute control, beyond simply raising and lowering the terrain.

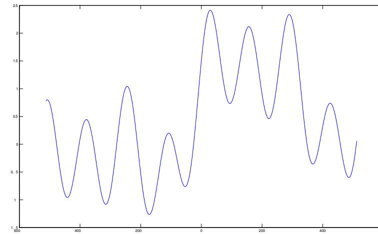


Figure 45: Noise function, $n(x)$

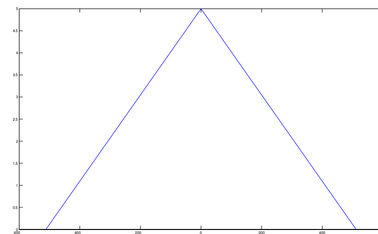


Figure 46: Designed function, or rather predefined samples in x, $d(x)$

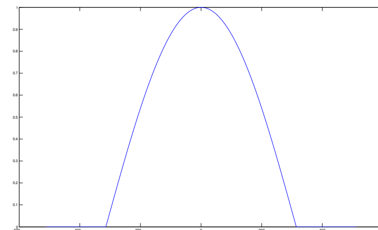


Figure 47: Scaling function, $s(x)$

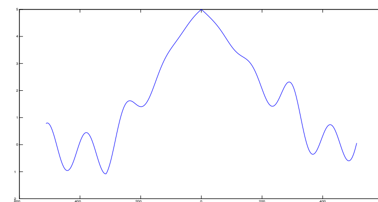


Figure 48: Sum of scaled functions, $n(x)(1-s(x)+d(x)s(x)$

Merging design into the world We first introduce the concept with 1D examples before dealing with 2D and how to project the design into the world.

Given a noise function n(x) and a designer generated function (or rather samples of the function) d(x), we seek to generate a seamless blend of the two. A rather obvious way to do this is to scale them and sum them together. With a scaling function s(x) in the interval 0..1, we could do as simple as (13).

$$blend(x) = n(x) * (1 - s(x)) + d(x) * s(x) \quad (13)$$

Figure 48 shows the result of blending a noise function with a predefined triangular shape, with a cut-off cosine blending function. Where the scale is near 1, the triangle is dominant, while it fades away when the scaling is near 0.

11.1 The blending function

Generally you will want your designed terrain to be represented perfectly in the final rendering, without any alterations. You will also want to leave the terrain, far away from your designed area, entirely up to the procedural generation. The area near the design, but not in the important part of the design, should be a blend between the procedurally generated and the designed.

A square blending function with the value 1 over the design, and 0 outside, will let the design show without errors, but the edge will be

clearly visible, while a cosine blending function, as shown previously, makes a smooth transition, but also does not enforce the design. A reasonable compromise would be a trapezoid shaped function with a smooth ramp on the edges. Figure 49 shows these three methods. It is evident that the square function is too simplistic, but the choice between the smooth and the trapezoid is very much dependent on what the designer wants. It will therefore generally be the better choice to let the designer select both the design function as well as the blend function. Perhaps a very smooth transition is more important than absolute representation of the design. Figure 49 shows blending with a square, a cosine and a trapezoid shaped blend function.

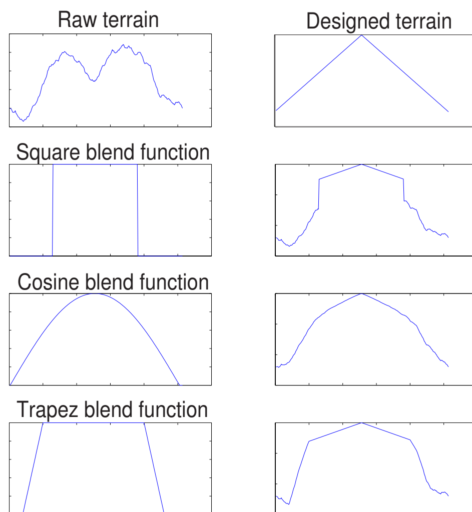


Figure 49: Blending performed with three types of blend functions

Optimal blending function There is no such function. Generally one will want a central area where the design has absolute dominance and around that a slow blend into the surrounding terrain, but sometimes one might want the design to appear more eroded and rough, in which case it should not be so dominant, but rather let the fractal function work on it. The only sensible thing to do is to let a design consist of the design itself and blending data as well. Then the designer can chose the best method.

11.2 Midpoint displacement and design blending

When using midpoint displacement, you do generally not have a mathematical function representing the terrain. Rather you have a large number of random offsets of the midpoints. You can therefore not just define functions and scale and sum, as shown previously. Instead you need to generate a new midpoint displaced point and then blend that with the designed/sampled function at that exact location.

The method is still simple. You locate the midpoint of the edge to be refined. The the midpoint is displaced in a pseudo random way as explained in section 9.6.1. This gives you the point M with position x,y. Next you look at the design function $d(x)$ and the blend function $s(x)$ and blend the displaced midpoint with the pre-designed value as in (14). This new value of M_y is then used as the true blended noise and design value, as seen in Figure 50.

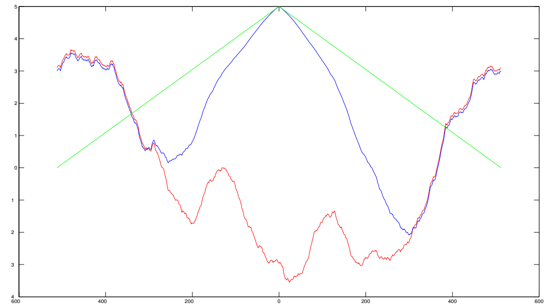


Figure 50: Midpoint displaced values in red, design in green and combination in blue

$$M_y(x) = d(x) * (1 - s(x) + M_y * s(x) \tag{14}$$

When using midpoint displacement rather than pure mathematical functions, the pre-designed data will generally influence a larger area than what you would expect. If an edge starts outside the design area but ends inside the designed data, then even though its midpoint might be outside the design area, it can still be pulled up or down based on the endpoint which *is* inside the designed area.

The larger region of influence can be a desirable feature, and generally is, because it ensures that the design blends very well with the procedurally generated landscape. It can also become too uncontrollable however. A reasonable alternative is to generate the terrain with midpoint displacement, while ignoring the design entirely, and first introduce design when the data is to be written into the graphics card. Just then, we can blend the terrain with design. This ensures that the design does not inadvertently influence terrain outside its designated area. It does have some drawbacks though. Mainly we can not optimize the mesh subdivision to take the design into account. The design might make abrupt changes in the terrain without ROAM being able to take those changes, and the following pixel errors, into account because those changes does not exist before the terrain is done being optimized. Secondly, the design will be less integrated into the whole planet, than what it could be with blending from the get go.

11.3 2D area design

Until now we have been dealing with an abstract form of blending in 1D. Most of the concepts can be carried directly over to 2D, which is what our planets surface is, but some need an little extra work. Mainly we need at method to position design on the planet, so that we get a one to one relationship between a position on the planet surface and a sample point in the design.

The data values for design and blending will be contained in a 2D structure. This can be a bitmap or a raw array of data. This is really a matter of which is more convenient. For a designer a bitmap will be an easy place to draw and store a height map, while on the other hand actual measurements from the real world, will often be stored in a more raw format. We have tested with both home drawn bitmaps and real world data downloaded from

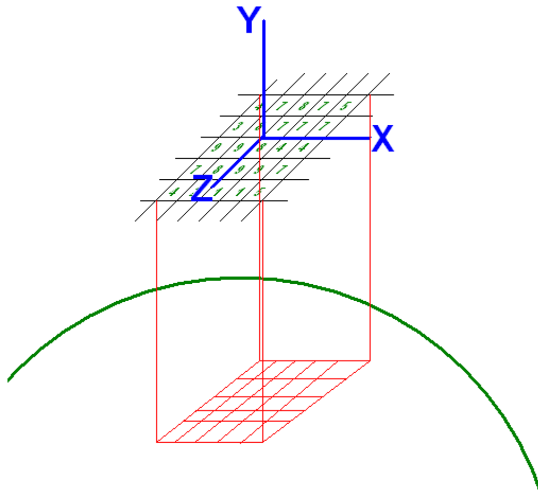


Figure 51: Design space and it projection to a sphere

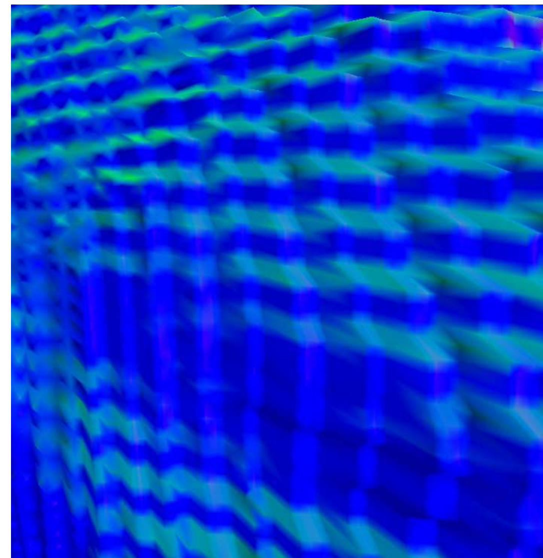


Figure 52: Closeup of blocks caused by pointsampling. The colors are defined to be the normal vectors to better show the form of the mesh

`http://seamless.usgs.gov/website/seamless/viewer.php`

11.3.1 Design space

In order to position and scale the design in the 3D world, we define a design space, as seen on Figure 51, with its own coordinate system. The X- and Z-axis are considered ”right” and ”up” in the design. The design data is located in the XZ-plane and the Y-axis is going up from this plane. All three axis are unit length and the x- and z-axis correspond to the data in such a way that -1,-1 is the first element in an array of data, while 1,1 is the last. The choice of scaling is arbitrary.

A design is placed in the 3D world by selecting a position for it as well as an orientation. The design then influences the terrain directly below it, but only to a certain depth, which is selected to be the spheres radius. This is to ensure that the design can be placed in such a way that it only influences one side of the sphere, and does not design its way all the way to the opposite side.

11.3.2 Sampling

When a 3D position is generated on the sphere, the design is asked if it has any influence at that position. This is done simply by projecting the 3D position into design space and testing its X- Y- and Z-coordinates. If X or Z are less than -1 or greater than 1, then the position is outside the designs area of influence. If the Y value is less that the defined threshold, then the point is also considered out of the design area.

If, on the other hand, the point is inside the design, then the design and blending data is sampled by converting the coordinate into array coordinates. The design will rarely have the exact same resolution as the points being designed, so some form of re-sampling must be used. We have opted for bilinear interpolation where the four closest values are being used to calculate a weighted average. Point sampling, where only the closest value is used, will give artifacts whenever the terrain mesh is finer than the design, as seen in Figure 52. The resulting mesh will then appear blocky, as it happens for images. Bicubic filtering uses the 16 nearest values and is generally considered better, but it is more time consuming, and we will settle for a simpler method now. The only real requirement is that some form of interpolation should be used, so that we avoid blocks, which bilinear lets us do, as seen in Figure 53.

On Figure 54 a sample point is located near the center of the grid. it does not land exactly on any of the values in the grid, but it is closest to 4,3,8 and 4. Had point sampling been used, the resulting sample value would have been 3, since this is the closest value. With bilinear sampling, we first interpolate between 4 and 8 as well as between 3 and 4. The resulting values are approximately 5 and 3,3. Next we interpolate between those two values and get approximately 3,6. The value 3,6 is the final result. The interpolation is linear, as implied by the name bilinear.

11.3.3 Blending

Whenever a pixel is in the area controlled by the design, we need two values from the design. We need the actual design value and we need the blending factor, which is a number in the interval 0 to 1. Given those two values and the fractal value generated for the given vertex, we simply blend them and arrive at some mix, which is hopefully exactly what we were looking for.

Scaling the blending factor can provide very different end results. A meteor crater is defined by a design as seen in Figure 55 and then places in the world. Figure 56 and Figure 57 shows the design with three different levels of blending. The top images is the design blended very hard, so that it is absolute where it is defined. The second image is the design blended more softly, so that the random terrain underneath has more influence. The last image is a very very faint design, which is not even visible. It is what the terrain looks like if it is not touched by design.

Another example of a more distinct design is shown in Figure 58 which is clearly not naturally looking. Even though it is such a ”hard” design, it can be blended with a terrain to make it become one whole. In Figure 59 the design is blended in way which lets it stand out very clearly, while it is more faded in Figure 60. The faded blend resembles an eroded version of the test design.

Figure 53: Closeup of bilinear sampling showing no blocks. The colors are defined to be the normal vectors to better show the form of the mesh

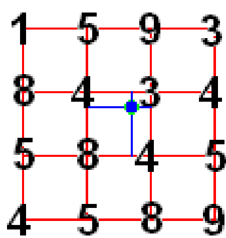


Figure 54: The bilinear resampling

Figure 55: A very rudimentary design for a meteor crater containing only the outer rim and the central point

11.4 Design and visibility

As mentioned earlier, in the section about bounding spheres section 7, a triangle should be contained inside a bounding sphere, from which none of its child triangles will ever escape. If a triangle can be subdivided in such a way that one of its children is outside the initial bounding sphere, then visibility testing will fail.

This can be a problem when dealing with design, since we have no upper limit on the distance the design can offset any given triangle. It can very well move a triangle outside its parents bounding sphere. This will cause the design to vanish from view when only the design itself is in the view frustum, and the triangles which are influenced by the design are not.

Figure 61 shows an example of this. From the viewpoint, the red design can be seen, while the rough terrain below can not. None of the bounding spheres are in view, so the rough terrain does not get refined, since it is not in view, and none of its child triangles will ever be. section 7 shows why this is certain. If however the terrain was refined, then the terrain would kick in and offset it enough for it to come into view. This is a situation where we should see the red mountain, but we do not. This is obviously caused by child triangles, of the rough terrain, being offset outside the bounding sphere. Something which we have previously stated that we can not allow. When dealing with design, it is next to impossible to prevent without seriously constraining the designer.

We see two solutions to the problem.

Expand the bounding sphere A triangle which realizes that it is entirely or partly inside a design area could expand its bounding sphere. It is difficult for it to know how much is enough, without sampling for all design for design and blend values. A design could in theory offset child triangles any distance, and we wouldn't know before it happened. If a design knows what its maximal offset might be, then we could still expand the sphere for triangles influenced by design. A design could inform us about its lowest and highest values and we could then expand the bounding sphere out at least that far. It would probably be too much, and the triangle would be considered in plain view - and be refined - too often, but we would be rid of the artifacts.

Always refine the designed geometry Whenever a triangle is touched by design, it could be split. This would go on till some predefined level was reached. It would mean that even when the rough mesh is not visible, as in Figure 61, the mesh would be refined, and the designed mesh would come into view. The risk we run with this method, is that perhaps the rough mesh was not refined just enough.

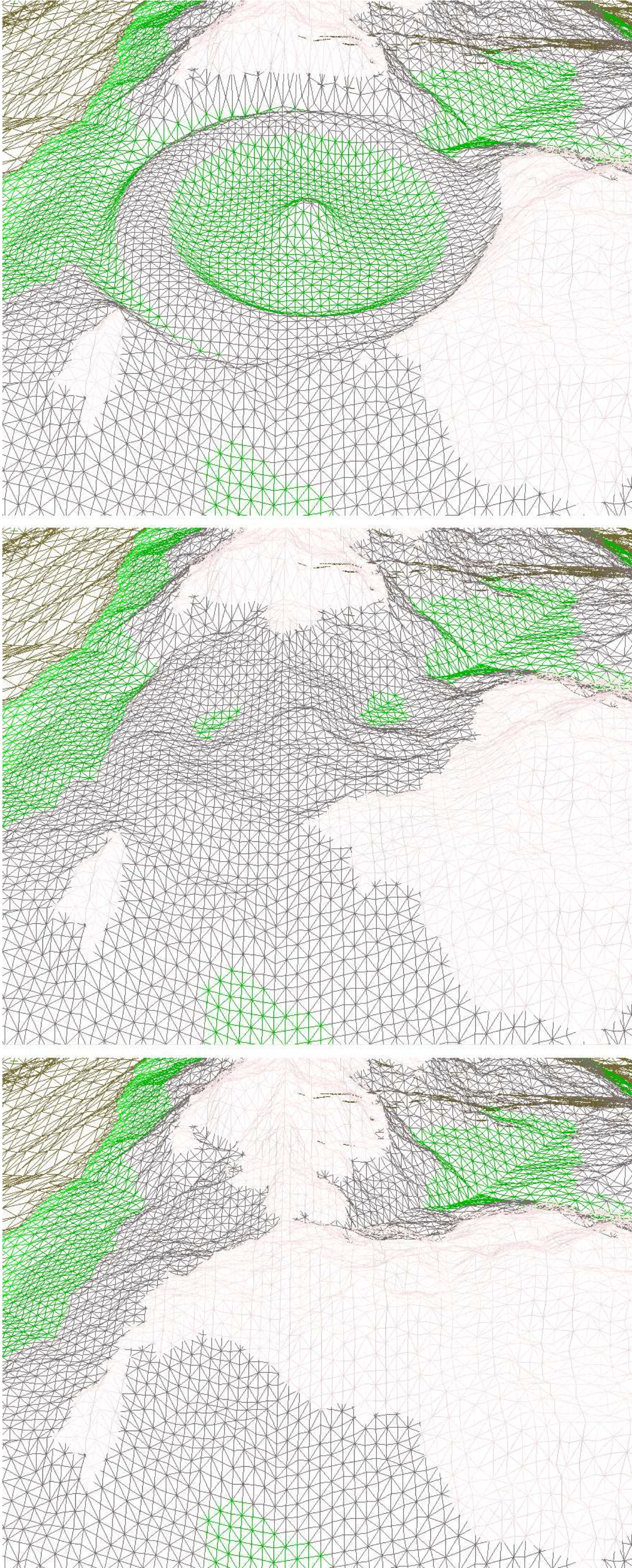


Figure 56: A crater blended into a terrain at three different levels. From top to bottom the blend is 1.0 0.5 and 0.1➠

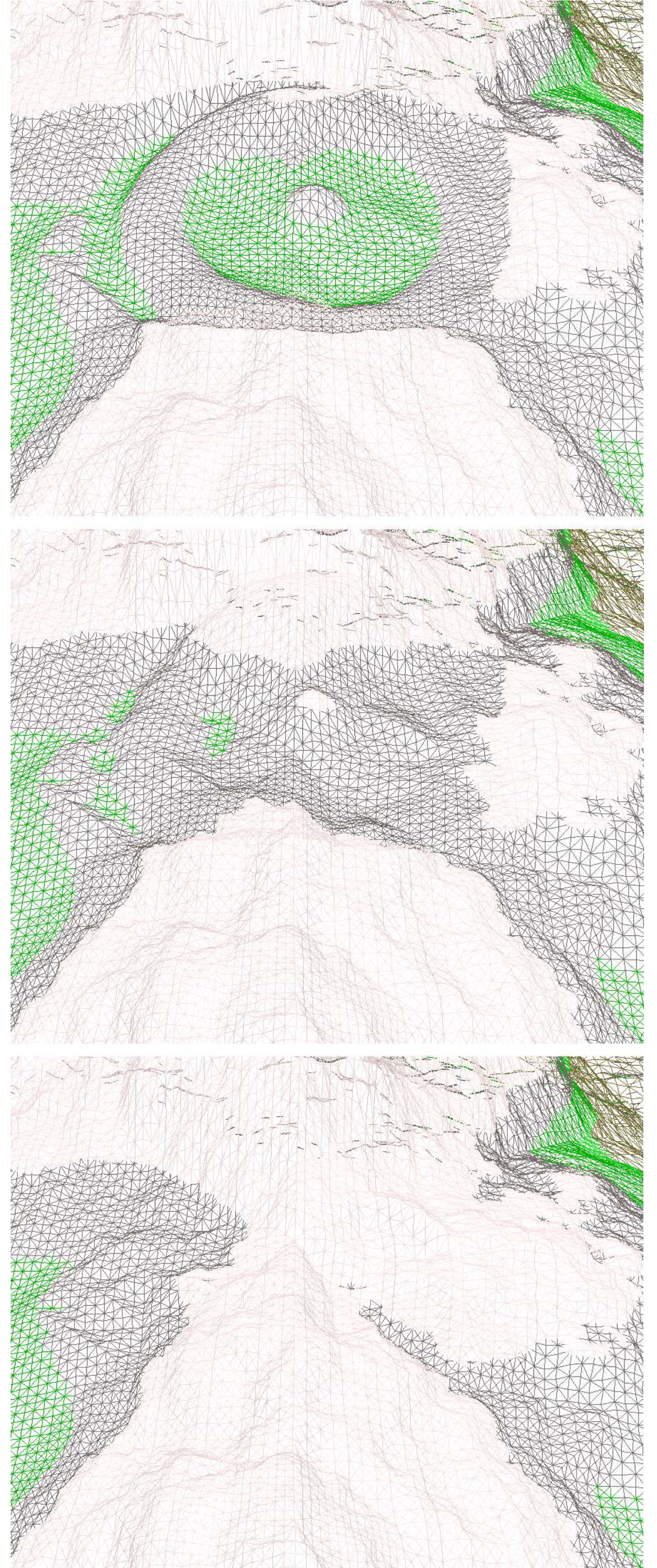


Figure 57: A crater blended into a terrain at three different levels. From top to bottom the blend is 1.0 0.5 and 0.1➠

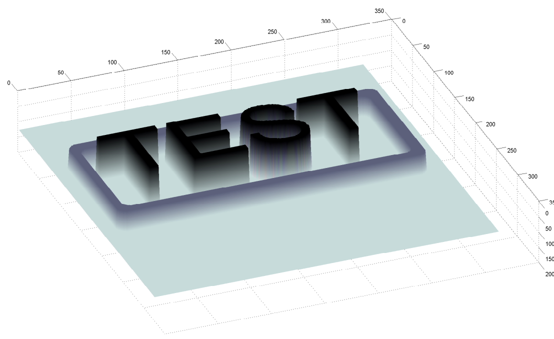


Figure 58: Manually generated height map used as design

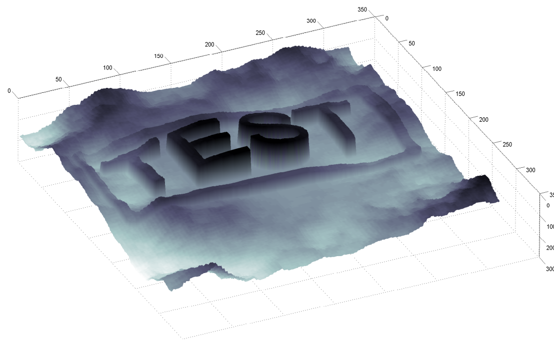


Figure 59: Design merged with noise, using the smooth blend function. It is apparent that the resulting landscape is never entirely equal to the design, though it is very close around the center, where the blend function gives heigh weight to the design

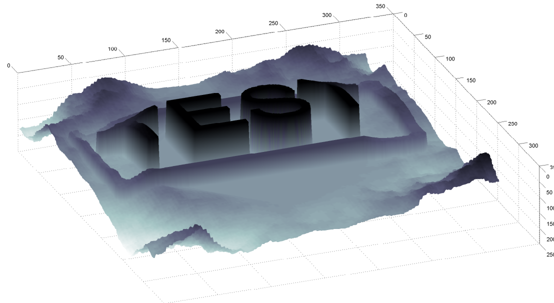


Figure 60: Design merged with noise, using the blend function with a central area totally dominated by the design. You can clearly see that the E and S are not influenced by the noise function, but are entirely defined by the design

This will in most cases not be a problem, if we refine just a few extra steps down below what we would else do. This is because that even user defined landscapes, defined through design, should behave relatively relaxed. When we generate some of the designed geometry, then it will get its own bounding triangles, which tells us when the triangles are in view of not. If *those* bounding spheres

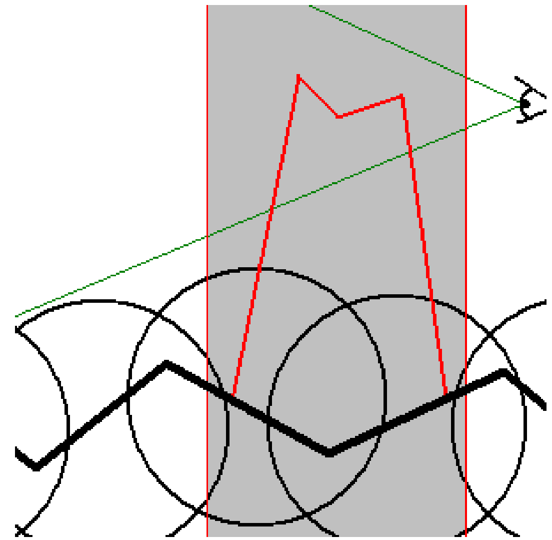


Figure 61: Design should be visible, but the triangles, which are to be transformed into the design, are not

are not broken by subsequent child triangles, as the previous ones were, then we are back in the game, and we will again refine what is visible. Figure 62 shows an example of this. The rough mesh is not in view, and therefore should not be refined. We do however refine at least one step now since the design area is influencing us. With the next subdivision we get the flat top red mountain. That mountain is still not in view, *but* its bounding sphere tells us that some of its children might be. Now we again refine, as we would always do, and the blue mountain peak shows up in the view field.

We opt for the refinement-ahead-of-time method, since it very easy to implement, and will not break. It will require a few subdivisions ahead of time, but not a lot, and the method of the expanding bounding sphere will, with a huge expanded bounding sphere, also be subdivided much more than it would else.

11.5 Blending in the vertex shader

The blend operation, we have described previously, would fit well in a vertex shader. The vertices arrive at the shader with the position they acquired from out fractal method, and next we blend that position them with some design and forward them with the new position. That is easy and it is fast. There is a problem though.

If the terrain is first blended with the design after we are actually trying to render it, then we have no way of optimizing the ROAM so it takes the design into account. Perhaps the design is very complex, and we need many triangles to represent it properly, but we can not realize this before the vertices arrive at the shader, and then it is too late.

Also note that you would need a vertex shader 3.0 or higher to do texture lookups in the vertex shader itself. Without that functionality, it would be difficult to do the design blending at all on the GPU.

Though the vertex shader seems like a natural place for blending, we can not use it well for our kind of terrain. If we used a mesh which was not taking terrain details into account when optimizing, then it would work well to define the terrain in a texture and project it onto the terrain. The texture coordinates would not even be a

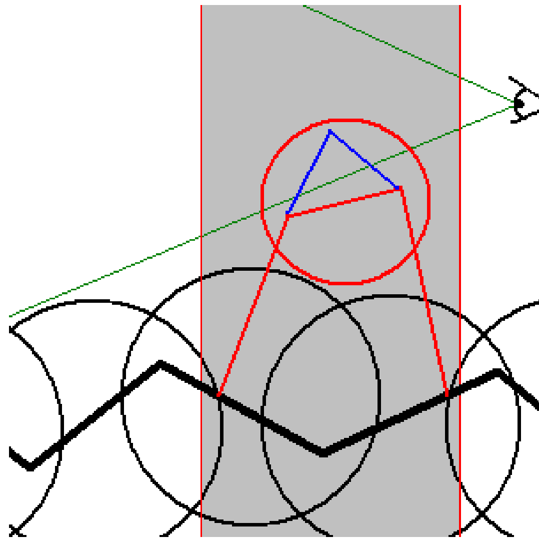


Figure 62: Design is becoming visible when we do an extra subdivision. Note that this is not the same terrain as in Figure 61

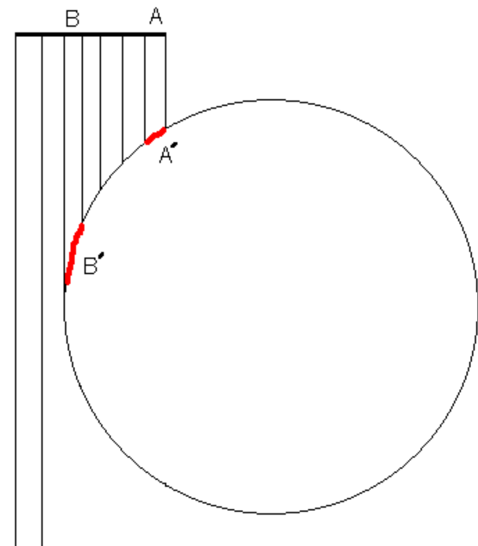


Figure 63: Design can be distorted when angle of "impact" is very low

problem when doing a projection. It would in fact be very much like that we are doing now, only on the CPU.

11.6 Distortion from projection

When a design is projected onto a planar surface, along the surfaces normal vector, then the projection is perfect. When it is projected onto a sphere, then it can be anything *but* perfect. As seen in Figure 63 a design will be stretched out when the angle of "impact" is very low. While a and B in the design are equal in size, their projections onto the planet, A' and B' are far from equal. This happens on steep mountainsides as well.

It can be used to create special effects, such as elongated meteor craters, and it can distort a design, if the design space is very large or if it is placed in a way that makes it project in a shallow angle. The solution to this "problem" is generally to not make that large designs and to not let them project along a vector very different from the general normal vector at the center of the target area.

11.7 Interactive real-time design

The design previously described is all done off-line before the actual visualization is started. Another form of design is done real-time by user agents inside the virtual world. The question one need to ask one self is, what do we do if this user agent in the virtual world drops a large bomb on that patch of grass?

Older computer games did nothing much really. An explosion was shown, but then the smoke cleared, the terrain was as before. Newer games draw a decal (extra detail texture showing some effect, such as bullet holes or scorch marks and tire tracks) on the terrain to show the result earth. The only game, that we know of, where the terrain is actually altered and the is a large hole afterwards, is the eighties game Red Fraction, where some, but not all terrain, could be destroyed.

We propose a method which will allow user agents to actually affect the world they exist in. Letting the terrain be altered is very easy. A mesh exists and it can be modified. That would however mean that the terrain returns to normal if we leave the area and return a while later, since the mesh is constantly being transformed by the ROAM-algorithm. What we want is to be able to blow a hole in the ground and go away and let some other user agent arrive at the scene and see the exact same hole. The effects should be persistent. We see two general ways of achieving this goal.

Real-time generation of ordinary design When a bomb goes off on the surface, we can dynamically calculate a design which will look like a bomb crater when projected unto the site of the explosion. First we need to make some procedural algorithm for generating bomb crater designs. Next we have to blend it just like any other design. This means that we need to have design generating methods for all the things a user agent can do to the landscape, which is not unreasonable. If we have not considered what happens when a nuclear explosion touches the landscape, then we can simply not let the users use those devices. If however we have decided on the effect from a falling meteorite, then that can happen.

Real-time recreation of the action Rather that generating a design and imposing it on the landscape, we could remember the action and recreate it whenever that part of the landscape comes into existence again. If a user agent drops a certain kind of bomb at a certain location and create a certain feature in the landscape, we could simply remember that kind of bomb was dropped at that position. When the landscape is later observed by a user agent, we "drop the bomb" again before showing the landscape. This will only require us to store very few details about the event, but it will require the effect to be recalculated whenever the terrain is recalculated. It will even have the action replay whenever the bomb crater is refined, when a user agent moves in closer.

Memory? What memory? If the world have to remember all effects from the start of time till the end of time, then it can be a tough job keeping room for it all. The question is really how long an effect lasts. We have craters from meteorites that "landed" many million years ago, but we do not have ordinary bomb craters which

are more than a few years old. It seems reasonable to let actions fade out over a period of time. The more significant, the longer they last. The question is then how long is long enough. As so often before the answer is ”it depends”. If the system running the virtual world has plenty of resources then they can last a long time. If only few actions are executed, then they can be remembered a long time. If not, then they should fade quickly.

12 Landscape features

In this section we describe some of the characteristic landscape features. These features are common for the Earth and for other planets as well. The following landscape features are of the larger scale and influencing terrain represented both at low and higher resolution.

To generate a realistic looking terrain the following landscape features are of considerably importance.

- Dry land
- Oceans and lakes
- River systems and lakes
- Mountain ranges
- Vegetation
- Wild life
- Civilization
- Atmosphere
- Shadows
- And as a consequence of the above, erosion
- Meteor crates

In the following sections, we will describe these landscape features in more detail and discuss how they can become a part of computer generated terrain.

12.1 Dry land

On Earth (and on other rock planets) on a macroscopic scale, continents of different sizes and shapes are the most conspicuous landscape feature. Larger continents have recognizable forms which allows us distinguish them from each other. Continents are a result of the underlying dynamics of tectonic plates, moving and altering the shapes of continents.

On a slightly smaller scale another predominant landscape feature are mountains. They are also a result of plate tectonics, resulting in mountain ranges where plates converges or diverges. Also volcanic activity results in volcanic mountains or even small islands.

The main land often consists of characteristic landscape as mountains, plains, plateaus, coastal areas etc. These landscape types typically occupies a coherent area due to geology and external factors like climate or erosion.

When creating a natural looking terrain taking these features into consideration would be advisable.

12.2 Oceans and lakes

70% of the Earth surface is covered by water, most of it being oceans. Oceans are largely homogeneous surfaces, maybe varying little in color. The water is covering land that is generally formed as described above but with some exceptions like erosion.

Where land emerges from the sea we have a pronounced landscape feature. Here the surface of the water and the land can be close to parallel and as will described later, this could cause visual artifacts when rendering if not handled correctly as described in detail in section 16.

Oceans could be represented by a simple spheric surface with a more or less constant radius. In this way, water will show where the land has a height which that is below the level of the spheric water surface. If one is to dive into the ocean, there is land under the ocean as well, since land and ocean surface are separate objects. The problem with this is that the ocean is world wide at altitude 0. No land can exist at a lower altitude, though there are areas on Earth below sea level. Lakes can exist at higher altitudes and need special consideration, which will be described in the river section.

Water could also be represented simply by coloring pixels below a certain height, blue, but one has to be careful if not shading per pixel as artifacts know as leaking might occur. This is when a triangle has a vertex under water and one above. If the submerged vertex is colored blue and the one above water is colored green, then the blue and green will blend into each other along the edge, generating a smooth color gradient rather than a sharp change from blue to green when the edge breaks the water surface.

12.3 River systems

River systems and their effect on the terrain are significant for computer generated terrains. Where and how rivers flow are a result of the shape of the terrain, running from higher to lower ground. But the flow of the water also alters the shape of the terrain. This can be seen when rivers are carving their way down in mountains and where the rivers run into the ocean, slowing down and releasing sediment at the coast.

In the late 1980's fractal terrains where lacking of river system, which made the terrains less believable. Efforts where made to solve this lack of reality. Two different approaches to solve the problem was developed shortly after one another. These two approaches are sometimes called bottom up and top down and described briefly in the following.

In 1988 F. K. Musgrave, C. E.Kolb and R. S. Mace, [Kelley et al. 1988], presented their solution on how to incorporate river systems in computer generated terrain. They took on a bottom up approach where a river system was created first and then generation a fractal terrain on top of this river system.

A year later, 1989, F. K. Musgrave, C. E.Kolb and R. S. Mace, [Musgrave et al. 1989], presented an alternative solution, a top down approach, where a fractal terrain was generated first and next a simulation of water flowing down this terrain and eroding it.

None of them are exactly representing what has happened on Earth (or any other planet), but the end result are close to what is seen in real life.

Some effort has been made to create solutions where erosion from water on terrain could be calculated real time. In 1993 [Prusinkiewicz and Hammel 1993] P. Prusinkiewicz and M. Hammel describes yet another way of generation terrain with river system. They introduce a method where generating mountains and rivers happens simultaneous which one could argue is closer to a natural process, because of a closer relation between terrain and river system.

Characteristic for the above methods are, that the terrain and river systems are precalculated through an expensive process that involves the entire terrain at once. We are trying to avoid this. Also river systems needs proper care when using a LOD algorithm as discussed in section 12.3.

We first looked into using predefined paths for the rivers and a 1D profile to represent a cross section of a river bed. That had some serious problems and we looked for another solution, which is actually quite realistic in how it works.

12.3.1 1D path following profile

An obvious extension to our design method was 1D profiles which could be dragged through the landscape along a predefined path. That way rivers could be carved into the terrain to give it a more realistic appearance. The method is simple, and powerful for many purposes, but rivers are not well suited for it.

The problem is that though a river does form the landscape, as it would with this method, the river also has to abide by some rules. The primary one being that it flows downwards. This can easily enough be implemented by starting it at some height and letting the path go deeper and deeper until sea level is reached. While testing this, we had some undesirable effects. If the rivers path was defined to fall y meters for every x meters traveled horizontally, but the terrain dropped by more than x/y then the river could end up hanging in midair and it was suddenly more an aqueduct than a river.

A simple solution would be to let the river path drop by max(terrain drop, y/x), so if the terrain dropped faster than the profile, then the river would follow along. This works better, but rivers not only flow downwards, they also flow around obstacles, whenever possible. The profile and path method would now carve its way straight through all obstacles without so much as flinching. Sometimes rivers do carve their way through mountains, but not if there is a perfectly good way around it.

The last problem is river basins. With a profile method, they would be missing. The river would always just flow right along like a canal and never widen to fill nearby lower regions.

A river transforms the landscape, but the landscape transforms the river. For that to happen, a feedback loop must be used, and that is certainly not happening with the design method, and it is generally a problem with a procedurally generated terrain, since the terrain does not come into existence before we look at it, and then even only a small part of the total terrain is generated.

12.3.2 Dropping water

This method is not real-time, but rather an extension to design. Rivers are generated procedurally in a way that makes them fit perfectly into the landscape. The resulting rivers are then stored as design and are blended into the terrain when it is generated. The more rivers, and the more detailed they are, the more memory and preprocessing time will be used on them. The problem with the profile method is that it does not follow the landscape, because there is not any landscape to follow, before an observer looks at it. The solution can be to actually generate the landscape and let water run downhill from some starting position. We do not have to define the entire landscape in order for this to work. A position is designated drop point and the triangle containing the point is refined until it reaches a reasonable level of detail. The finer the detail, the more correct a solution is obtained, and the more time it takes.

This method for river calculation is describes in detail in section 13.

12.4 Vegetation

Vegetation is a constituent part of a realistic looking landscape. It is so on different scale and for different reason. When looking at a

terrain from far away one would expect to see plant belt up along the incline of mountain. The absence of vegetation at the top of mountain and in dry areas are also indirectly linked to the role of vegetation. But when moving closer to the surface of the terrain and looking across flat areas, vegetation can break the monotony by adding variation to the scene. Especially so when the vegetation is rendered as billboards or even as 3D models.

Much of the landscapes color is defined by vegetation, as we describe in more detail in section 14, but there we deal only with ”flat” vegetation, which defines ground color, but adds no extra form.

12.5 Erosion

Dunes along the coast line and in deserts. Volcanoes with their once rough edges rounded. River bed winding through the valley. Glaciers moving larger rocks across the country. All examples of erosion of the landscape. One type of erosion is already described in section 12.3.

Erosion is an important factor in making the terrain believable. But also the lack of erosion or only minor erosion is expected in some cases. Like meteor crates on for example the Moon. Meteor crates are highly visible due to the close to non-existing erosion on the Moon.

Simulating erosion can produce quite amazing terrains. Soft, half moon shaped dunes like seen in deserts on the Earth or natural looking river systems. In H. Nishimori and N. Ouchi article,[Nishimori and Ouchi 1993], they are simulating erosion and disposal of sediments. They achieve to produce two types of dunes commonly seen in dessert on the Earth.

Jacob Olsen [Olsen 2004] has come up with a solution to produce eroded terrain realtime. Erosion generally produces heigh altitude rough terrain and low altitude soft terrain. This is due to material being washes, blown or falling down from above to settle in the lower areas. Further more, areas near coastal lines will tend to either have very steep slopes, or be very smooth, depending of whether the ground is consisting of rocks or softer soil.

Erosion can be simulated real-time by smoothing the terrain more or less, depending on the level of erosion. Erosion of design elements (section 11) can be simulated by ajusting the blend factor down towards zero. At zero then design is entirely gone and only the roughed landscape remains.

13 Calculating the natural flow of a river

The ”rule” in ROAM is that any given triangle will only have neighboring triangles which are one level larger, one level smaller or the same level. No two triangles with a level difference of more than one will ever exist. It is easy to see that a neighbor will either share the left, the right or the bottom edge. If the neighbor shares the left or right side edges, then it does so with either one of its side edges or its bottom. If it does so with a side edge, then it is obviously the same size. If it does so with its bottom, then its bottom is as small as the current triangles side, and it is one level smaller. If on the other hand a neighbor shares one of its side edges with the current triangles bottom edge, then the neighbor is one level larger.

This means that if the drop point is located in a very small triangle, then the three neighbors are at most one level larger, and so they are very small and detailed as well. We only ever need the current and the neighboring triangles to be detailed. The world outside this small area is of no relevance, since the water is at the current triangle, and can only flow into the neighbors. The surroundings can be as rough as they want to be, as shown in Figure 64.

We can now look at the current triangles normal vector and directly see which neighbor should receive the water, or even how much water each of the neighbors should receive. We can also just decide on the lowest positioned neighbor and send all the water into it. Looking at it at the fine scale, the normal should be used, but at a slightly larger perspective the lowest neighbor is a good choice. We use lowest neighbor in order to avoid local minima. To let the water flow in a slightly wider stream, one could decide on some randomness in how the receiving neighbor is selected. A weighted randomness, where the lowest is most likely would be easy to implement.

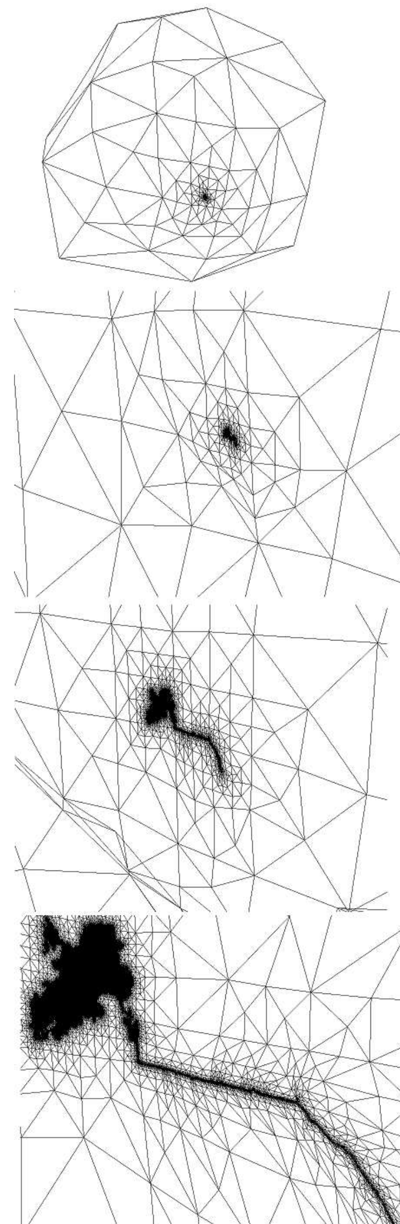


Figure 64: Four levels of zoom into a lonely river on a planets surface. The first image shows how rough the surroundings of a river can be, while the last one shows the flow of a river which ends up filling an irregularly shaped lake

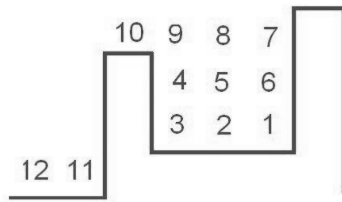


Figure 65: Twelve discrete steps in flowing water

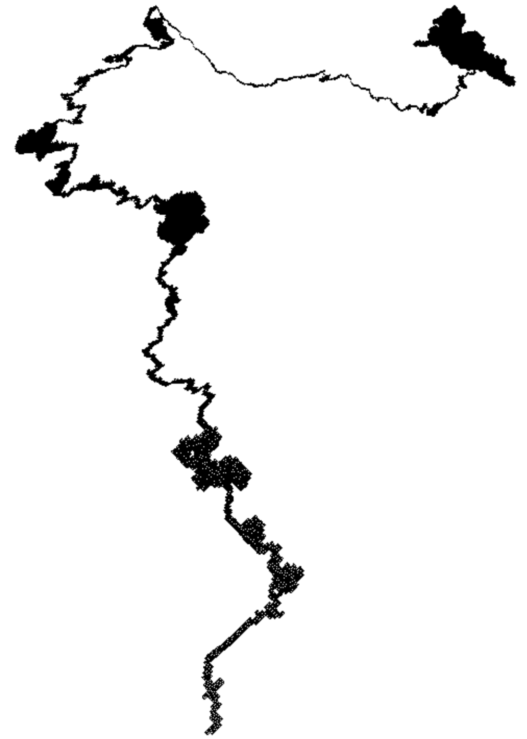


Figure 66: The calculated path of a river in a 3D landscape, showing both the winding flow and river basins filling up

A drop of water can be moved around like this, but it will quickly be caught in a local minimum, and that would be the end of that river. The solution is to always move towards the neighbor with lowest altitude while considering water level as well. A triangles altitude should then be the altitude plus the level of deposited water on top of it. This means that whenever the water moves, it moves as a stream, which means that it drags water with it. There is a front of the water, which is the ”drop” and there is a trailing stream of water at the same water level, or higher.

The method new becomes dropping water on a triangle and raising its water level by one unit. Then we find the lowest neighbor triangle and move to it and raise its water level by one unit. If more than one triangle are equally low then we select one at random. If we arrive at a local minimum, where there is no lower neighbor, then we raise the local water level by one unit, while remaining stationary, and suddenly the previously traveled triangles are once again lower, and we can move back. This has the effect of water filling local minima in the terrain until the lowest point on the edge is reached in which case the water rushes out and moves along its path.

A simple example of water starting in a local minimum is shown on Figure 65. The water is dropped at 1. Its lowest neighbor is to the left. It moved left to three from where there is no lower neighbor (1 and 2 had their water level raised by one), so it raises its own level, and becomes 4. Now 2 is a neighbor which is just as low, and the water moves there in step 5. It moves to the right until it again gets stuck at 6. It raises its level and becomes 7 and can now move left again. This time is does not stop at 9 but keeps moving left into 10 from which it drops down into step 11. From there it flows on. The resulting mesh is shown in Figure 66 where one can see the detailed mesh containing water and the coarse surroundings.

13.1 Moving point of interest

Before choosing where to flow the water next, the three neighbors all need to be at the same subdivision level as the current triangle. If not, then we don't know exactly which neighbor is lowest, since that can change when the triangle is refined. Neighbors are at most one level lower in subdivision level, so it will be quick to ensure the correct level for the neighbors by splitting those which are too large. Remember that we do not generate a new mesh each time the water flows. We merely refines the existing one. When water flows and raises the water level, we need to store this information on a per triangle basis. Any given triangle, which contains water, knows its current water level.

An example of a mesh, which has been optimized as described, to let water flow the correct way, and to form river basins, is shown in Figure 67. The landscape is not colored blue or raised to the water level. What is shown in high detail is the landscape which is under water. In the example, the landscape under water is subdivided into triangles of around 3 by $1\frac{1}{2}$ meters in size. It is apparent that the landscape outside the water filled region can be as rough it want to. It does not affect the accuracy of the calculation. What is shown is a river flowing into view from the right hand side in the top of the image. From there it flows down and left to a point where it reaches a local minimum and fills a lake. At some point the lake overflows near the center of the image and starts filling up a neighboring minimum. In this example the flow was stopped when a triangle count of 100 000 was reached.

A river forming a string of interconnected lakes can be seen in Figure 68, while Figure 69 shows two lakes created and connected by a single river.

The tracing of a river stops when it meets the ocean as seen in Figure 123.

13.2 From 3D river to 2D design

As described previously, a river can be calculated by optimizing a mesh and flowing water. A mesh defining a river is not a form which is easy to store for further reference, and it is very much unlike the design methods we are using.

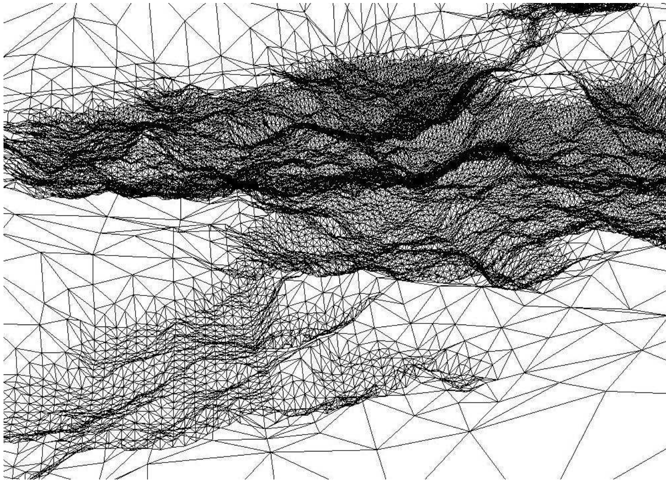


Figure 67: A mesh optimized by flow calculation for a river

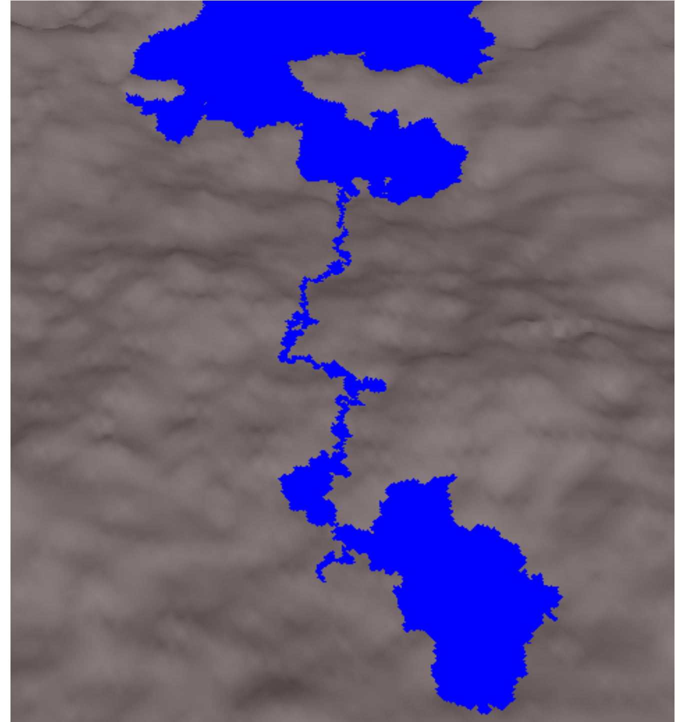


Figure 69: Two lakes are created by and connected through a single river

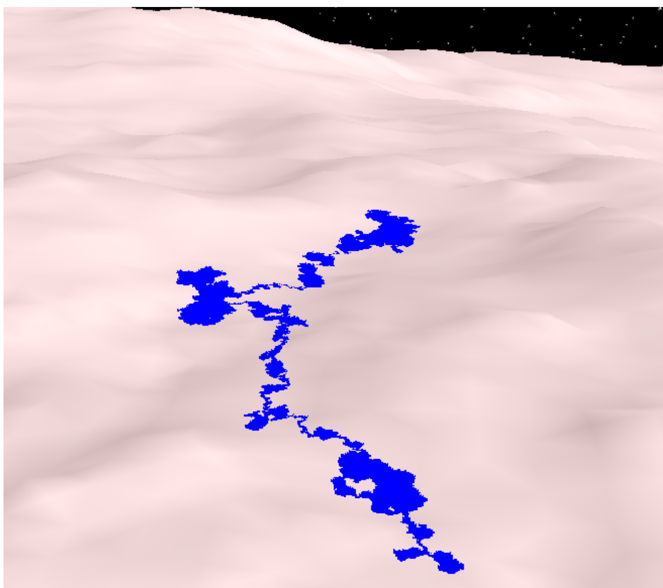


Figure 68: A river forms a system of interconnected lakes

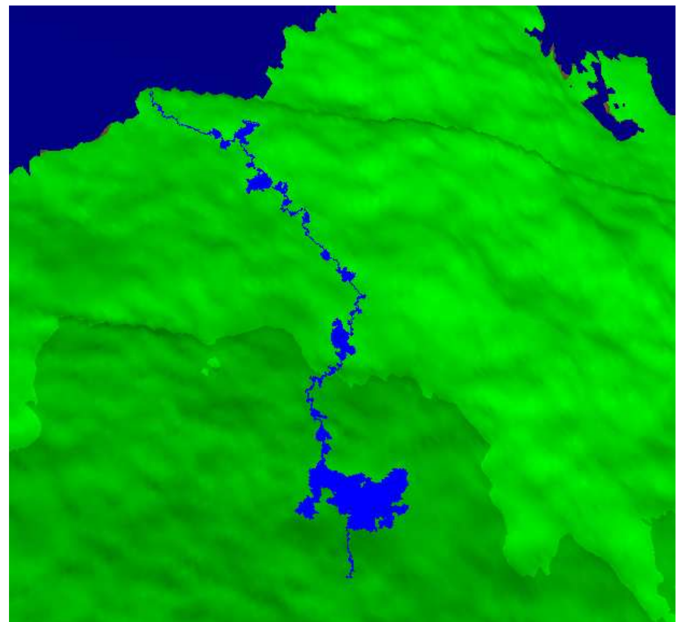


Figure 70: A river flows through the landscape, forms a single lake and then moves on till it reaches the ocean

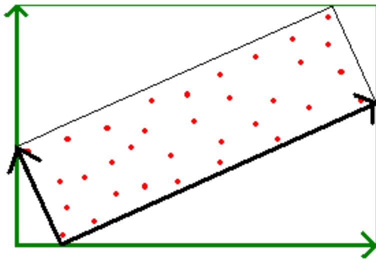


Figure 71: Two different bounding areas for the same collection of points

What we need is to transform the 3D river mesh into a 2D design. This is easily done, since this is in essence what we do when sampling a design for a match section 11.3.1, given a 3D position. We then iterate though all "below-water" triangles and project them into design space where they can be stored in an array as water level values.

A design space needs an orientation, which is not given. In general we can decide on a y-axis which is perpendicular to the spherical defining the planet. In other words, the y-axis should point away from the center of the planet. The x- and z-axis can be any two vectors which are normal to each other and the y-axis. Some are more optimal than others.

As an example. Consider a river flowing in a straight line with the length L. If we define x and z so that the river flows along x=z in design space, then the array used for storage will need to have a width and height of $\sqrt{L^2/2}$. If on the other hand the axis were selected so that the river flows along x, then the array would be L wide and 1 heigh, which is obviously more effective. The only situation in which the choice makes no difference is when the river triangles are spread evenly in all directions, which is rarely the case,

A rivers optimal design space We find the optimal coordinate system through principal component analysis of the river triangles. By aligning our design space axis with the principal axis, we obtain an optimal design space representation. An example of a group of positions and their resulting principal axis, as well as the unit sphere in the principal space, is shown in Figure 72. It should be evident that the points can more compactly be represented by the principal coordinate system, as shown in Figure 71 where the same collection of points are contained in two different coordinate systems. The system, with its axis aligned to the principal axis of the points, can represent the points more compact.

We decide to define the x-axis as the larger of the principal axis, and the z-axis as the smaller. We perform this analysis 2D space, by first transforming the 3D position into an arbitrary design space and setting the y-values to be zero. This is done to make the calculation of the covariance matrix and the eigen vectors and values easier by dropping a dimension. When we have performed the principal component analysis in this design space, we rotate the axis to align with the principal axis.

13.3 Lakes

Lakes can be created by rivers filling larger local minima, and they can be at any altitude. If a river is limited in how many steps it can make, then it might end its journey in a lake, which will then only have an inlet. If the river is not limited in number of steps, then it will fill up the lake until it overflows and then flow on from there. This will give a lake with both an inlet and en outlet.

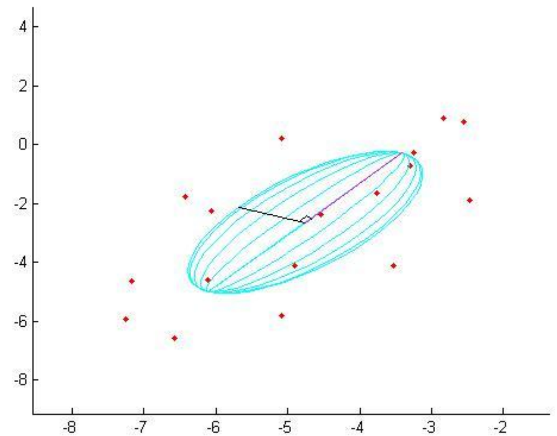


Figure 72: The principal axis, and unit sphere, for a group of 3D positions

Figure 73 shows a lake with no apparent inlet or outlet. The source of the river is at the bottom of the lake, or can be though of as being many very small streams, which are not themselves visible.

Figure 74 shows a lake with both an inlet and an outlet. It is formed by a river flowing through the landscape, and getting stuck at a local minimum until a lake is formed and the water level is raised enough for the river to flow out through the lowest point of the local minimum.

13.4 Erosion

The model can be extended in a way so that not only does water flow and deposit more water - it also erodes the terrain while flowing. The moving drop of water could absorb and deposit soil when moving across the landscape. The absorption and deposition would be a function of the gradient of the current triangle. When water rushes down a steep slope, it erodes and does not deposit. When it oozes along over flat terrain, it will not erode but deposit. This will have the effect of rivers carving deep into steep slopes and flattening out in the lowlands. If implementing erosion, then not only the water level should be recorded, but also the resulting ground level should be stored to be used when rendering the terrain.

13.5 Dry inseas and filling the ocean

Just as we can raise the ocean to become rivers and lakes, we could lower it, in order to get regions of dry land below sea level, which are cut off from the ocean. We can randomly sample position on the globe, and if its altitude is below sea level, then we can flow a river in it and fill it up until we are above sea level. There needs to be an upper limit for this river, because we could very well have started at a position which is actually in the ocean, and then we will really just be filling the ocean, which could take some time. If we rise above sea level before we run out of water, then we have a dry insea. The river is not actually used to provide water this time. It is just to sample the area to ensure that we are not connected to the

ocean. This method will generally take a long time, and will often fail to find a low area not connected to the ocean. The question is also how much we would miss those low areas if they are just all filled with the ocean. If the MESH resolution was quite coarse, then we could actually use this method to generate the water. Rivers would actually flow and fill the ocean on an inertially dry planet.

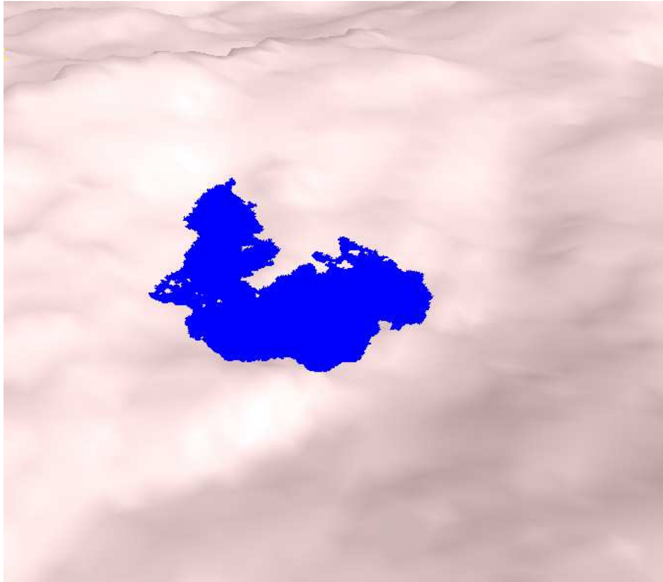


Figure 73: A lake with no visible inlet or outlet

13.6 Visualization of rivers and lakes

A precalculated river is a collection of water levels. It has to be visualized in some way to indicate the existence of a river at the specific terrain location.

Raising the ocean An option would be to raise the ocean at that position. Rather than moving the landscape, to show the river, which is unrealistic, we would raise the ocean. The ocean would need to be highly detailed at that location, to avoid raising water into neighboring land triangles, where there is no river.

Coloring the pixels While it would give a very rough end result to color vertices blue, whenever they intersect with the river, one could instead color the pixels blue when they were at the location of a river. One could in theory convert the river into a real texture and project it onto the terrain, but a simpler method would be to use the pixel shader more directly. Whenever a vertex is generated at a location which intersects with the design spacesection 51 of a river, the vertex gets projected into design space and has the resulting position stored in the vertex itself. When the vertices are later rendered, they contain design space coordinates and the pixels in between two vertices will get a linearly interpolated design space coordinate (just as if it was texture coordinates for the vertices). When the pixel shader outputs a color it samples into the rivers design space and outputs blue of the samples water level is above zero.

A few problems exist with the pixel coloring in the pixel shader. If a landscape has many rivers then a vertex could be inside more than one rivers design space and it would then need to store more design space coordinates (like multiple texture coordinates). While we can easily provide a vertex with storage for more than one design space coordinate, we need to define some upper limit to the number of coordinates. If this upper limit needs to be very high then it is a problem. A more general problem is that this method would not show the real water level - but only a color - and it would not allow the observer to dive into the river, or lake, and see the bottom. It would however allow rivers to be seen from far away as thin blue lines, which the raised ocean would not, unless the subdivision level near the river is very high.

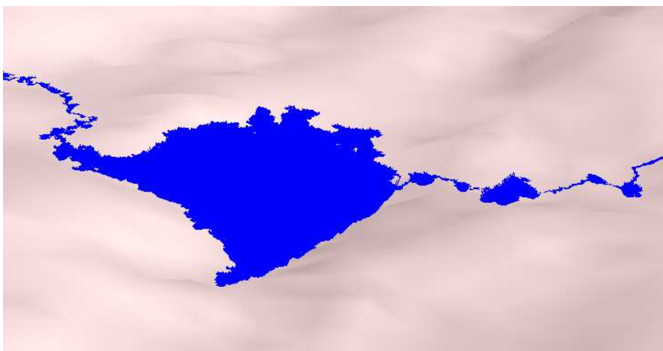


Figure 74: A lake with both an inlet and an outlet➠

13.7 The river conclusion

This method will flow in a natural way in the landscape. It flows downhill and fills up river basins in the local minima. The flow can keep on until the sea is reached, or until a predefined number of steps has been taken, in order to simulate water being absorbed into the ground or evaporating. An interesting feature is that if a two rivers meet, then the combined water level will rise, since the second river will be depositing water on top of the first one in order to move around. Remember that when the drop of water moves, it always deposits water. Figure 66 shows a quite realistic river in a 3D landscape. Only the river itself is shown, or rather the river bottom is shown. Near the top there is a sharp turn which is

caused by the fact that the river flows in 3D and at the position it starts falling down a very steep mountain side. The river ends in a large lake. Figure 64 shows both the final river flow as well as the roughness of its surroundings. While the river itself used 50 000 triangles, while the rest of the landscape used only slightly more than 500 triangles.

A feature not shown here is rivers merging and splitting. Merges happens when two rivers end up in the same channel, while splitting generally only happens at river deltas. Splitting could be added by letting the river flow in a more or less random direction whenever the gradient is below a certain threshold. This will require the ”drop of water” to split into several drops, or it will take several iterations with restarts to form the entire river delta.

14 Visualization

Having the mesh stored as a connected group of vertices does nothing for the experience of watching a terrain. The mesh obviously needs to be rendered.

The ROAM is already a collection of triangles, so it is easily rendered, but a few extra things need attention before it looks good. These things are mainly surface color and lighting. A mesh consisting of tiny triangles with nice detailed surfaces and properly lit will go long way for the experience of watching an actual terrain.

Another thing which most planets need is an ocean and an atmosphere. We will not go into great detail about how to implement those to make it look realistic, but refer the reader to other texts dealing with the subject.

What we do want to point out however, is the way we feel one best deals with ground, ocean and atmosphere. Each can be a separate ROAM with its own split and merge priorities and its own color scheme. This means that the ground is actually one whole uninterrupted mesh, which just happens to be below the ocean surface at some locations. Then one can color the ground differently when it is over water from when it is below water, and it means that one can easily fly under water and explore there as well as seen on Figure 75. Letting the ocean ROAM become more detailed near the observer also allows for details such as waves to be rendered in great detail. Having a sperate mesh representing the atmosphere allows for nice effects such as partial transparency when the sun rices and sets as seen in Figure 76 and in Figure 76, and the entirely different look when one is above the atmosphere as seen in Figure 78. Clouds could be generated by coloring the atmosphere white and letting a noise function define the alpha component as seen in Figure 79. It would not be complicated to select a color from dark gray to clear white based on the noise function as well. That way the cloud cover could vary from heavy dark clouds to thin white clouds.

14.1 Lighting

In order to calculate correct lighting we need the normal vectors four our mesh. We calculate the normals as an area weighted average of the normals of the triangles surrounding the vertex. This is done to ensure that large triangles have more influence on the resulting vertex normal. There are other methods, but that is a subject on its own. Figure 80 show an example where a vertex's normal is calculated as the average of four triangle normals. All four triangles are equal in size, so the normal is just $N = (N0+N1+N2+N3)*\frac{1}{4}$.

A thing one should keep in mind is that there actually is no normal at a vertex, unless all surrounding triangles are in the same plane. The triangles do not form a continuous surface. At the vertices the first derivative is undefined, and that is what defines the normal. Mathematically speaking the normal is undefined at the vertices, so which ever normal we decide on can be equally correct. It all comes down to what we think looks best, and what seems right when we consider the underlying continuous surface which we are trying to approximate with the mesh.

Figure 75: The ground a muddy color under water. The light is also made more ambient to simulate the scattering effect of the water and particles therein

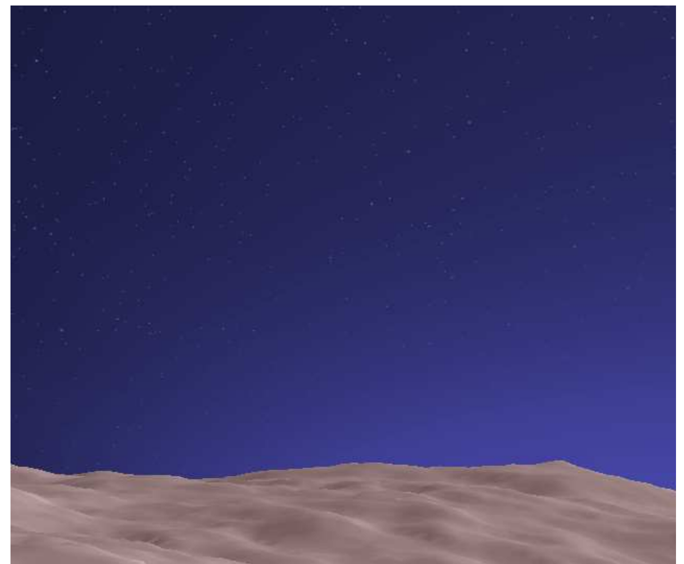


Figure 76: Looking towards the day break. The sky is starting to become more bright and blue in the lower right corner, but the stars are still slightly visible, more so when looking away from the light

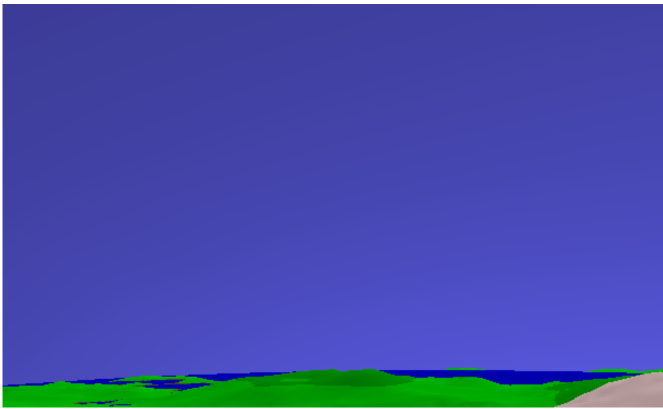


Figure 77: The sun is back on the sky, and we can no longer see the stars. Only the clear blue sky is visible

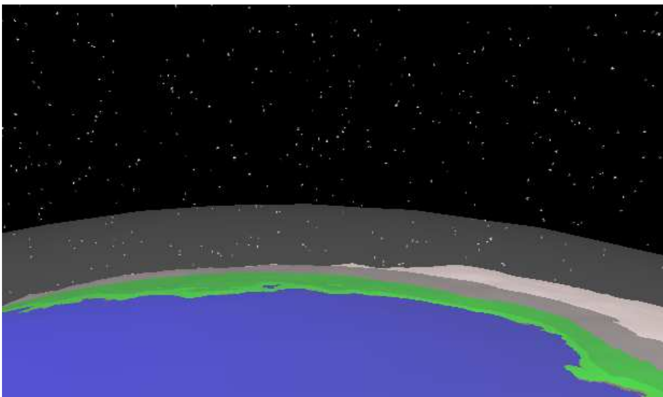


Figure 78: From a position high above the atmosphere, it looks like a semi transparent layer of air. This is not very realistic, but it shows that the atmosphere can easily look very different depending on the observers position

14.1.1 Enter the shade

We start this section with a very brief introduction to the three most common shading methods. It is in no way meant to be a thorough explanation. We just want to bring the terms into focus before dealing with the details relating to ROAM-generated landscaped.

The most common ways of shading a mesh is faceted, Gouraud and Phong. All calculate the diffuse lighting at a given location as the dot product between the normal vector and the vector pointing towards the light. This means that a face, which normal points directly at the light, receives full light intensity, while a face, which normal is at a right angle to the light, received no light. What they do with this light intensity next is what separate them.

- Faceted - A triangle has one normal vector for its entire surface, which gives it the same light intensity all over. This is not the most common method, but in a sense it is the most correct. Faceted, or flat shading as it is also called, actually lights the triangle mesh as it is supposed to be lighted. The triangles are shaded as if they were flat, which they are.

- Gouraud - Given normal vectors at the vertices, the light at the vertices are calculated and then that light is interpolated in between the vertices to shade the entire surface. This is a quick method to give some smoothness to the shading. It is however an approximation. The method attempts to shade the collection of flat triangles, as if they were more smooth. They *are* flat, but generally the grid of flat triangles is trying to represent a continuous surface which is not faceted. A problem with Gourauds shading is that no position on the face of the triangle can be brighter than the corners, since the light intensity on the face is an interpolated value obtained from the corners.

- Phong - Besides the ability to provide highlight, Phong separates itself from Gouraud by interpolating the vertex normals across the surface, and not interpolating the light. For every pixel, the current normal is used to calculate light intensity. This is a much more accurate method, since the light is in a sense sampled more often. rather than sampling the light at three corners of a triangle, Phong samples for every pixel.

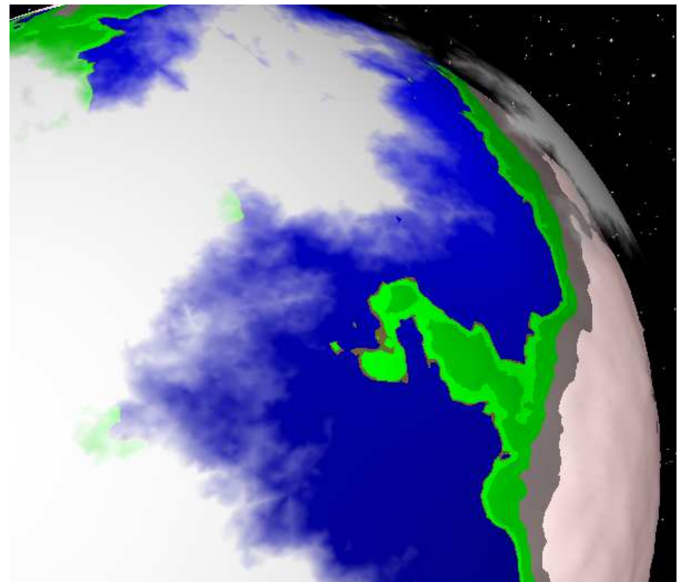


Figure 79: The atmosphere is colored white, but the alpha component is defined by a noise function to emulate cloud cover

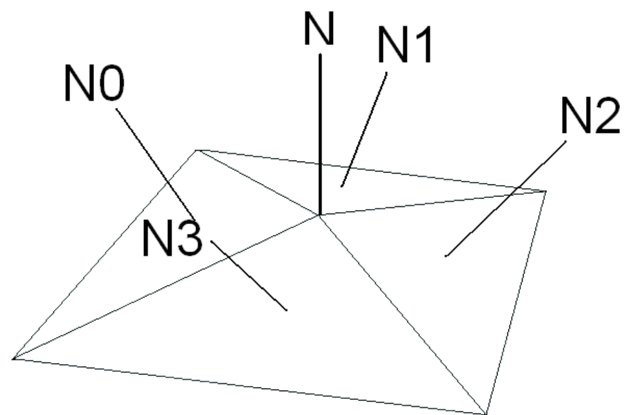


Figure 80: Vertex normal calculated as average of surrounding triangle normals

> This is smoother, it allows for the brightest spot to be somewhere on the face of the triangle, and not just on the corners. Phong also assumes that we want to represent a smooth and continuous surface with our faceted mesh of triangles.

We have claimed that faceted shading is actually the correct shading of our mesh, and we stand by that. However, we need the mesh to be so fine that every triangle is no larger than one pixel in order for this to represent our terrain accurately. Since that is generally not an option, we have to resort to smooth shading. Using Phongs shading model should let us have a somewhat rough triangle mesh and still show it as if it is more finely detailed and continuous. That it does, but it is not without problems.

Phong vs. ROAM Using ROAM and Phong naively soon generates nasty artifacts as seen in Figure 81. The terrain is filled with odd looking squares and triangles, which are clearly visible. We have named the artifacts ”pyramids” because of their shape. They all consist of four triangles meeting in a common center vertex. All triangles have the same size and they are all generated the same time by the same split operation in ROAM. The center vertex is offset from the four corners and this forms a more or less regular pyramid.

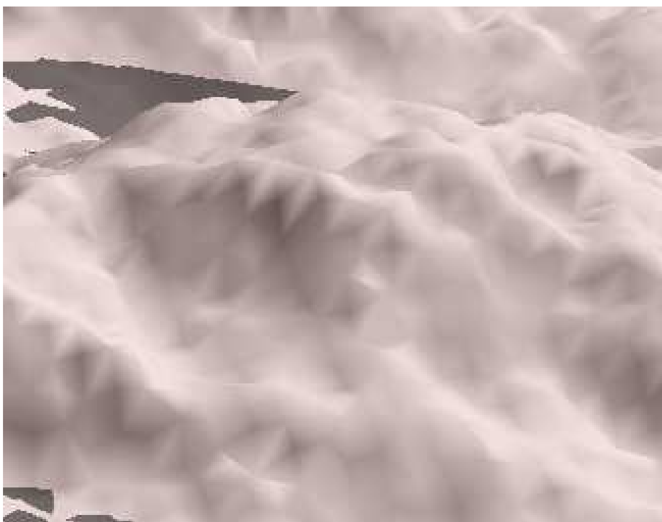


Figure 81: Artifacts from Phong shading a ROAM generated mesh

At first this looks like an error in the normal calculations or in the shader itself. It is however not an error. It is how Phong lights that shape. Though a smooth interpolation is used by Phongs shading model to generate normal vectors for every pixel on the surface, it is still a simple linear interpolation, which has its problems at times. See Figure 82 to see a demonstration of the problem. A pyramid is drawn as seen exactly from the top. The light is coming from above and the four sides each point 45 degrees away from vertical and out to one of the four sides. The vertex normal at the top is therefore pointing straight up. The four vertex normals at the base are all defined to be horizontal, as if they are connected to other triangles (out of view) which have normals pointing downwards. The normals are therefore interpolated from $|0,1,0|^T$ to $|+/-0.71,0,+/-0.71|^T$. This is done linearly, so half way down the sides, there will be a square line going around the pyramid, where the normals are all 45 degrees away from vertical. At this line the light intensity of the pixels will all be $cos(45) = 0.71$, and all pixels on the line will have the same appearance. Going one pixel up or down will change the light, but again all pixels at the same level will look alike. This way the top will be bright and the base will be dark and we get Figure 82, which is not smooth looking - even though we do a smooth interpolation of the normal.

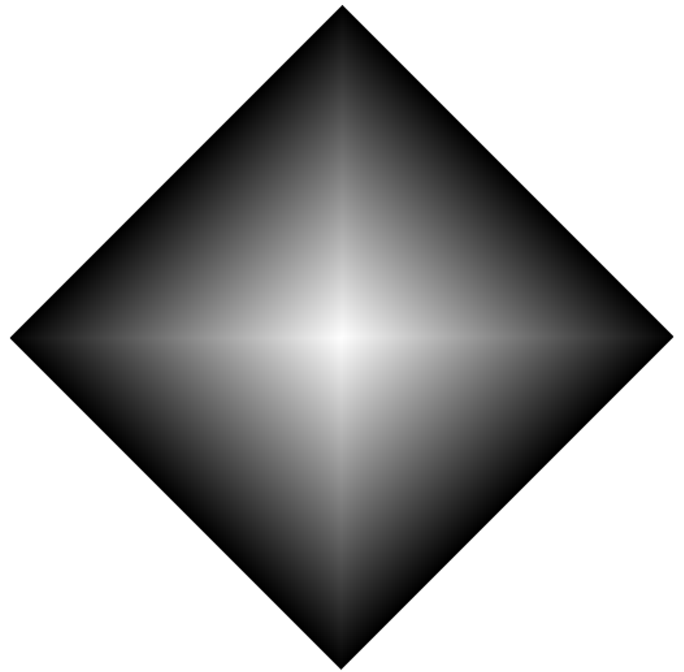


Figure 82: Demonstrating how Phong lighting a pyramid will generate artifacts which are in no way looking smooth

The reason this paragraph has the title ”Phong vs. ROAM” is that even though the pyramid artifacts exists on every mesh, it is particularly frequent in a mesh generated by ROAM. This is because of the way triangles are subdivided. Whenever two triangles are split into four, a pyramid shape is generated, and that shape is more prone to visible errors, than for example 8 triangles, meeting on a single vertex, are. The more triangles we have around a vertex, the more the pyramid will look like a circle, and the less prominent then bright edges between the triangles will be. On Figure 82 we have four distinct lines of ”brighter” pixels, which would be less visible if we had more of them, since it would become more smooth that way.

Being brighter than most An interesting fact is that the brighter lines are not really brighter. If you draw a square parallel to the sides and centered around the top, then it has the same brightness all the way. What we see as bright lines is an artifact known as Mach bands[Watt 1992], which is also quite prominent in Gouraud shading. In [Olano and Yoo 1993] the artifacts of Phong shading, and a solution to them, are described in detail. It is not so much the color changing as it is the change in color changing. The human eye is very sensitive to changes in the change of a color, or put another way; while we may not see the color change it self, we are sensitive to changes in the colors first derivative. If you try to get a clear look at the lines in Figure 82, then it is not that easy. The more you focus on it, the more blurred the lines become. If you zoom into the line, then it almost disappears. That does us little good however. Sometimes graphics is easy because we can cheat the eye and just make something that looks right. Other times, like now, it’s hard because the eye is being unreasonable.

Avoiding the pyramids pyramids are very visible, and they look wrong. ROAM makes loads of pyramids. It sounds like a problem, and that is exactly what it is. There are two somewhat easy solutions to the problem.

- Split the pyramids - If we always make it a priority to split pyramids, then we can make them go away. We only have so many triangles to do with, and whenever we split something, we make new smaller pyramids, so we can't simply split away. What we can do is to make splitting pyramids a somewhat higher priority than splitting triangles which are not in pyramids. We can also define pyramids with very different vertex normals as being more of a problem and prioritize those even higher.
- Smooth the light - The reason we see the pyramids so clearly is that the five vertices (top and four at the base) are not evenly lit. If they all receive the same light, then a pointy pyramid will look flat, and in essence not be there any more. We can not just make pyramids look flat, since this will make it impossible to see how the landscape is formed. So if we can somehow make the light more even for all the vertices, then the problem will go away. In lighting we have direct and indirect light. The dot product of the normal and light direction only deals with direct lighting and it strongly connected to the normal. Indirect lighting, considered to come from everywhere, would be the same for all vertices, no matter what their normal is. The quick hack to provide indirect lighting is to create ambient light which lights up any surface equally no matter how it is oriented. If we do this on a planet, then we get light in the middle of the night, which is generally not something we want. An alternative to ordinary ambient light is to calculate the light which would hit a horizontal surface at that position on the planet, and then use that as ambient light. It will only give light during the day, and it will give most light at midday. At the same time, this light will be almost exactly the same intensity for all vertices (unless the triangle is huge, in which case it will probably be subdivided anyway). This can easily be implemented by taking a vertex normal to be the average of the neighboring triangles and the vector representing vertical. We propose weighting the old style normal and vertical 50/50, so for a pyramid top we have $N = \frac{1}{8}(N0+N1+N2+N3)+\frac{1}{2}Vertical$. A pyramid infested landscape and its ”vertical normal”-fixed counterpart can be seen on Figure 83 and Figure 84. The pyramids have not gone away, but they are far less intrusive.

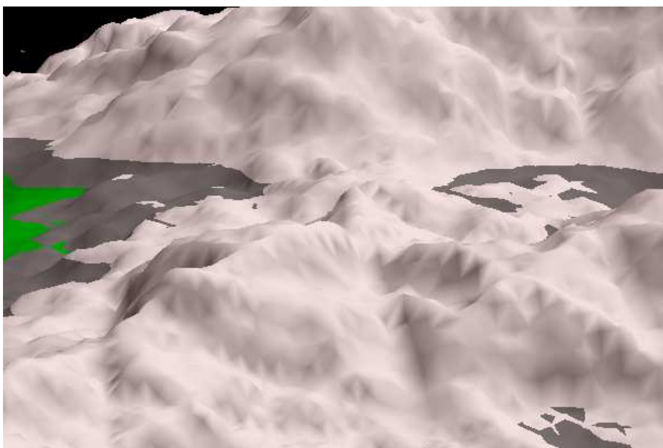


Figure 83: Landscape with plenty of pyramids and no attempt to hide them

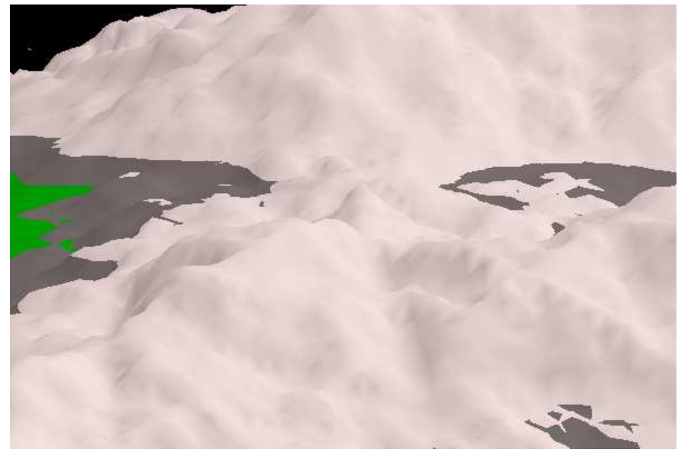


Figure 84: Landscape with plenty of pyramids which are made less apparent by blending the vertex normals with a vertical normal vector to simulate ambient light which changes across the face of the spherical planet

Light on a sphere The usual lighting scheme is all about checking if the normal vector points towards the light and how much. Lighting is never that simple. Objects cast shadows on themselves which require some extra thought to be put into the lighting. It would not be terribly difficult to implement volume shadows for the optimized mesh, and get that part of lighting into the scene as well, but we opt for an easier solution. For a planet it is not difficult to check if a position is in the general shadow of the entire planet. The general shadow is the one cast by the spherical base shape of the planet, and for a directional light, as the sun, this forms a cylinder of shadow behind the planet.

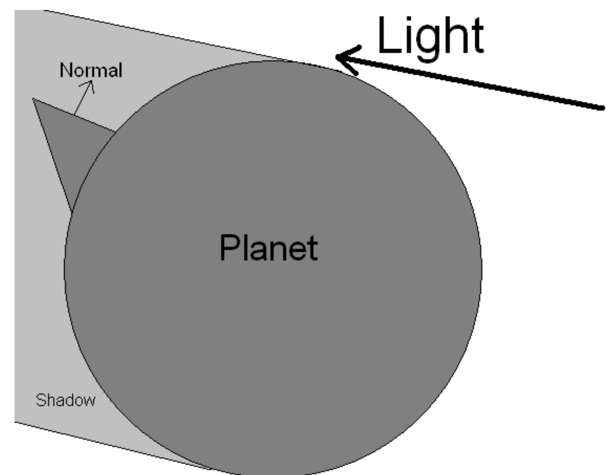


Figure 85: A surface can be in darkness even though it is facing the light

14.2 Terrain color

Having a bumpy landscape, with nothing but uniformly colored surfaces, will not do much for the feeling of ”being there” at another world. In order for the experience to become more intriguing, we

need at the very least some colors, and the colors should have fine detail to make the surface look interesting.

A very common method to color procedurally generated terrains is to use a 1D texture where the terrain altitude is used as a texture coordinate. This lets the terrain be white on mountain peaks, gray on rocky surfaces and green in the lowland. This can look good enough at times, when the texture has many different colors with smooth transitions. It does however generate a ”banding” effect where horizontal cross sections of the terrain all has the same color. On Figure 86 this is shown on a terrain with only 4 colors. The sky and the water is bluish and should appear separated from the terrain, as they do, but the terrain itself does not look natural at all. In the real world many other factors than the altitude play a role in defining the color. Figure 87 shows another example while Figure 88 shows the contour lines for the same landscape. It is evident that the colors are following the contour lines exactly.

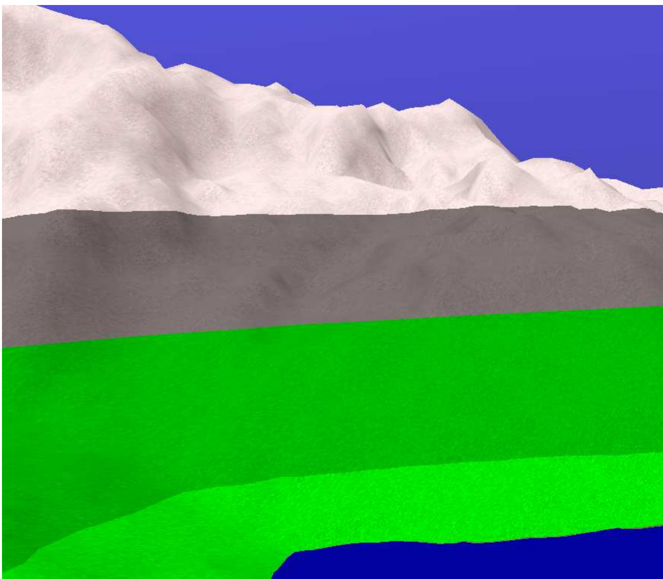


Figure 86: Terrain colored based only on its altitude

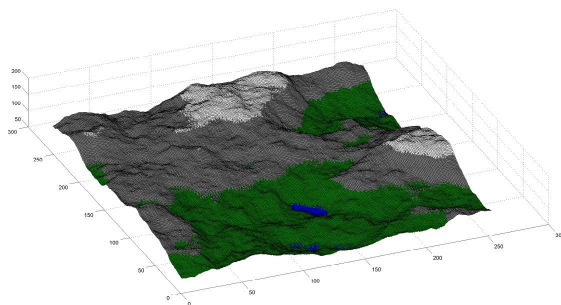


Figure 87: A terrain is colored according to its altitude. The colors are noticeably arranged in bands where the border between two colors are at one specific altitude.

Adding something random In nature, does the snow line or tree line on mountains form a perfectly horizontal line? The answer is of course that it doesn't. That is really the problem. The solution could be to base the color of the landscape on more than the altitude. The altitude and some noise function could let the snow line and tree line move up and down, within certain bounds, as they move across the terrain.

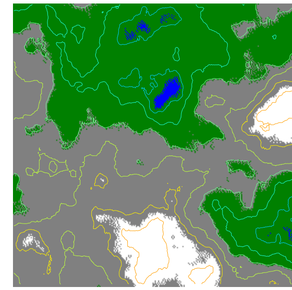


Figure 88: A terrain is colored according to its altitude and it is clear that the borders between colors follow the contour lines.

If we perturb the altitude with the noise, before inputting it to the color-by-altitude function, then we will easily get this effect.

On Figure 89 a terrain is shown, where the color is based on altitude as well as a random perturbation. The terrain clearly looks more real now that the snowy areas are not cut off at a certain altitude, but sometimes stretch downwards and sometimes stretch up. The contour lines are again shown on the landscape in Figure 90 and now it is clear that though the terrain color is related to the altitude, it has a random element as well.

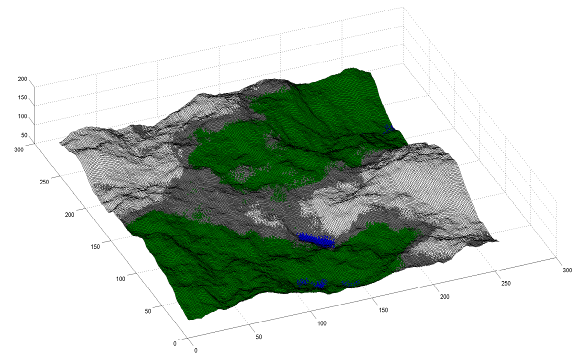


Figure 89: A terrain is colored according to its altitude and a noise offset which perturbs the color selection, so the border between colors no longer occur at specific altitudes, though snow is still dominant at high altitudes, water is below a certain altitude, and grass is dominant at the lower areas.

Two more examples of color by altitude with and without noise are shown in Figure 91 and Figure 92 where it again is evident that noise adds something extra which makes the image a little less synthetic looking. Some parts of the terrain, which are high enough to be defined as rocks, are now grassy, while other parts in the lowland now have small rock formations.

The noise can be defined in different ways. Figure 93 and Figure 94 show the same terrain but this time with slightly more high

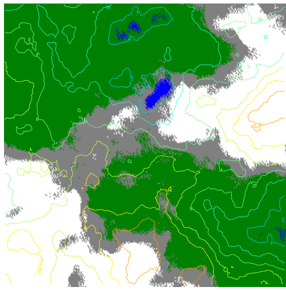


Figure 90: A terrain is colored according to its altitude and an extra noise function which perturb the altitudes. It is now clear that the borders between colors do not entirely follow the contour lines.

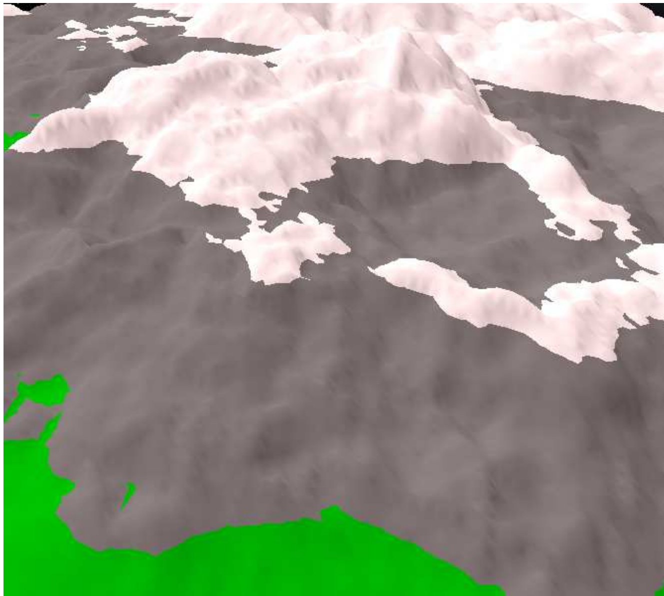


Figure 91: A terrain is colored according to its altitude

frequency noise. Which frequency and amplitude looks better is dependent on the use.

Other parameters for selecting a terrain color/type As well as using the altitude to select a terrain type, one can decide to use the gradient of the terrain, the facing of the terrain, the latitude on the planet, procedurally generated weather patterns and so on and on.

A few general observations, which can easily be integrated into a terrain engine are that steeply sloped terrain is less likely to carry snow, hight altitude terrain is more likely to be snowy, terrain near the poles are more likely to have snow, while terrain near the equator is more likely to be hot and dry. Mountain and hill sides facing north, on the northern hemisphere, is likely to be a little colder then sides facing south. On the southern hemisphere this is reversed. In the end the terrain type, and resulting color, is easily given though a few simple probability functions.

Figure 95 shows an example of a terrain where lighter green patches

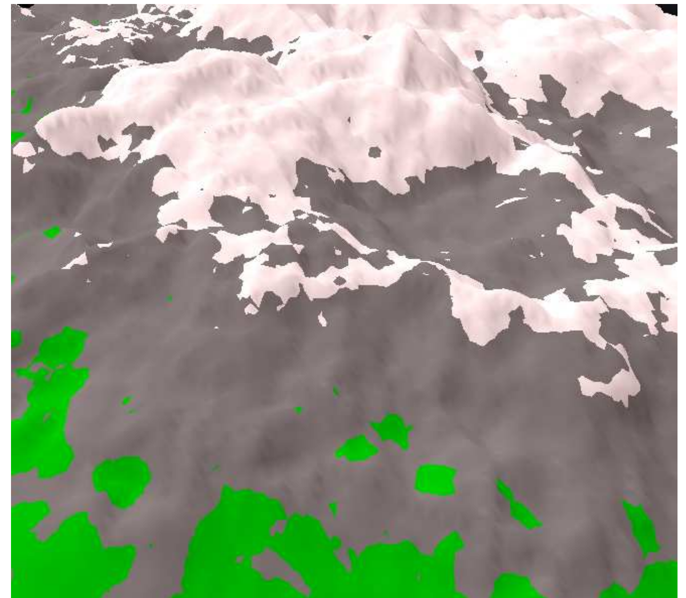


Figure 92: A terrain is colored according to its altitude where the altitude is perturbed by a noise function before a color is selected. The perturbation is low frequency and with medium amplitude

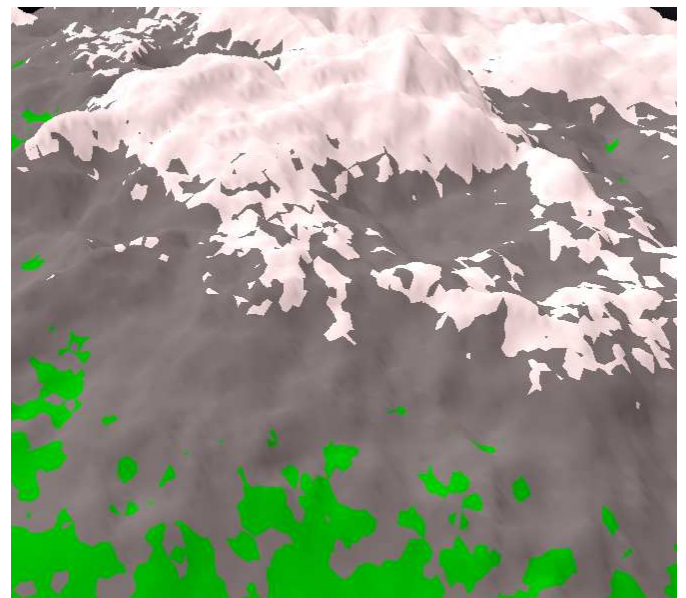


Figure 93: A terrain is colored according to its altitude where the altitude is perturbed by a noise function before a color is selected. The perturbation is medium frequency and with medium amplitude

of grass are spread on the darker green based on randomness where the probability depends on the facing of the terrain. Terrain facing "left" is more likely to be light green, but not all terrain facing "left" is green.

Another basic observation, which holds true for any planet which receives heat from only one Sun, and which does not have an atmosphere which can provide absolutely uniform temperature across the planet and which does not have a rotational axis which is tilted 90 degrees relative to its orbital plane, is that the poles are colder

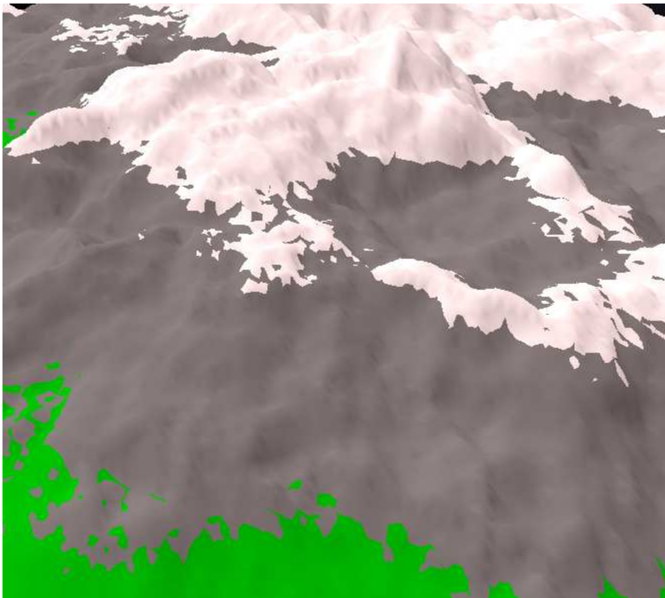


Figure 94: A terrain is colored according to its altitude where the altitude is perturbed by a noise function before a color is selected. The perturbation is high frequency and with low amplitude

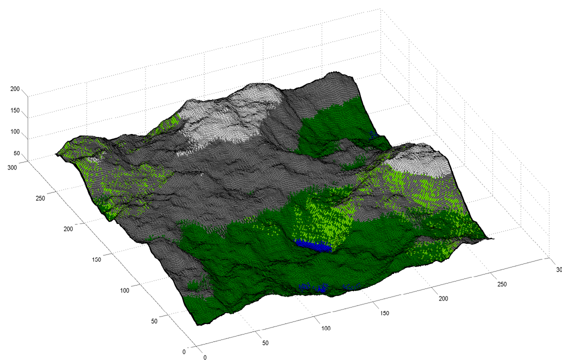


Figure 95: A terrain is colored according to altitude and its gradient along with another noise function. On slopes facing "left" there will sometimes be patches of light green grass, when the noise function also allows it.

than the equator. This is most planets, if not all.

This can be simulated for an Earth like planet by generating "ice" in the polar regions. An example was generated where everything north of the 80th parallel is colored white. The result is shown in Figure 96. While it *is* a polar cap, it looks quite silly. The exact same thing is then done while perturbing the latitude with a noise function before deciding on a pixel color. The result is shown in Figure 97. This time it looks very much more real. The ice stretches south in some areas while the blue ocean at other points go up into the high north. Even a few small icebergs have become visible.

Using the pixelshader This method can be implemented on a per vertex basis, which will often look good enough, but when you need higher detail than the mesh can provide, it needs to run in the pixel shader, which is what we have done in the demo from which the pictures are taken.

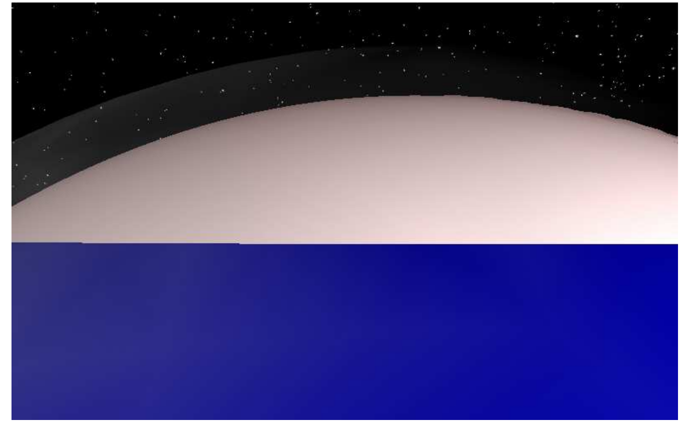


Figure 96: An ocean is colored according to its latitude and only the latitude. Over a certain latitude, the ocean is frozen, and rendered as white ice

Figure 97: An ocean is colored according to its latitude and a noise function. Over a certain latitude plus minus noise, the ocean is frozen, and rendered as white ice

14.2.1 Pixel shader to color the landscape

Rather than using textures, one can use the pixel shader to color the landscape on a per pixel basis. If the above mentioned random offsets, as well as terrain properties such as 3D position and normal, are fed into a pixel shader, then it can be programmed to select the appropriate color.

This can be thought of as a way to procedurally generate the relevant textures real-time, but it is more than that. When blending textures directly, the textures blend factor will be defined on a per vertex basis. If we have an edge from v0 to v1 and v0 is grass and v1 is rock, then the grass texture will blend smoothly over into the rock texture along the edge. In the point right in the middle, the textures will each contribute with 50%. This will not look like what is seen in nature. The central parts will be smudged and look unrealistic.

14.2.2 Textures to color the landscape

On a small scale landscape a detailed and artistic texture could be draped on top of the landscape. An orthogonal (or near orthogonal) projected photograph of a real landscape could be used as a texture, and it would look great. There is a problem in the size though. For an entire planet this would be one huge texture, and on a procedurally generated planet the texture would have to be generated procedurally as well. As a means to color the entire landscape, a texture is not a usable method.

Tileable textures Textures can however be used to generate detail. Grass is not just plain green and rocks are not just plain gray. If the solid colors are replaced with textures which look somewhat like grass and rocks, then the terrain would come much more to life. This is another common method, and it works well in most cases. Based on altitude you sample from different textures. this is not without problems though. Even large textures, which are designed to tile seamlessly in horizontal and vertical direction, produces repetitive patterns when used on the large scale. Imagine a texture which consists of random noise but with a slightly darker center. That texture will tile very well, but when this texture in a grid of 100 by 100 tiles, then you will see a pattern of equally spaced dark spots, which is not what we want. This adds an extra requirement for detail texture, which is that it should either be noisy or have only one solid color. Any strong change in color over the surface of the texture should be avoided. On Figure 98 an extreme example of this is shown. The texture used does tile quite well, but it is much darker in the center, and this is a problem. The same terrain is shown in Figure 99. This time a more homogeneous tileable texture is used, and here it looks just fine.

Figure 98: Even textures which tile seamlessly generates patterns when used on a large scale

Figure 99: Tileable textures can be used if they are very homogeneous

Wrapping a texture As any cartographer knows, there is no way to flatten out the surface of a 3D sphere to get a 2D surface without distortion or cuts. This is just as big a problem when doing it the other way around, as is the case when using a 2D texture on a 3D sphere. Generally the texture will have to be pre-distorted so that when it is put on the sphere, and is distorted, it returns to its correct look. That is a clever little trick, but it doesn't really work with tileable textures if they are supposed to cover the sphere by tiling. The distortion will not be the same for every texture, but will vary depending on the latitude on which it is placed. One could use slim and very high textures which only tile around the longitudes but stretch from pole to pole, and this could be pre-distorted and tiled. That will work, but the textures will have to be insanely tall for them to stretch from pole to pole with sufficient detail.

If one naively uses an ordinary texture and define polar coordinates for the sphere, then pinching is the result, which means that all the pixels in the texture in the top and bottom row will meet in a single point at the poles.

Dealing with an initial cube, as we are with our choice of ROAM base mesh, we could try and select texture coordinates for our cube as shown in Figure 101, but even that would not solve our problem, since we will get singular U-texture coordinates for the left and right side, resulting in lines as seen in Figure 100.

We really need an extra set of coordinates for each vertex. Then we could use both and every face could have both lines and dots, which could give a nice effect.

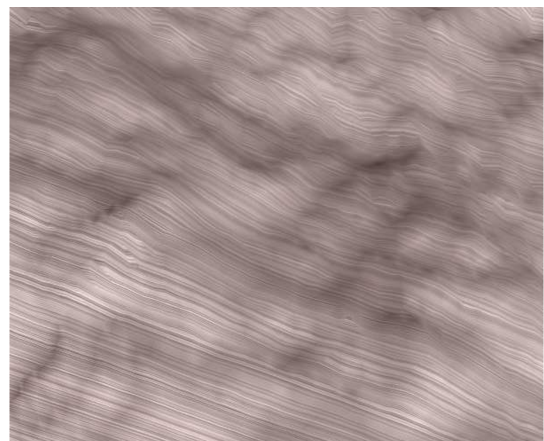


Figure 100: A line artifact from singular U-texture coordinates and a noise texture

That would solve the problems with serious distortion and pinching. It still doesn't solve the entire problem. It works badly with ordinary tileable textures and it places a serious constraint on the texture used since many more edges must fit together seamlessly. In ordinary tileable textures, the top fits the bottom and left fits right. A texture can further more be made to be tileable in all directions and orientations, in which case top, bottom,left and right

must all match each other. With the unfolded cube we need even more edges to match: a and b, c and d, e and f, g and h as well as 4 other edge pairs not all named in the figure. That is not easy to get, but one texture which tiles so perfectly is white noise. Brown noise[NOI 2006] would possibly work better since its intensity falls when its frequency rises, just as is the case with the landscape noise functions.

You can cut and slice a texture with noise any way you like, and it still looks fine. If white noise is good enough to provide that extra little touch of detail, then it is a good choice. One could replace the noise texture with noise in the pixel shader, but it would look odd because the noise will always be per pixel and you can never move in close to a spot and see it become larger. The spot will always be exactly one pixel large. Two examples of a texture containing white noise being applied to a mesh using the layout in Figure 101 are shown in Figure 102 and Figure 103. A very close look at the terrain in Figure 104 shows that the spots are indeed larger when observed from close range. The noise does bring the surface more to life than what you get from a solid color, but it is clearly not what you would call an outstanding result.

0,0 a b	Top	c d	1,0 i
Left	Front	Right	Back
e f 0,1	Bottom	g h	j 1,1

Figure 101: Texture coordinates selected for the six sides of a cube. It is shown unfolded inside the texture

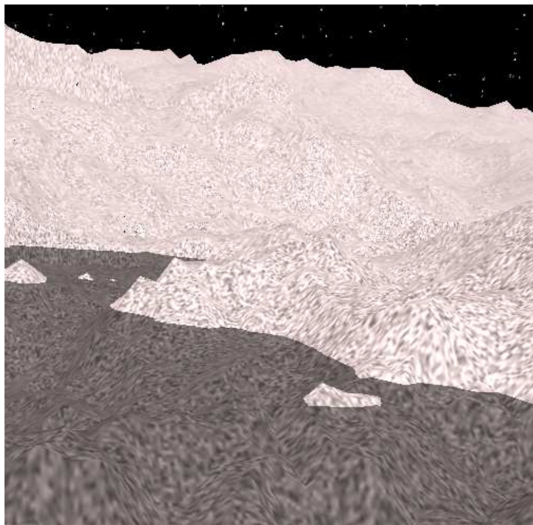


Figure 102: A terrain is textured with white noise, which changes the brightness of the terrain relative to the texture

Blending In a simple scenario you have grass at altitude 0 and rocks at altitude 1. In between you have a mix of the two. The result is that the terrain will get more colors than it might else get from a low resolution 1D texture. When using textures blending is more of a requirement, since you will generally not have a wide range of different textures to use. Each texture is assigned to an

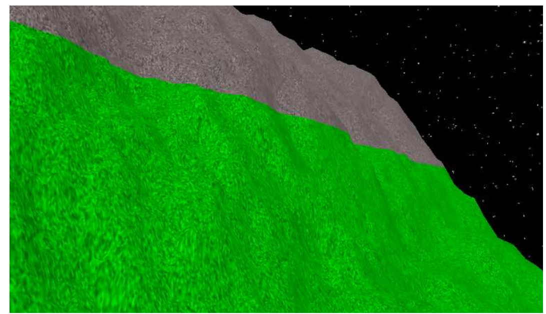


Figure 103: A terrain is textured with white noise, which changes the brightness of the terrain relative to the texture

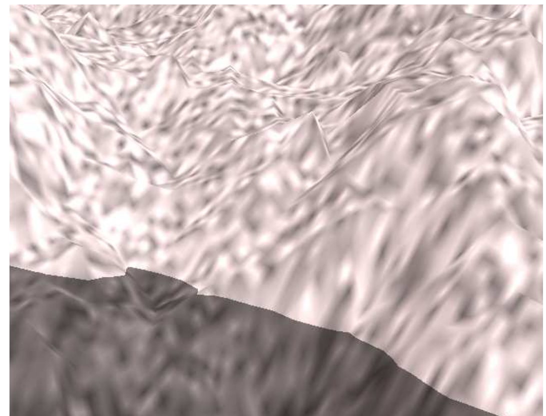


Figure 104: A very close view of a terrain textured with white noise, which changes the brightness of the terrain relative to the texture

altitude, but in between a mix or the textures, above and below, is used.

Texture conclusion Textures are an effective (computational wise) method of adding a little extra detail to large flat triangles. That explains why it is almost always used when rendering 3D geometry. It does have its problems when the surface wraps around and connects to itself, as is the case with a sphere. The usual sphere mapping method where one pre-distorts a texture and then map it using polar coordinates is not usable when textures need to be tiled. Using naive cube mapping, to avoid distortion, makes some part of the texture mapped sphere appear error free, but other areas will experience singularities.

15 View frustum culling

This technique is often used in conjunction with LOD methods. In some cases these LOD methods makes use of quad trees and are easily combined with view frustum culling. In our case, using the ROAM algorithm, triangles are structured in a bin tree and triangles are bound by a bounding sphere, which also bounds the triangles descendants as described in section 16.

We use the view frustum culling test to modify the priority of triangles used in the ROAM algorithm section 6.2. If a triangle is outside the view frustum its priority is lowered and if it is inside the

frustum its priority is raised. This of course results in more triangles in the view frustum, which is desirable. An example of this is shown in Figure 115, the view frustum is highly detailed and the outside the opposite.

Testing whether a triangles bounding sphere is inside the view frustum is not all that complicated, but a few special scenarios could arise.

Testing a bounding spheres inclusion in the view frustum results in three cases. Either the bounding sphere is completely outside, completely inside or partially inside the view frustum. When a bounding sphere is either completely inside or outside, then the triangles, to which the bounding sphere is bound, descendants are likewise all inside or outside the view frustum. It is only when a triangles bounding spheres is partially inside that its descendants needs to be checked for inclusion as well.

In sections where more than one clip plane of the view frustum meets there is a possibility of falsely ruling a triangle partial inside, when it is in fact completely outside or inside. A 2D example of this is illustrated in Figure 15. One could argue that this need not be a big problem as long as triangles inside are never ruled as outside. In our case there is a minor performance issue combined with these triangles. If a triangle is outside the view frustum but falsely ruled inside, it would probably be split into two new triangles. The two new triangles could individually have tighter bounding spheres and in the next iteration they would possibly be ruled outside the view frustum, which they are, and be merged together reconstructing the problem once more.

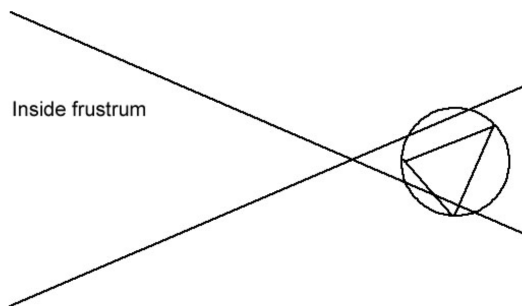


Figure 105: Bounded triangle mistakenly considered partially inside view frustum

This problem is more pronounced when clip plane meet in a near to parallel angle and when bounding spheres does not fit tightly around their triangle. A solution could be to use more clip planes like a near clip plane in the cameras origin (which at that location is redundant) as shown in Figure 15.

We transform the planes making up the view frustum from camera space to world space, as we generally need to test *many* triangles, and a transformation of all the bounding spheres to a clipping space would be more costly than transforming the clip planes to world space.

Triangles outside the view frustum could be culled, but as we want a continuous mesh at all times, this is not done. This is due to the fact that rendering and optimization of the mesh are decoupled, as described in section 6.7, and if parts of the mesh where culled in the optimization, we had to make sure, that what is rendered is alway the continuous part of the culled mesh.

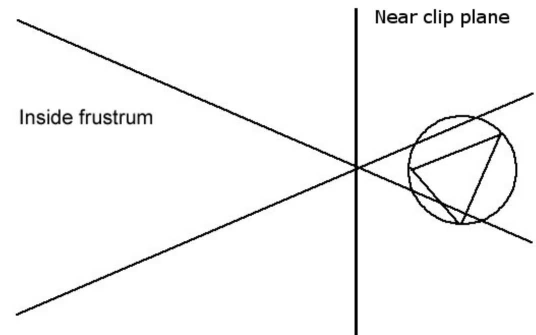


Figure 106: Adding more clip planes reduces false positive triangles

16 Various error metrics

As mentioned previously, error measurements covers a variety of metrics used in determine the quality of a simplified scene. It ranges over difference in color, normal, texture coordinate and difference in geometric models. Last mentioned is also treated in section 6.9.

It is important to realize that the term ”error” can be anything. It depends on what is important. Error is generally to be thought of as the difference between the current world representation and the infinitely detailed ”real” world, but scaled by an importance factor. Some differences may be large, but have little interest to us, and then the resulting error is not large.

Error measures are in some cases representative for our visual perception. And when simplifying scenes one can exploit deficiency in our visual perception. For example objects placed in the periphery of our field of vision contributes less in the overall notion of the image. A similar effect is seen with objects moving fast.

On the other hand there are changes which we are especial sensitive to. This could be changes to shapes, changes to colors or color intensity or changes in contrast. Thus error measures should somehow reflect characteristic from our visual perception to have optimal effect.

Error measures are used at different levels in conjunction with a level of detail algorithm. They are primarily used locally, when deciding a smaller part of a models error. This could be the error contributed by a triangle in a mesh. Error measures are used globally for a scenes total errors. This could be an accumulation of all the errors contributed from triangles in a mesh.

Global error Global errors are used primarily in two different ways. In the first place, error measures are used to define an acceptable threshold for which a scene would be accepted when simplifying it. Secondly, it is used in when optimizing a scene when certain resources are limited. For example, when a scene is limited to a certain number of polygons, one would try to optimize the use of these polygons to minimize the global error.

Local error Local error measures are used when deciding if simplifying a part of a model is feasible, but could be used the other way around also. Deciding to add more detail to a model are based on local error metrics as well. As described in section 6.1.2 the ROAM algorithm decides which triangles should be merge, simplifying the mesh, and which triangles should be split, adding detail to the mesh, based on error measures.

16.1 Error measures related to geometry

As the ROAM algorithm easily handles different types of errors we will describe some relevant types of error measures which are relevant when visualizing planets focusing on its geometry.

One class of errors relate to visibility or view culling. Different techniques for view culling are well know in computer graphics and some of these techniques are suitable for error metrics. Instead of actually culling away triangles, the result of the culling test is used for prioritizing whether a triangle should be split or not. The result of a very low priority will be a very large triangle. As we shown in section 17.5, this result in very little geometry outside the field of view, which is just as good as no geometry outside the field of view.

View frustum culling A method to test if objects are inside or outside the view frustum. We describe this in more details in section 15.

Contribution culling If a object in a scene contributes with little detail to the rendered image, one might consider to cull it all together. The objects contribution relates to its size, orientation and distance to the viewer. For terrain it is not easily implemented, but one way could involve calculation of a bounding spheres contribution to the image, for example the triangles bounding spheres section 7. In the case of the ROAM algorithm a triangles size and the mesh subdivision level are correlated, so a analogy to contribution culling is already employed here.

Occlusion culling A method to decide whether some object are occluded by other objects and therefore culled. In a terrain mesh, an analogue situation could happen when mountains are occluding other parts of the terrain. So, strictly speaking, it is not one object occluding another object, but self occlusion. Having the mesh organized in quadtree or in another way having it sorted in a spatial way would make it possible to determine if parts of the mesh is occluding for other parts. Occlusion queries are a method with which modern graphics cards can tell exactly how many pixels an object contributed with. One could render the scene and then turn off frame-buffer and z-buffer writes and re-render and this time see which triangles are actually visible. Those not visible could receive a lower priority for the next frame.

Backface culling Backfacing triangles should never be visible for terrain and could therefor be culled. As meshes could be used to represent water surface and the atmosphere and these could be visible from both above and below, backface culling should not be used here, or the decision on weather a face is back facing or not, should be reversed when moving from inside to outside the ocean or the atmosphere.

16.1.1 Surface layers

Because we are using one mesh to represent terrain and another mesh to represent the surface of the ocean (and yet another as atmosphere), a situation which leads to sizeable visual error arises.

At the coast line the two surfaces are near parallel and as the terrains LOD changes, the spatial location of the coast line could move. As the coast line has large contrast this relocation is very conspicuous. Geo morphing would help little, since a small change in the height of the terrain still could lead to a large vertical offset of the coast line. Only with morphing it would not be a sudden pop, but rather a quick motion of the coastline inwards or outwards, depending on whether the land was lowered or raised.

The cause of this phenomena lies in the problem we are trying to solve. We are trying to find the intersection between two surfaces and because these planes are near parallel the problem is ill-conditioned. A relative little change in the input data to the problem, results in a relative large change in the solution as illustrated in Figure 107.

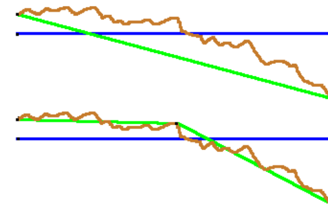


Figure 107: Finding the intersection between the surface of the ocean and the terrain is an ill-conditioned problem. The brown line is the terrain the green line is approximating and the blue line is the ocean.

To minimize coastal popping we try to place a vertex from a triangle close to the intersection in the terrain mesh. This is done by prioritizing triangles higher if they have vertices on either side of sea level, as seen on Figure 108 where the triangles spanning the coast line are given a very high priority. The effect is that from high orbit, the planet will still have highly detailed coast lines, even though the overall number of triangles is small.

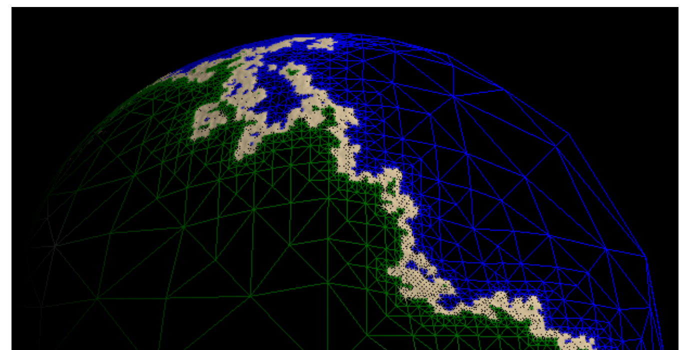


Figure 108: The triangles spanning the coast line have high priority and are subdivided more than triangles which are entirely on land or entirely in the ocean. The coast is drawn in a yellowish color➠

This resembles the bisection method for root-finding. Every time a

triangle which has vertices above and below water level, its priority is raised and eventually subdivided. This leads to a closer approximation to the intersection point much like how the bisection method approximates the root. This method is guaranteed to converge when values are of opposite sign (below and above sea level) but converges slowly. When the end values are *not* of opposite sign, then the situation is less optimal.

Sample frequency for coastline optimization Shannons sampling theorem states that in order to correctly sample (for later reconstruction) a signal with a frequency f, one needs to sample with at least 2f. This implies that for a correct sampling of a terrain which consists of frequencies up to f, one need the triangles to be small enough so that the vertex distance is smaller than half that f's corresponding wavelength.

We do not want to make such a sampling. We merely want to sample well enough to differentiate between a signal with only a DC[12] component and one with an AC[13], where we even simplify the signal to become plus, when above sea level, and minus, when below sea level, and disregard the actual values, and actual frequencies. This boils down to sampling in a way so that we can see if the signal changes sign or if it is steady state. For this to be correct, we need not to sample at a frequency of 2f, since we are not interested in the signals actual frequency, but only if it is AC or DC. Sampling at a frequency higher than the lowest frequency will be good enough.

This can easily be confirmed by a simple example. Imagine that the point A is just above the sea and that the point B is just above the sea, while all points in between are below.If we sample at point A and at point B, then we do not detect the ocean. If we sample any distance from A which is even so little smaller than the distance between A and B, then we sample at a point below the sea level, and detect the ocean correctly.

We can therefore conclude that if we want to detect ocean areas down to a size of S, then we need to sample (place vertices) closer than S. If the vertices are S or farther apart, then we can potentially miss the ocean and thereby miss the coastline which we want to refine. A good strategy will therefore be to refine the mesh to the point where the vertices are close enough apart and only then start giving high priorities to coastal triangles.

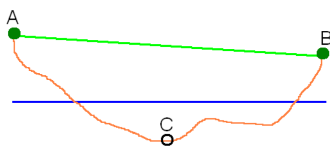


Figure 109: If the distance between sample points A and B is too large, then a coast line can potentially be missed when both are above or below sea level. If a third sample point C was added at the midpoint then the ocean would be correctly detected.

If sections between surface, for example between the terrain surface and water surface, is not being 100% transparent, then triangles of the terrain being below the water could be prioritized lower, if the viewer is above the sea level and vice versa.

[12]Direct current, no variation but a steady value

[13]Alternating current, a signal with a periodic change in value

16.1.2 Silhouettes

As with coast lines, silhouettes generally has a large contrast and therefor triangles being part of an objects silhouettes are prioritized high. Looking at a planet at far distance, higher LOD on the silhouettes makes the planet appear round. Looking at a mountain its silhouettes also needs to be detailed so it does not look unnaturally jagged. Looking at the surface normal of a triangle ,and its neighbors, compared to the view vector, we can determine if the triangle is part of a silhouette. If the triangles normal ”points towards” the viewer and one of its neighbors normals points away from the viewer, then the edge between the two forms a silhouette. The same exact method is used when calculating volume shadows in other applications. In Figure 110 and Figure 111 an example of a planet with its silhouette at higher LOD is shown. In Figure 111 it is also noticeable that the right half of the planet is higher detailed than the left. This is due to the fact that the right half is facing the camera.

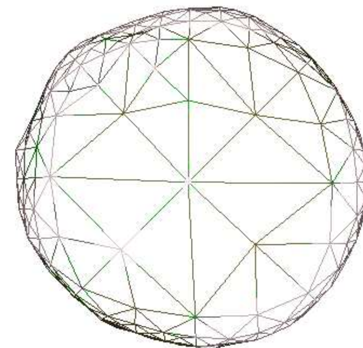


Figure 110: The planets silhouette has a higher LOD to make it appear round.

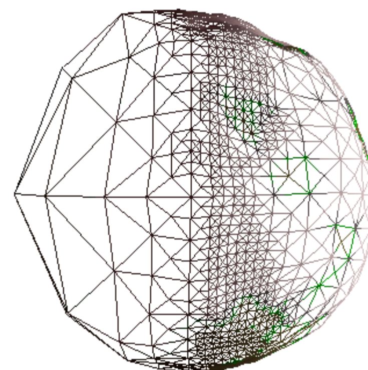


Figure 111: The same planet turned to show the silhouette. To the left is the part of the planet facing away from the viewer, and to the right the part facing the viewer.

16.1.3 Visual perception

Alternative error metric are related to the nature of our visual capabilities. Objects placed in the periphery of our field of vision contributes less to the perceived image. If the viewer is moving

fast, they would contribute even less, since they will more relatively more in the field of view, than would objects at the center of view, so they could therefor be prioritized lower.

17 Test and analysis

Throughout this paper, we have tried to show by example how the methods that we describe can be used to solve the problem of generating an artificial world. Therefor we have in a sense shown that the methods do in fact work and even when they sometimes do not work quite as well as one would have liked. Much of the testing is therefore located elsewhere along with the explanation of how the results were obtained.

This section provides a few extra tests and benchmarks which have been neglected until now.

17.1 Frame coherence

Whether split-and-merge outperforms split-only section 6 depends mainly on how much the bin tree changes from frame to frame.

A mesh with 50 000 leaf triangles were used for testing. Such a mesh is stored in a bintree with approximately 100 000 triangles. High Frame coherence means that the bin tree is changed very little from frame to frame. We tested fast and slow flight at high and low altitudes and the results were promising.

We count change in a tree as nodes which exist in only one of the two trees. Both leafs and intermediate nodes are counted. If a node is moved from one location to another, then that will result in two changes. One removal and one insertion.

High altitude low velocity A flight at 100 km above ground level with a velocity of 200 kph resulted in a change in the tree of 300 to 400 depending of the terrain underneath. That is less than one half of percent.

High altitude very high velocity A flight of 100 km above ground level with a velocity of 20 000 000 kph resulted in a change of 20 000 to 28 000. This is a large change of almost 30 percent.

Low altitude low velocity A flight at 100 m above ground level with a velocity of 200 kph caused a change of 2 700 to 3 200, which is less than 4 percent.

Low altitude very high velocity A flight 100m above ground with a velocity of 20 000 000 kph resulted in a change of 200 000. This is a completely new tree.

The frame coherence conclusion It is clear that when the movement is not insanely fast then there really is a high degree of frame coherence. The entirely new tree at the fast flight close to ground, is what could be expected, but at the same time, even that is acceptable since no fine terrain details will be visible at that speed, so a very rough mesh will be acceptable.

For a large mesh the tree changes very little from frame to frame, and it is very wasteful to rebuild the entire tree each time, when a change of 4% is sufficient to obtain an optimal mesh.

17.2 Multiple threads

We decoupled the rendering from the mesh optimization to let a computer with multi core processor utilize all its powers, which would not be possible with a single thread. The performance on two systems with identical memory and graphics card were compared.

SINGLE - 3.2GHz single core, Galaxy GF6800, 1GB RAM
DUAL - 3.0GHz dual core, Galaxy GF6800, 1GB RAM

The test showed that a mesh with 100 000 visible triangles rendered with 15FPS and 1.5 UPS (optimizations per second) on SINGLE while it was 60FPS and 3.2UPS on DUAL. This relationship of double UPS on a dual core and more than double frame rate on a single core was quite constant during all tests.

The multi thread conclusion While previous single thread ordinary ROAM implementations would not use more than one core, we clearly use both, and with a high degree of efficiency.

17.3 Fractal terrain generating algorithm

A problem, though we elsewhere talk about this as being a desirable attribute, with the midpoint displacement algorithm is that when a point is offset into its position, it never again moves. This is good because it should not move around, but it is bad because the initial position is selected based on only the previous subdivisions. This means that the positions selected early on in the subdivision can have a profound, too profound, impact on the final mesh. As seen in Figure 18, the early displacements form dominant features in the terrain. This is shown on a real terrain, as seen from above, in Figure 112 This is very prominent in our terrain, when, by chance, a large early displacement is followed by a few small displacement. Generally speaking, a displacement in level 0 can be influenced fairly strongly on level 1, but from level 1 and downwards the displacements will be too weak to make much of a difference, and the original large displacement will end up forming a very straight ridge or groove. The more the maximal offset is diminished each step, the stronger this effect is.

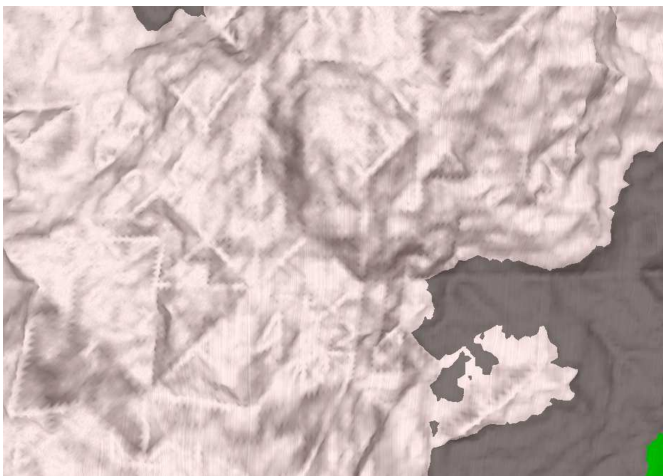


Figure 112: A terrain with visible ridges from early midpoint displacements➠

While ridges can be good, the ridges from bad midpoint displacement are certainly bad. They appear very unnatural with their straight lines which are arranged in a grid of lines at 45 degree offsets.

A comparison with a landscape generated from Perlin noise and not midpoint displacement shows no ridges.

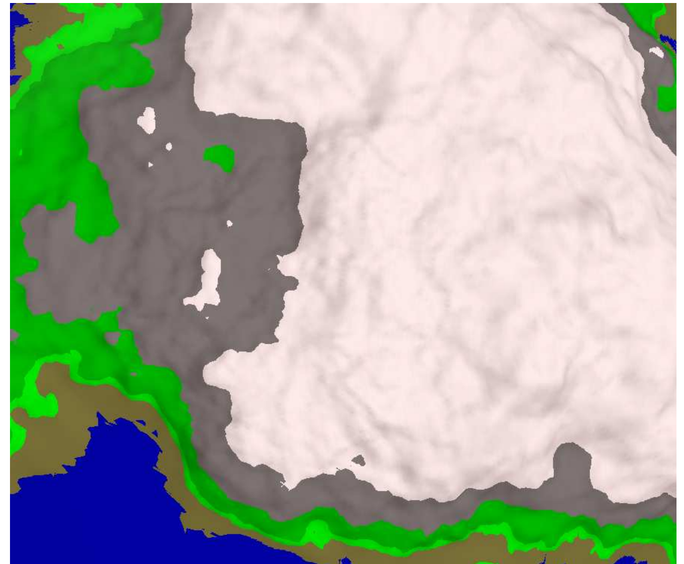


Figure 113: A terrain generated with Perlin noise and not showing any visible ridges➠

17.4 Effect of different number of triangles

The more triangles in the landscape, the more work is required for each update. The question is how much of a difference it really makes to double the number of triangles. Figure 114 shows the same scene with four different resolutions. From top to bottom, the triangle count is 100 000 50 000 25 000 and 5 000. While it is evident that the highest triangle count does provide the best visual experience, it is not that much of a different from 100 000 to 50 000. Even the low resolution view is reasonable. It is generally more pleasing to have a smooth and high frame rate than having a pretty terrain which can not be drawn and updated at a reasonable speed.

This observation falls well in thread with the following quotation.

```
Today the absolute number of triangles is not as
important. As of 2003, games such as "Unreal 2"
have been released which render up to 200 000
triangles per frame. An attractive terrain
triangulation takes some 10 000 triangles. This
means that it is no longer important if we need
10 000 or 20 000 triangles for the terrain mesh as
long as it is done fast enough.
Daniel Wagner, GameDev
```

The terrain looks better the more triangles is used, but unless the terrain is static, we have more need for speed in rendering and optimization.

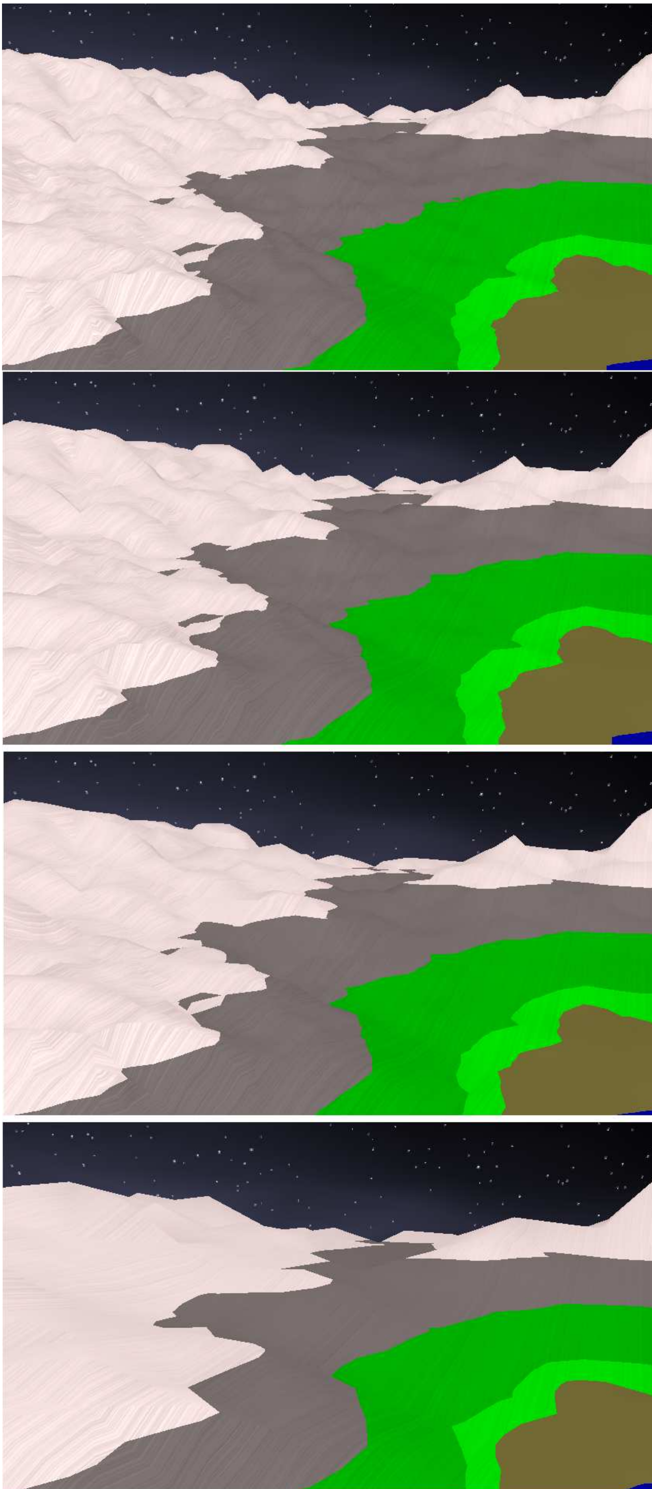


Figure 114: Four identical views with different number of triangles. From top to bottom, the triangle count is 100 000, 5 000, 25 000 and 5 000➠

17.5 Visible triangles

The way ROAM works means that all the triangles in the mesh are woven together as a whole. You can not optimize one section without influencing the rest of the mesh to some degree.

Figure 115 shows a planet which has been optimized for a view point close to the surface and looking ”north”. As expected the detail level of the mesh drops off sharply when it is not inside the view frustum. Only 612 triangles, out of a mesh with 100 000 triangles, are outside the visible area. When looking at a planet from a distance, where the entire planet is contained in the view frustum, then naturally no triangles are wasted outside what is seen, but even when moving very close to the terrain, we see that the number of triangles outside the view frustum, for a 100 000 triangle mesh, never became larger than 800. This shows that even though ROAM does waste triangles on areas which are clearly not visible, this waste is very small.

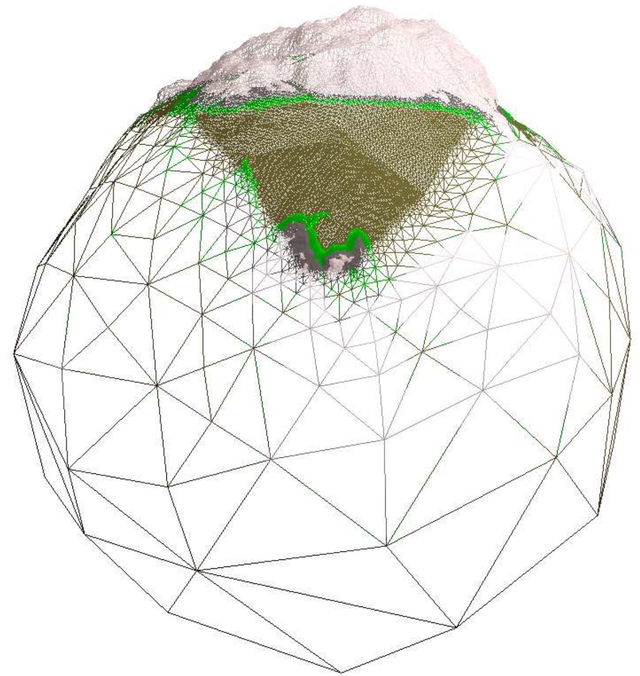


Figure 115: An outside view of the mesh being optimized for a view point close to the planets surface

18 Conclusion

We found that it is not that difficult to quickly implement a planet sized mesh with CLOD and a decent visualization of the terrain types. While we didn't have the time to implement real split and merge, tests showed us that given the degree of frame coherence that we measured, split-and-merge would speed up the optimization significantly compared to split-only.

The fractal generation clearly needs more than simple midpoint displacement. All too often distinct ridges and grooves were visible in the landscape. Even if midpoint displacement was assisted by some of the other simple landscape generating functions, the ridges will most likely show up again.

Decoupling mesh optimization and rendering was a clear success. The old style ROAM is quite difficult to properly split into parallel processing, but by at least separating rendering and optimization, a system with more than one core would always gain speed compared to a single thread implementation.

ROAM is getting old We feel that ROAM is becoming an outdated algorithm which was clearly designed to work in a single thread and on the CPU. With the recent developments in landscape generating algorithms, ROAM is starting to look old. It can not be moved to the GPU and it is not good for multiprocessing systems. This means that one has to struggle with moving the mesh to the GPU every time it is changed, rather than leaving it there for some length of time, and it means that modern computers will not benefit from their extra processors.

If we were to do this project again, we would clearly choose an algorithm which is more suited for GPU processing. When the mesh remains on the GPU, one can render a very suboptimal mesh with very simple error metrics, and still get good results, simply because one can use many more triangles.

Fractal terrain generation We realized that midpoint displacement is not really suitable for triangle meshes. Ridges or creases are a very common artifact of that combination, and there is no real easy solution to the problem.

19 Future work

It would be interesting to see how ROAM could be implemented to run on the new geometry shaders, where the algorithm might be able to exist entirely on the GPU, which would allow it to work very fast and without constantly moving large meshes across the bus.

Another interesting thing to work with is artificial life in a procedurally generated world with CLOD. As with rivers, the creatures living in the world can move around and interact without the need to generating the entire world. Only the regions of the world near by need be detailed. It would be interesting to dive into such a world, and observe a group of BOID-like creatures roaming around.

An obvious thing that needs more experimentation is the generation of an entire universe. How well can one generate galaxies, and visualize them, through hashing and procedural algorithms? Could the universe be populated by civilizations which would change their status based on the passing of time, and only spring into live detail when a lonely space traveler arrives at their planet? That way a civilizations current state, when the traveler arrives, would be defined by the arrival time and yet another pseudo random number. Only if the traveler actively changes the civilization, the system needs to store detailed state information.

In broad terms, the future work is about testing the boundaries. How photo realistic can the universe be, how detailed can the inner workings become and how large?

List of Figures

1 Planetary landscape with continuous level of detail, rendered real-time . . . 1
2 Showing same model with different Levels of Detail 3
3 Showing models with different Levels of Detail at different distance . . . 3
4 Showing View dependent LOD . . . 4
5 Showing the distant, large mountain and the nearer, smaller pyramid . . . 5
6 The standard ROAM triangle . . . 5
7 Different subdivision levels [Duchaineau et al. 1997] 6
8 Cracks in a mesh which was not recursively split [Luebke 2003] . . . 6
9 Recusive triangle splitting [Duchaineau et al. 1997] 6
10 Splitting and merging with ROAM [Duchaineau et al. 1997] . . . 7
11 ROAM subdivision in 8 steps . . . 8
12 A binary triangle tree containing a ROAM . . . 8
13 Triangles ordered as strip or list . . . 10
14 A winding, but ordered, way through the ROAM . 10
15 Projected ROAM error [Duchaineau et al. 1997] . . 11
16 A minimal bounding sphere for a right angled isosceles triangle . . . 12
17 The bounds being broken by subdivision . . . 13
18 The bounds being broken by subdivision . . . 13
19 Naive placement of near and far clipping planes for arbitrary geometry . . . 14
20 Almost optimal placement of near and far clipping planes for arbitrary geometry, taking into account that not all the geometry is inside the view field . . 14
21 Optimal placement of near and far clipping planes for a sphere which is partially outside the view field. The green line is the line of site for the horizon point . . . 15
22 The optimal near and far clipping points for a view frustum observing a sphere. The near clipping point A' is A projected onto the view vector and A is the intersection of the view frustum and the sphere. The far clipping point B' is B projected onto the view vector and B is the point where the vector from the eye is the spheres tangent . . . 15
23 The intersection points of a view frustum and a sphere 15
24 Height map and resulting mesh . . . 17
25 One level of midpoint displacement[Polack 2003] . 18
26 Midpoint Displacement at level one . . . 19
27 Midpoint Displacement at level two . . . 19
28 Midpoint Displacement at level four . . . 19
29 Midpoint Displacement at level six . . . 19
30 A different midpoint for spheres . . . 19
31 Fault line with 1024 faults . . . 20
32 Fault line with 1024 faults and average smoothing of size 100 . . . 20
33 Fault line with 1024 faults and average smoothing of size 1000 . . . 20
34 One fractal function generated using midpoint displacement, F_1 . . . 21
35 One fractal function generated using midpoint displacement, F_2 . . . 21
36 One multi fractal function generated by multiplying F_1 and F_2 . . . 21
37 Ordinary Perlin noise . . . 21
38 The absolute value of Perlin noise. The valleys are now sharply defined and no longer rounded . . . 21
39 The inverted perling noises absolute value, which makes the valleys into sharp ridges . . . 21
40 Midpoint displacement not behaving well when implemented as a ridged function. The midpoint M between A and B is moved up to a positive value, without generating any ridges. The function is more flat than before. . . . 21
41 A scene showing ridged terrain from MojoWorld . 22
42 Midpoint displacement working correctly together with ridges. The dotted line represents the original data generated by midpoint displacement, while the solid line is the ridged version. . . . 22
43 Perlin noise used to generate an image. Linear interpolation was used . . . 22
44 Perlin noise used to generate an image. Cosine interpolation was used . . . 22
45 Noise function, $n(x)$ . . . 25
46 Designed function, or rather predefined samples in x, $d(x)$ . . . 25
47 Scaling function, $s(x)$ . . . 25
48 Sum of scaled functions, $n(x)(1-s(x)+d(x)s(x)$. 25
49 Blending performed with three types of blend functions . . . 26
50 Midpoint displaced values in red, design in green and combination in blue . . . 26
51 Design space and it projection to a sphere . . . 27
52 Closeup of blocks caused by pointsampling. The colors are defined to be the normal vectors to better show the form of the mesh . . . 27
53 Closeup of bilinear sampling showing no blocks. The colors are defined to be the normal vectors to better show the form of the mesh . . . 28
54 The bilinear resampling . . . 28
55 A very rudimentary design for a meteor crater containing only the outer rim and the central point . . . 28
56 A crater blended into a terrain at three different levels. From top to bottom the blend is 1.0 0.5 and 0.1 29
57 A crater blended into a terrain at three different levels. From top to bottom the blend is 1.0 0.5 and 0.1 29
58 Manually generated height map used as design . 30
59 Design merged with noise, using the smooth blend function. It is apparent that the resulting landscape is never entirely equal to the design, though it is very close around the center, where the blend function gives heigh weight to the design . . . 30
60 Design merged with noise, using the blend function with a central area totally dominated by the design. You can clearly see that the E and S are not influenced by the noise function, but are entirely defined by the design . . . 30
61 Design should be visible, but the triangles, which are to be transformed into the design, are not . . . 30
62 Design is becoming visible when we do an extra subdivision. Note that this is not the same terrain as in Figure 61 . . . 31
63 Design can be distorted when angle of "impact" is very low . . . 31
64 Four levels of zoom into a lonely river on a planets surface. The first image shows how rough the surroundings of a river can be, while the last one shows the flow of a river which ends up filling an irregularly shaped lake . . . 34
65 Twelve discrete steps in flowing water . . . 35
66 The calculated path of a river in a 3D landscape, showing both the winding flow and river basins filling up . . . 35
67 A mesh optimized by flow calculation for a river . . 36

68 A river forms a system of interconnected lakes . . . 36
69 Two lakes are created by and connected through a single river 36
70 A river flows through the landscape, forms a single lake and then moves on till it reaches the ocean . 36
71 Two different bounding areas for the same collection of points 37
72 The principal axis, and unit sphere, for a group of 3D positions 37
73 A lake with no visible inlet or outlet 38
74 A lake with both an inlet and an outlet 38
75 The ground a muddy color under water. The light is also made more ambient to simulate the scattering effect of the water and particles therein 39
76 Looking towards the day break. The sky is starting to become more bright and blue in the lower right corner, but the stars are still slightly visible, more so when looking away from the light 39
77 The sun is back on the sky, and we can no longer see the stars. Only the clear blue sky is visible . 40
78 From a position high above the atmosphere, it looks like a semi transparent layer of air. This is not very realistic, but it shows that the atmosphere can easily look very different depending on the observers position 40
79 The atmosphere is colored white, but the alpha component is defined by a noise function to emulate cloud cover 40
80 Vertex normal calculated as average of surrounding triangle normals 40
81 Artifacts from Phong shading a ROAM generated mesh 41
82 Demonstrating how Phong lighting a pyramid will generate artifacts which are in no way looking smooth 41
83 Landscape with plenty of pyramids and no attempt to hide them 42
84 Landscape with plenty of pyramids which are made less apparent by blending the vertex normals with a vertical normal vector to simulate ambient light which changes across the face of the spherical planet 42
85 A surface can be in darkness even though it is facing the light 42
86 Terrain colored based only on its altitude 43
87 A terrain is colored according to its altitude. The colors are noticeably arranged in bands where the border between two colors are at one specific altitude. 43
88 A terrain is colored according to its altitude and it is clear that the borders between colors follow the contour lines. 43
89 A terrain is colored according to its altitude and a noise offset which perturbs the color selection, so the border between colors no longer occur at specific altitudes, though snow is still dominant at high altitudes, water is below a certain altitude, and grass is dominant at the lower areas. 43
90 A terrain is colored according to its altitude and an extra noise function which perturb the altitudes. It is now clear that the borders between colors do not entirely follow the contour lines. 44
91 A terrain is colored according to its altitude 44
92 A terrain is colored according to its altitude where the altitude is perturbed by a noise function before a color is selected. The perturbation is low frequency and with medium amplitude 44
93 A terrain is colored according to its altitude where the altitude is perturbed by a noise function before a color is selected. The perturbation is medium frequency and with medium amplitude 44
94 A terrain is colored according to its altitude where the altitude is perturbed by a noise function before a color is selected. The perturbation is high frequency and with low amplitude 45
95 A terrain is colored according to altitude and its gradient along with another noise function. On slopes facing "left" there will sometimes be patches of light green grass, when the noise function also allows it. 45
96 An ocean is colored according to its latitude and only the latitude. Over a certain latitude, the ocean is frozen, and rendered as white ice 45
97 An ocean is colored according to its latitude and a noise function. Over a certain latitude plus minus noise, the ocean is frozen, and rendered as white ice 45
98 Even textures which tile seamlessly generates patterns when used on a large scale 46
99 Tileable textures can be used if they are very homogeneous 46
100 A line artifact from singular U-texture coordinates and a noise texture 46
101 Texture coordinates selected for the six sides of a cube. It is shown unfolded inside the texture 47
102 A terrain is textured with white noise, which changes the brightness of the terrain relative to the texture 47
103 A terrain is textured with white noise, which changes the brightness of the terrain relative to the texture 47
104 A very close view of a terrain textured with white noise, which changes the brightness of the terrain relative to the texture 47
105 Bounded triangle mistakenly considered partially inside view frustum 48
106 Adding more clip planes reduces false positive triangles 48
107 Finding the intersection between the surface of the ocean and the terrain is an ill-conditioned problem. The brown line is the terrain the green line is approximating and the blue line is the ocean. 49
108 The triangles spanning the coast line have high priority and are subdivided more than triangles which are entirely on land or entirely in the ocean. The coast is drawn in a yellowish color 49
109 If the distance between sample points A and B is too large, then a coast line can potentially be missed when both are above or below sea level. If a third sample point C was added at the midpoint then the ocean would be correctly detected. 50
110 The planets silhouette has a higher LOD to make it appear round. 50
111 The same planet turned to show the silhouette. To the left is the part of the planet facing away from the viewer, and to the right the part facing the viewer. 50
112 A terrain with visible ridges from early midpoint displacements 52
113 A terrain generated with Perlin noise and not showing any visible ridges 52
114 Four identical views with different number of triangles. From top to bottom, the triangle count is 100 000, 5 000, 25 000 and 5 000 53

115 An outside view of the mesh being optimized for a view point close to the planets surface 53
116 Blending performed with three types of blend functions 60
117 Design merged with noise, using the smooth blend function. It is apparent that the resulting landscape is never entirely equal to the design, though it is very close arround the center, where the blend function gives heigh weight to the design. 61
118 Design merged with noise, using the blend function with a central area totally dominated by the design. You can clearly see that the E and S are not influenced by the noise function, but are entirely defined by the design 61
119 A pre-designed ”crater” was blended with a noise function. The crater was a wide stroke circle which had been smoothed slightly to give it more rounded edges. The blend function was turned somewhat down, in order to not let the design become too dominant. A somewhat realistically looking noisy crater is clearly visible, and it blends perfectly with the landscape. The designed crater model was very simplistic which may have resulted in too tall sides. 62
120 A crater blended into a terrain at three different levels. From top to bottom the blend is 1.0 0.5 and 0.1 63
121 A crater blended into a terrain at three different levels. From top to bottom the blend is 1.0 0.5 and 0.1 64
122 The calculated path of a river in a 3D landscape, showing both the winding flow and river basins filling up 65
123 A river flows through the landscape, forms a single lake and then moves on till it reaches the ocean . . 66
124 A lake with both an inlet and an outlet 67
125 Looking towards the day break. The sky is starting to become more bright and blue in the lower right corner, but the stars are still slightly visible, more so when looking away from the light 68
126 The sun is back on the sky, and we can no longer see the stars. Only the clear blue sky is visible . . . 69
127 From a position high above the atmosphere, it looks like a semi transparent layer of air. This is not very realistic, but it shows that the atmosphere can easily look very different depending on the observers position 70
128 The atmosphere is colored white, but the alpha component is defined by a noise function to emulate cloud cover. 71
129 A terrain is colored according to its altitude. The colors are noticeably arranged in bands where the border between two colors are at one specific altitude. 72
130 A terrain is colored according to its altitude and it is clear that the borders between colors follow the contour lines. 72
131 A terrain is colored according to its altitude and a noise texture which perturbs the color selection, so the border between colors no longer occur at specific altitudes, though snow is still dominant at high altitudes, water is below a certain altitude, and grass is dominant at the lower areas. 73
132 A terrain is colored according to its altitude and an extra noise function which pertubs the altitudes. It is now clear that the borders between colors do not entirely follow the contour lines. 74
133 One noise function defined the altitude of the terrain and another noise influenced certain terrain characteristics in order to create irregularly shaped muddy areas in the lowland. 75
134 A terrain is colored according to altitude and its gradient along with another noise function. On slopes facing ”left” there will sometimes be patches of light green grass, when the noise function also allows it. 76
135 Four identical views with different number of triangles. From top to bottom, the triangle count is 100 000, 50 000, 25 000 and 5 000 77
136 A terrain with visible ridges from early midpoint displacements➠ 78
137 A terrain generated with Perlin noise and not showing any vissible ridges 79
138 The triangles spanning the coast line have high priority and are subdivided more than triangles which are entirely on land or entirely in the ocean 80

List of Tables

1 A distant large object and a near small object projected to same screen size with constant Level of Detail 5

Listings

1 Pseudo code for optimization by splitting 6
2 Pseudo code for Midpoint Displacment 18
3 Pseudo code for fault line algorithm 20
4 Hashing with permutation tables 24

References

ARTS, E., 2006. http://www.spore.com/. web page.

BELHADJ, F., AND AUDIBERT, P. 2005. Modeling landscapes with ridges and rivers: bottom up approach. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*.

BRADLEY, D. 2003. *Evaluation of Real-Time Continuous Terrain Level of Detail Algorithms*. PhD thesis, Carleton University.

CLARK, J. H. 1976. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*.

COSMAN, A., AND SCHUMACKER, R. 1981. System strategies to optimize cig image content. In *Proceedings of 1981 Image II Conference*.

DUCHAINEAU, M., WOLINSKY, M., SIGETI, D. E., MILLER, M. C., ALDRICH, C., AND MINEEV-WEINSTEIN, M. B. 1997. Roaming terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*.

DUCHAINEAU, M., 2006. http://www.cognigraph.com/roam_homepage/roam2. Web page.

EBERT, D. S. 2003. *Texturing and Modeling - A Procedural Approach 3.Ed.* Morgan Kaufmann.

ELIAS, H., 2006. http://freespace.virgin.net/hugo.elias/models/m_perlin.htm. Web page.

ELITE, 1984. http://www.frontier.co.uk/. web page.

GREUTER, S., PARKER, J., STEWART, N., AND LEACH, G. 2003. Real-time procedural generation of 'pseudo infinite' cities. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*.

HOPPE, H. 1996. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*.

HOPPE, H. 1997. View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*.

J. S. FALBY, M. J. ZYDA, D. R. P., AND MACKEY, R. L., 1993. Npsnet: Hierarchical data structures for real-time three-dimensional visual simulation.

KELLEY, A. D., MALIN, M. C., AND NIELSON, G. M. 1988. Terrain simulation using a model of stream erosion. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*.

LAGAE, A., AND DUTRÉ, P. 2006. Long period hash functions for procedural texturing. In *Vision, Modeling, and Visualization 2006*.

LEVENBERG, J. 2002. Fast view-dependent level-of-detail rendering using cached geometry. In *VIS '02: Proceedings of the conference on Visualization '02*.

LINDSTROM, P., AND PASCUCCI, V. 2002. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*.

LINDSTROM, P., KOLLER, D., RIBARSKY, W., HODGES, L. F., FAUST, N., AND TURNER, G. A. 1996. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*.

LINDSTROM, P., HOLLER, D., HODGES, L. F., RIBARSKY, W., FAUST, N., AND TURNER, G. 95. Level-of-detail management for real-time rendering of phototextured terrain. Tech. rep., Georgia Institute of Technology.

LOSASSO, F., AND HOPPE, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*.

LUEBKE, D. 2003. *Level of Detail for 3D Graphics*. Morgan Kaufmann.

MACIEL, P. W. C., AND SHIRLEY, P. 1995. Visual navigation of large environments using textured clusters. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*.

MANDELBROT, B. B. 1977. *Fractals : form, chance, and dimension*. Freeman.

MANDELBROT, B. B. 1983. *The fractal geometry of nature*. Freeman.

MUSGRAVE, F. K., AND MANDELBROT, B. B. 1991. The art of fractal landscapes. *IBM J. Res. Dev.*.

MUSGRAVE, F. K., KOLB, C. E., AND MACE, R. S. 1989. The synthesis and rendering of eroded fractal terrains. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*.

MUSGRAVE, K., 2006. http://www.pandromeda.com/. web page.

NISHIMORI, H., AND OUCHI, N. 1993. Formation of ripple patterns and dunes by wind-blown sand. *The American Physical Society*.

2006. http://en.wikipedia.org/wiki/colors_of_noise. Web page.

OLANO, T. M., AND YOO, T. S. 1993. Precision normals (beyond phong). Tech. rep., Department of Computer Science, University of North Carolina.

OLSEN, J., 2004. Realtime procedural terrain generation. Web page.

PARISH, Y. I. H., AND MüLLER, P. 2001. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*.

PERLIN, K., AND HOFFERT, E. M. 1989. Hypertexture. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*.

PERLIN, K. 1985. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*.

PERLIN, K., 2006. http://mrl.nyu.edu/ perlin/. Web page.

POLACK, T. 2003. *Focus on 3D Terrain Programming*. The Premier Press.

PRUSINKIEWICZ, P., AND HAMMEL, M. 1993. A fractal model of mountains with rivers. In *Proceeding of Graphics Interface '93*.

REEVES, W. T. 1983. Particle systems-a technique for modeling a class of fuzzy objects. In *SIGGRAPH '83: Proceedings of the*

10th annual conference on Computer graphics and interactive techniques.

ROETTGER, S., HEIDRICH, W., AND SLUSSALLEK, P., 1998. Real–time generation of continuous levels of detail for height fields. Proc. 6th Int. Conf. in Central Europe on Computer Graphics and Visualization 1998.

SNOOK, G. 2003. *Real Time 3D Terrain Engines Using c++ And Directx 9*. Charles River Media.

TERRAGEN, 2006. http://www.planetside.co.uk/terragen/. web page.

WATT, A. W. . M. 1992. *Advanced Animation and Rendering Techniques*. Addision Wesley. Pages 23-27.

WEHOWSKY, A., 2001. Procedural generation of a 3d model of mit campus. http://graphics.lcs.mit.edu/ seth/pubs/WehowskyBMG.pdf.

YAN, J. K. 1985. Advances in computer generated imagery for flight simulation. *IEEE Computer Graphics and Applications*.

A Enlarged figures

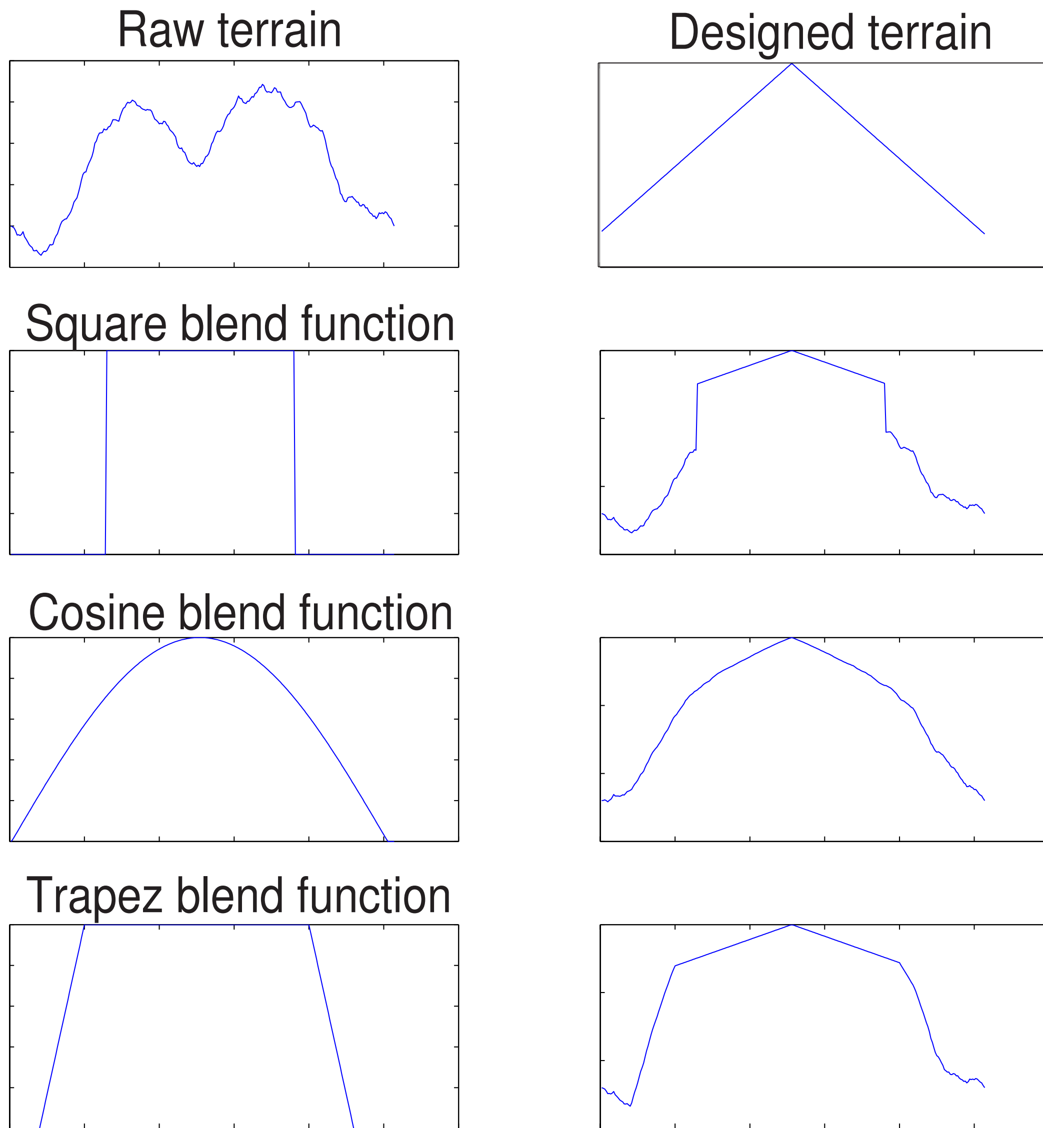


Figure 116: Blending performed with three types of blend functions

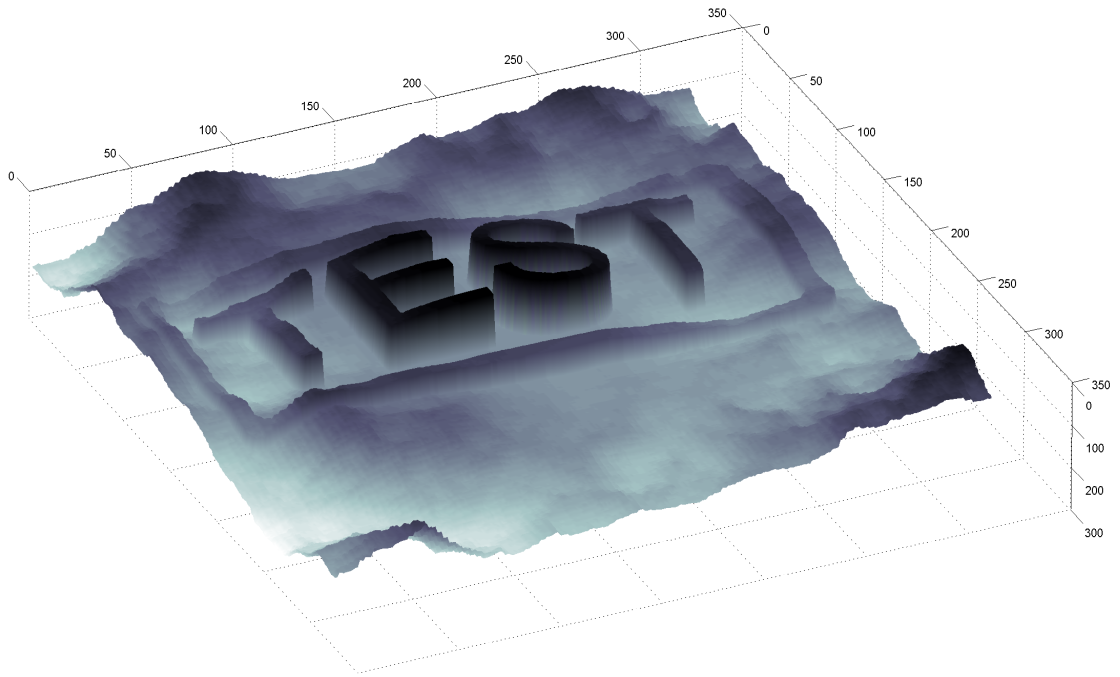


Figure 117: Design merged with noise, using the smooth blend function. It is apparent that the resulting landscape is never entirely equal to the design, though it is very close arround the center, where the blend function gives heigh weight to the design.

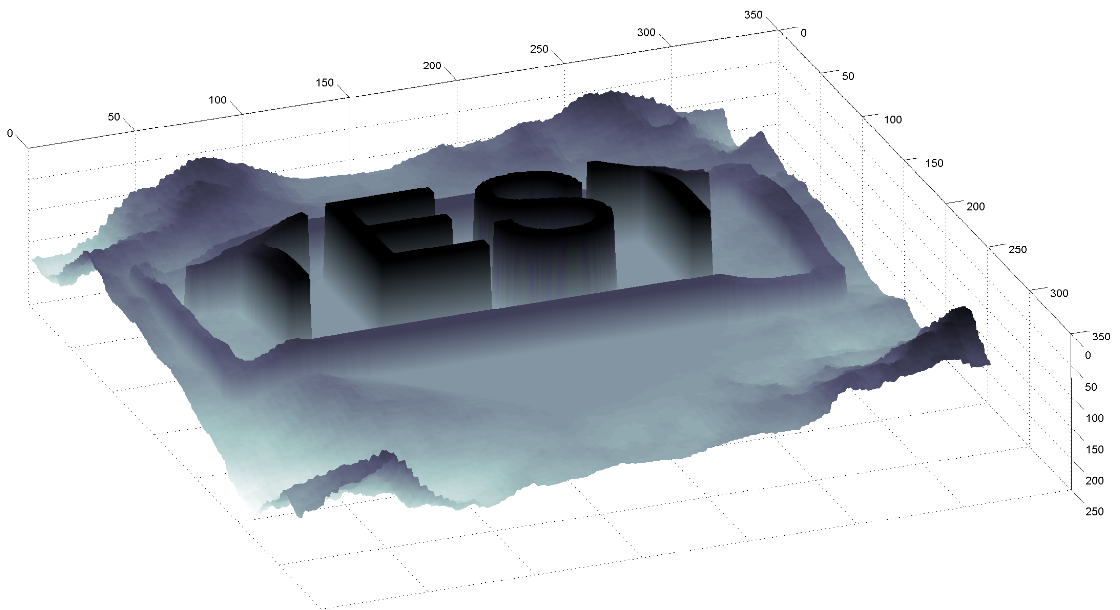


Figure 118: Design merged with noise, using the blend function with a central area totally dominated by the design. You can clearly see that the E and S are not influenced by the noise function, but are entirely defined by the design

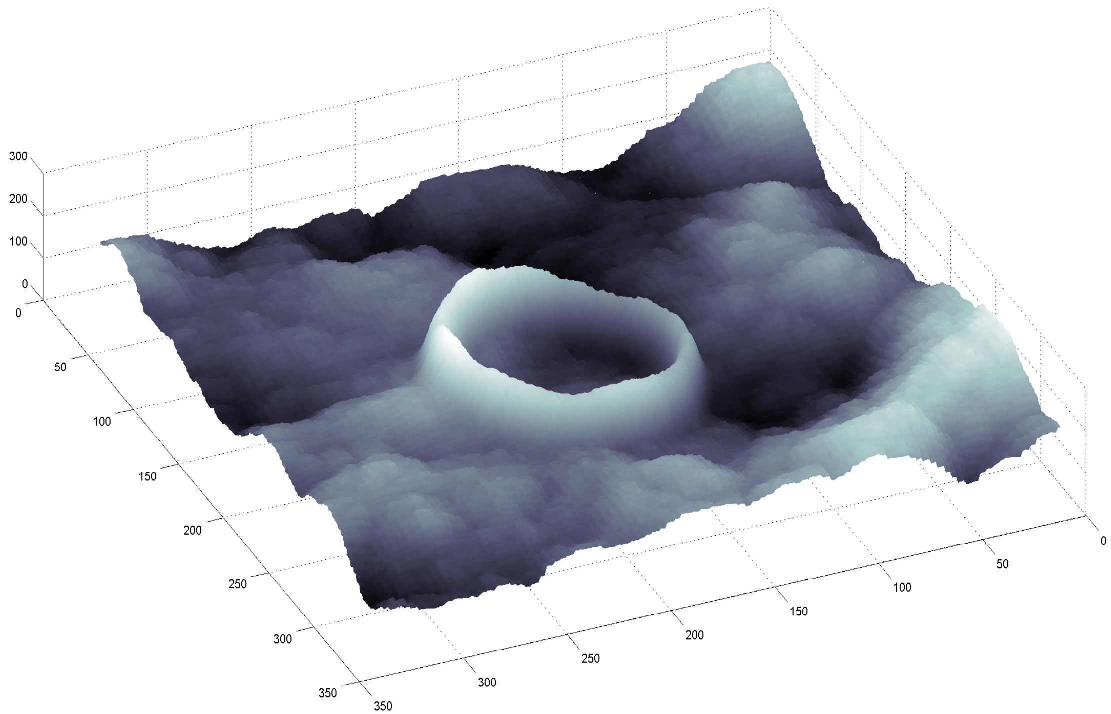


Figure 119: A pre-designed ”crater” was blended with a noise function. The crater was a wide stroke circle which had been smoothed slightly to give it more rounded edges. The blend function was turned somewhat down, in order to not let the design become too dominant. A somewhat realistically looking noisy crater is clearly visible, and it blends perfectly with the landscape. The designed crater model was very simplistic which may have resulted in too tall sides.

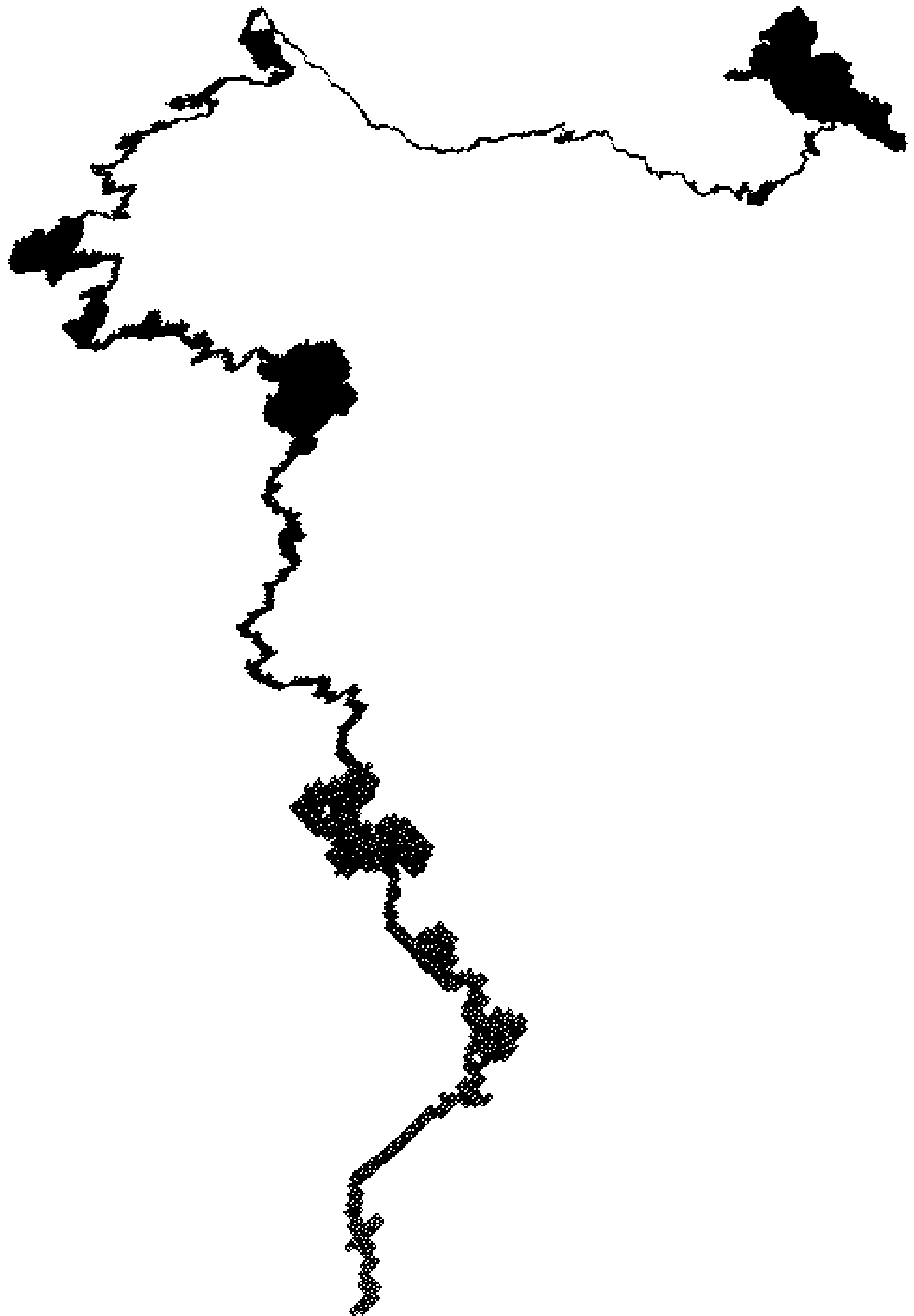


Figure 122: The calculated path of a river in a 3D landscape, showing both the winding flow and river basins filling up

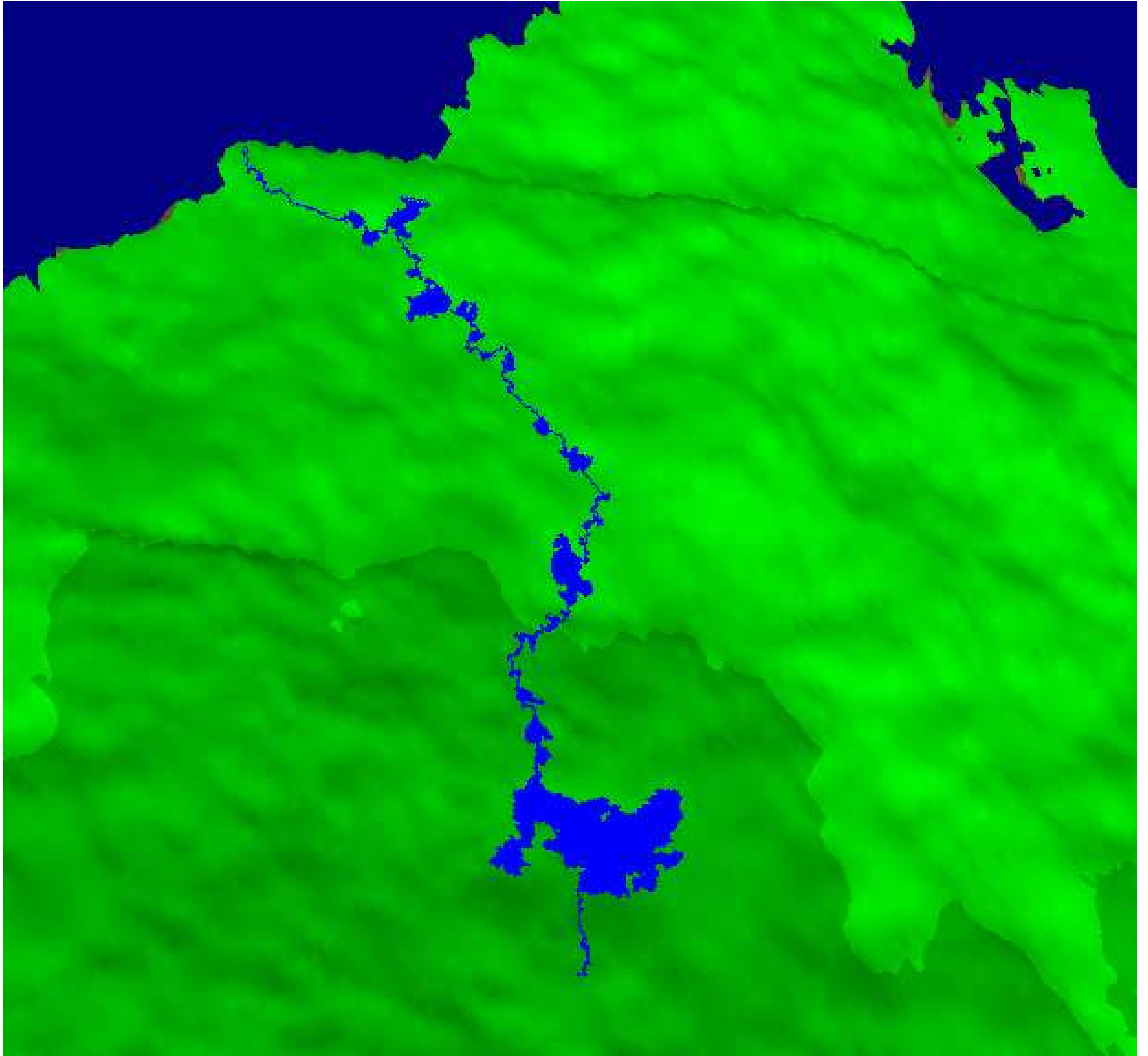


Figure 123: A river flows through the landscape, forms a single lake and then moves on till it reaches the ocean

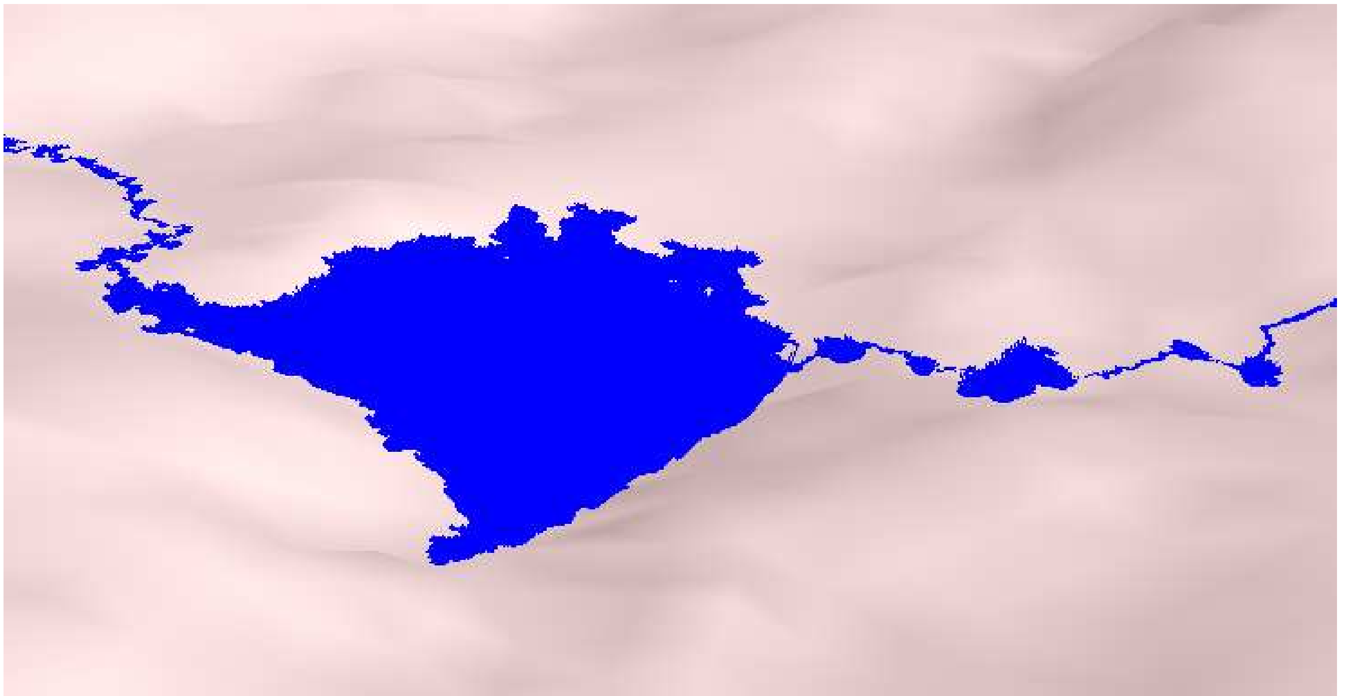


Figure 124: A lake with both an inlet and an outlet

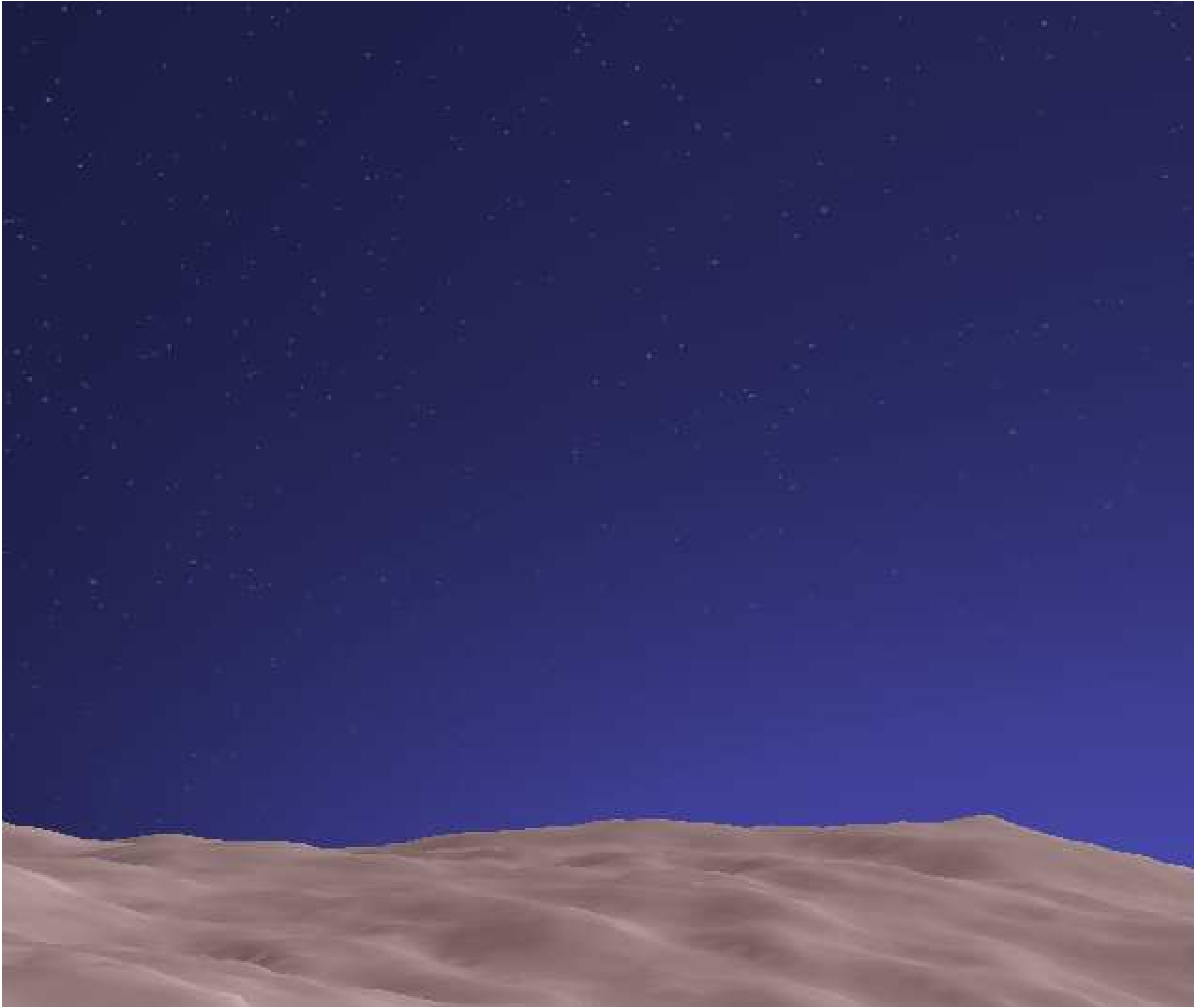


Figure 125: Looking towards the day break. The sky is starting to become more bright and blue in the lower right corner, but the stars are still slightly visible, more so when looking away from the light

Figure 126: The sun is back on the sky, and we can no longer see the stars. Only the clear blue sky is visible

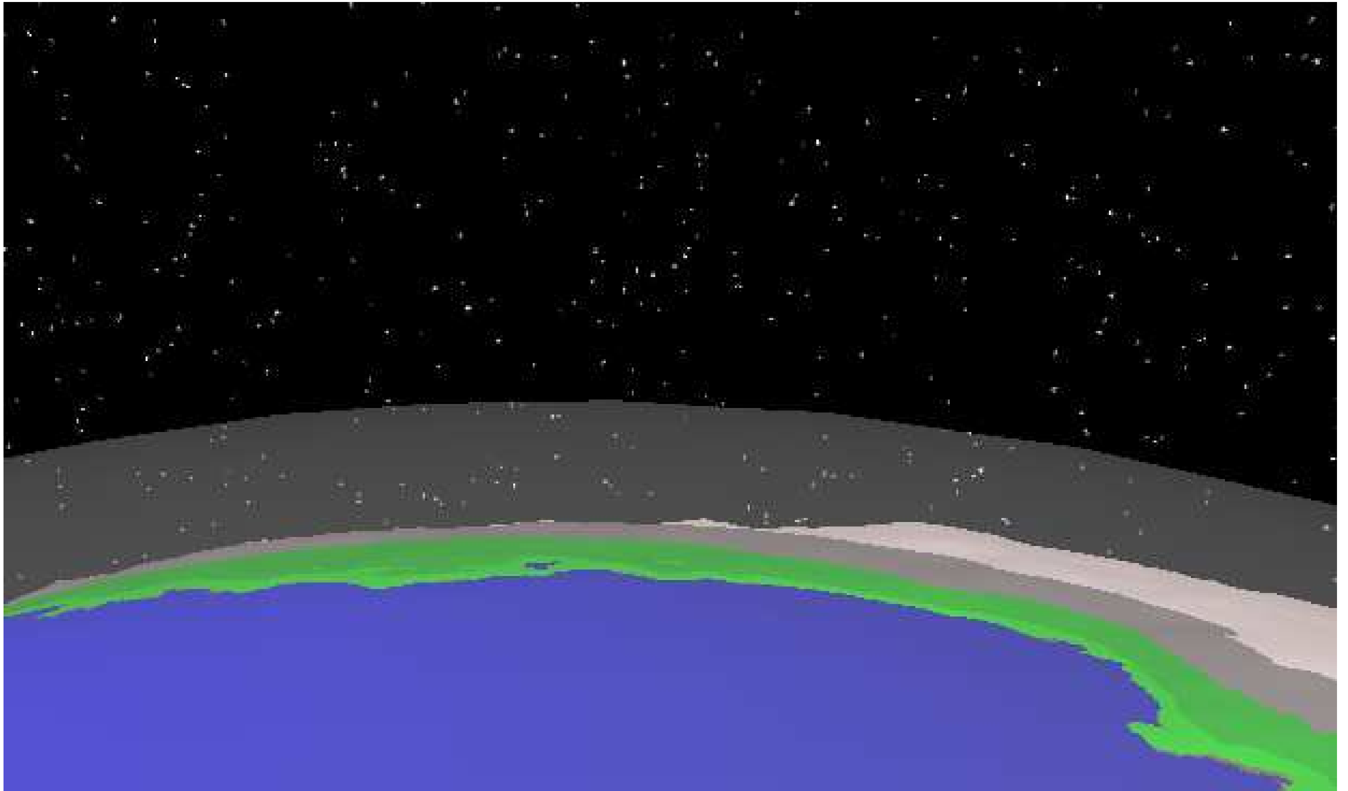


Figure 127: From a position high above the atmosphere, it looks like a semi transparent layer of air. This is not very realistic, but it shows that the atmosphere can easily look very different depending on the observers position

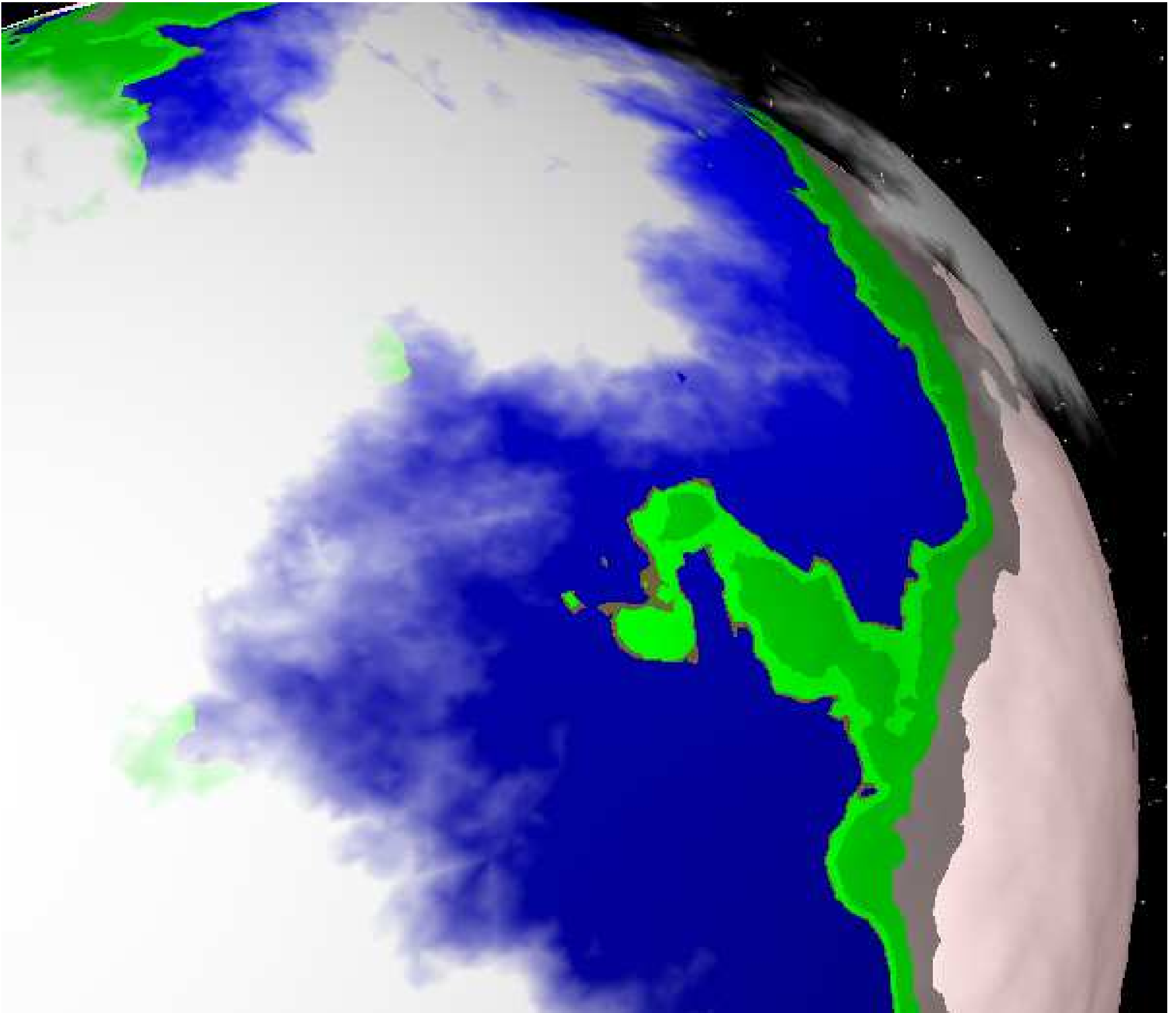


Figure 128: The atmosphere is colored white, but the alpha component is defined by a noise function to emulate cloud cover.

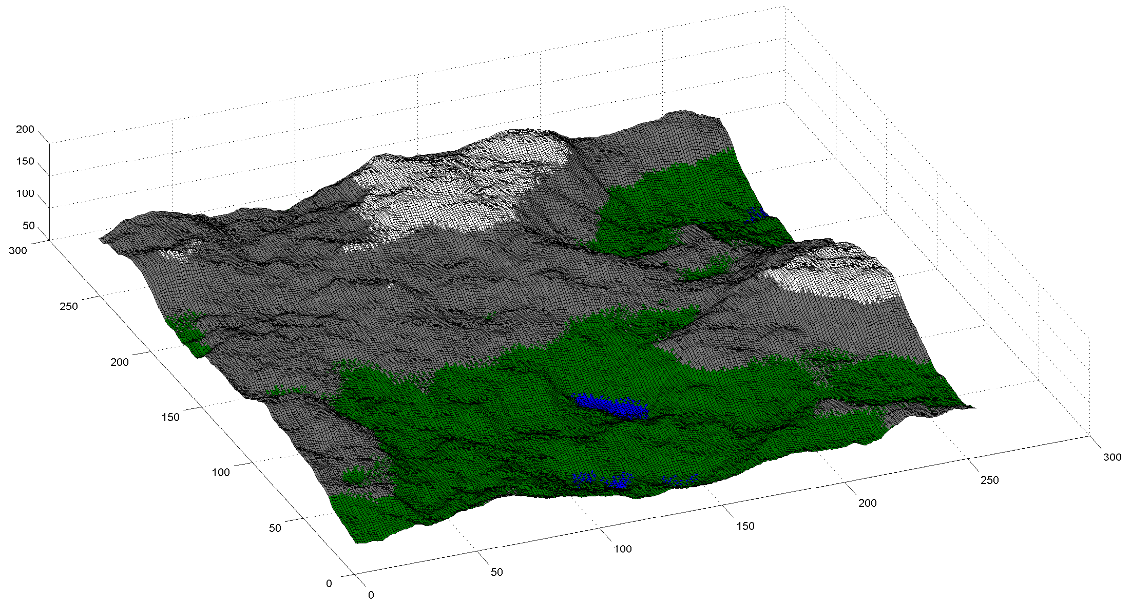


Figure 129: A terrain is colored according to its altitude. The colors are noticeably arranged in bands where the border between two colors are at one specific altitude.

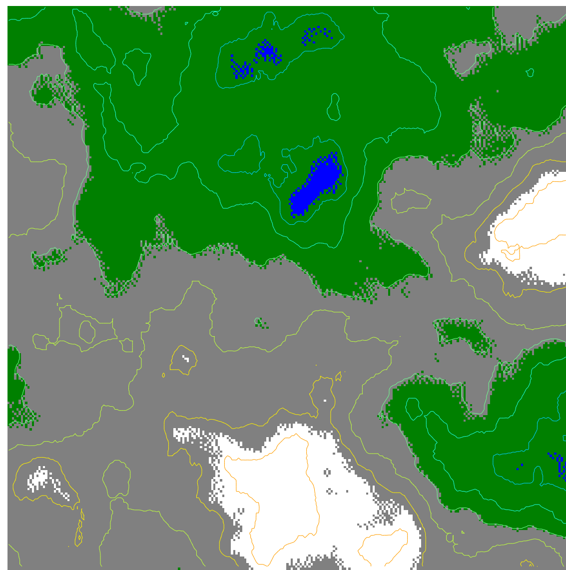


Figure 130: A terrain is colored according to its altitude and it is clear that the borders between colors follow the contour lines.

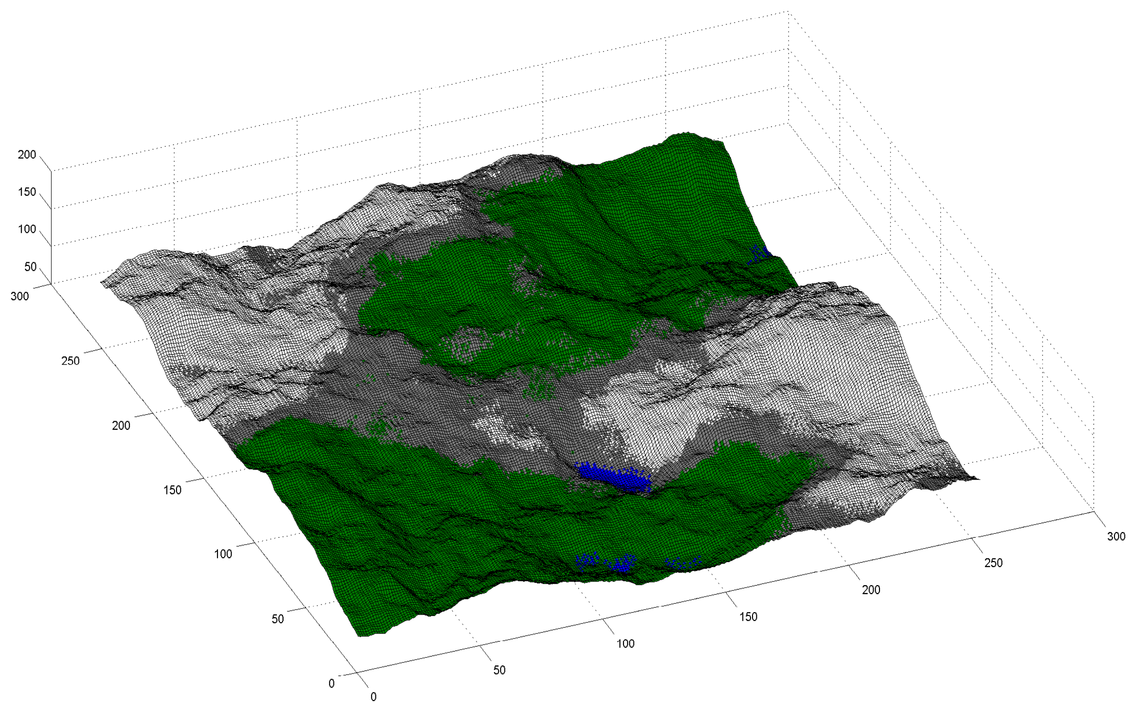


Figure 131: A terrain is colored according to its altitude and a noise texture which perturbs the color selection, so the border between colors no longer occur at specific altitudes, though snow is still dominant at high altitudes, water is below a certain altitude, and grass is dominant at the lower areas.

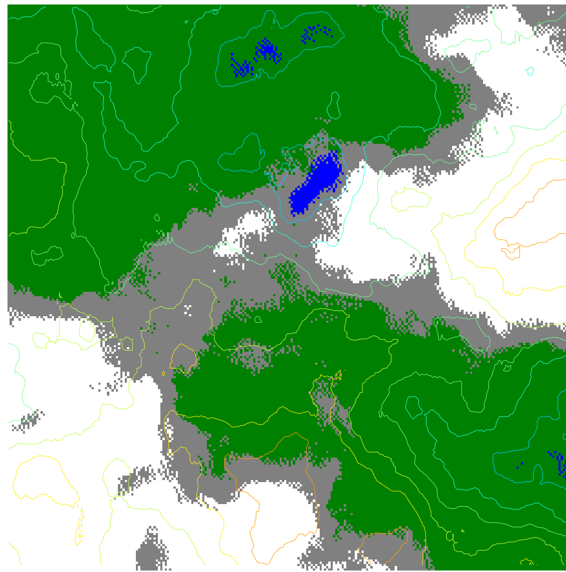


Figure 132: A terrain is colored according to its altitude and an extra noise function which pertubs the altitudes. It is now clear that the borders between colors do not entirely follow the contour lines.

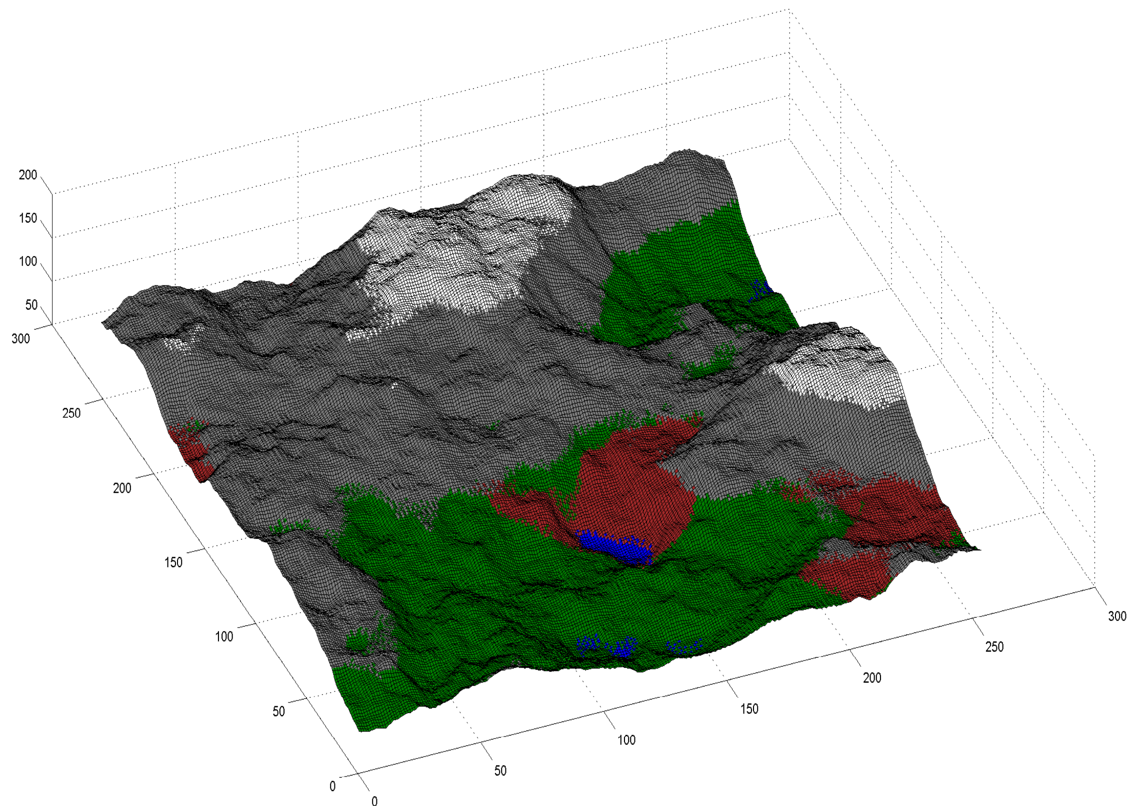


Figure 133: One noise function defined the altitude of the terrain and another noise influenced certain terrain characteristics in order to create irregularly shaped muddy areas in the lowland.

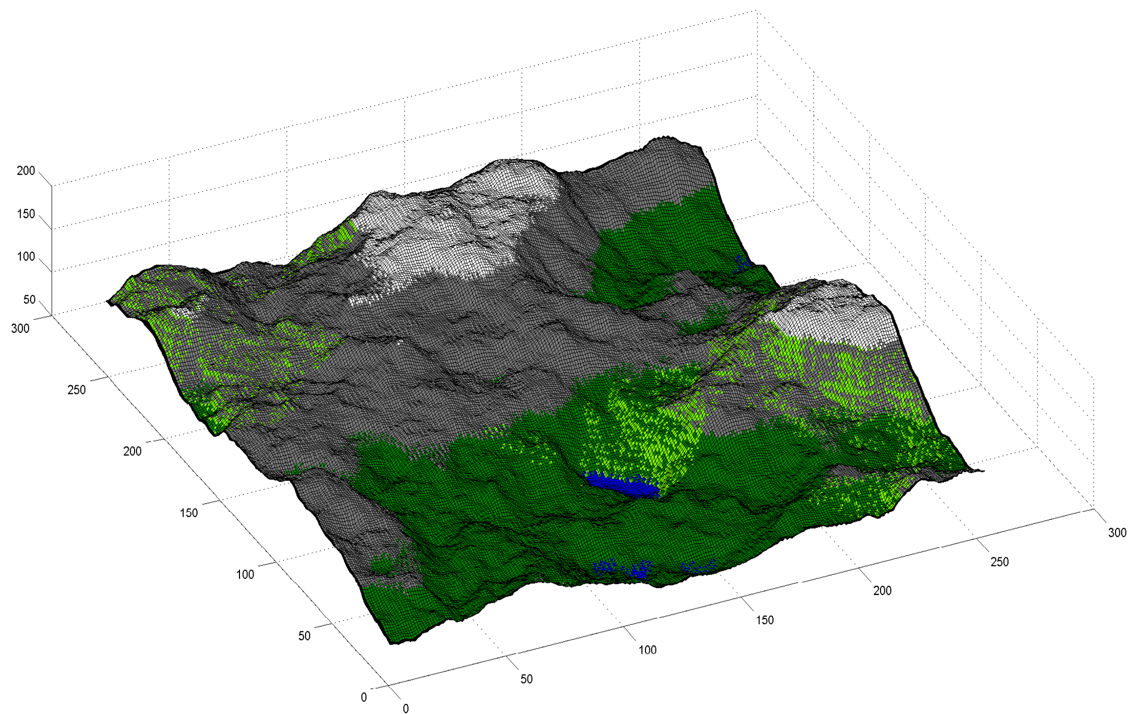


Figure 134: A terrain is colored according to altitude and its gradient along with another noise function. On slopes facing ”left” there will sometimes be patches of light green grass, when the noise function also allows it.

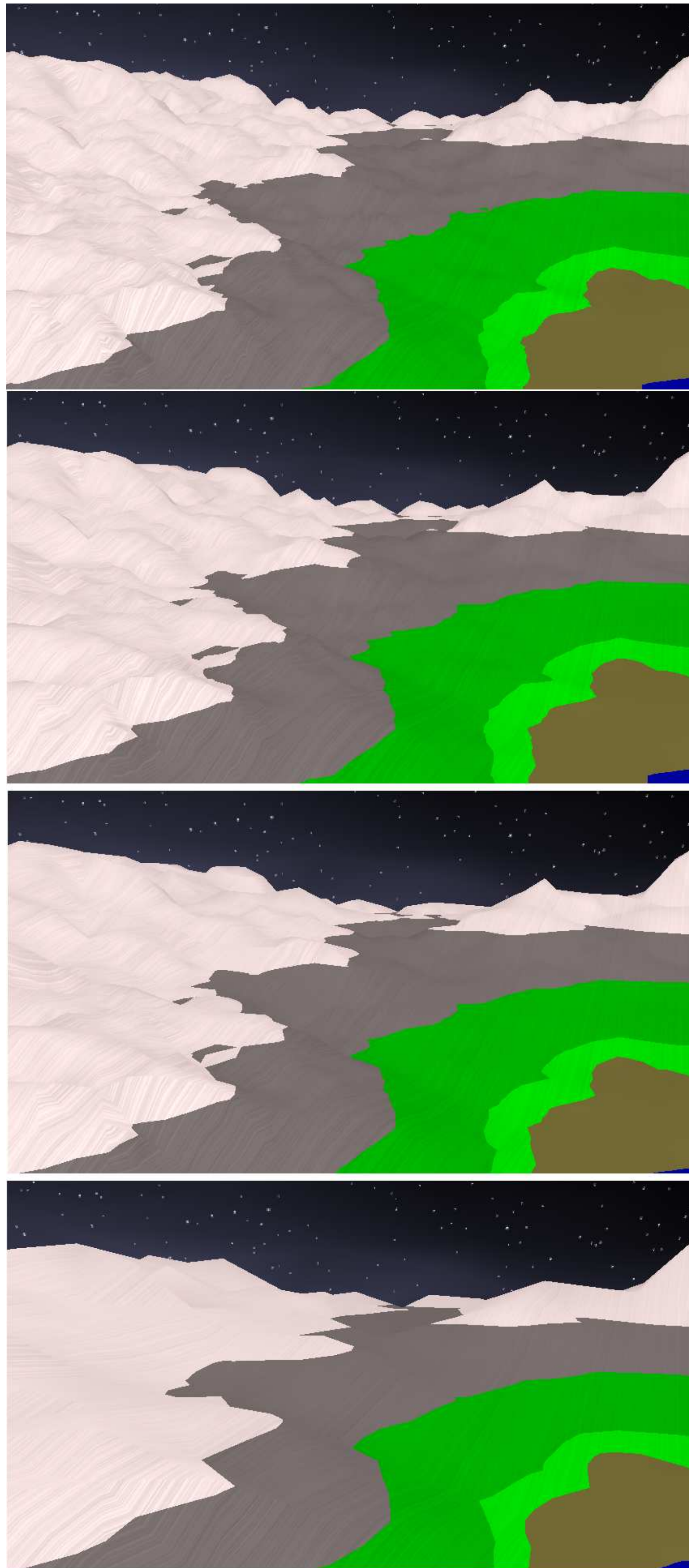


Figure 135: Four identical views with different number of triangles. From top to bottom, the triangle count is 100 000, 50 000, 25 000 and 5 000

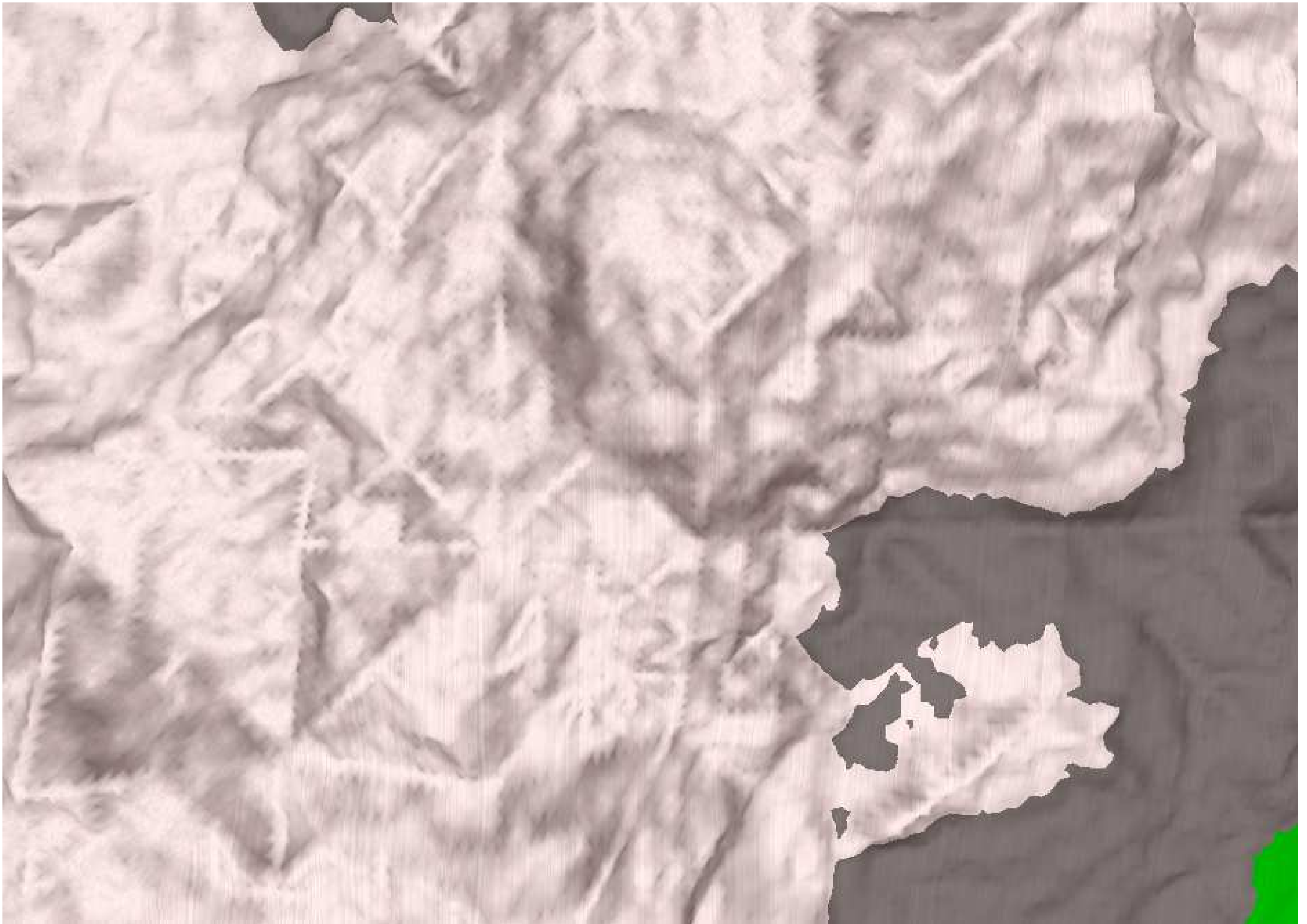


Figure 136: A terrain with visible ridges from early midpoint displacements

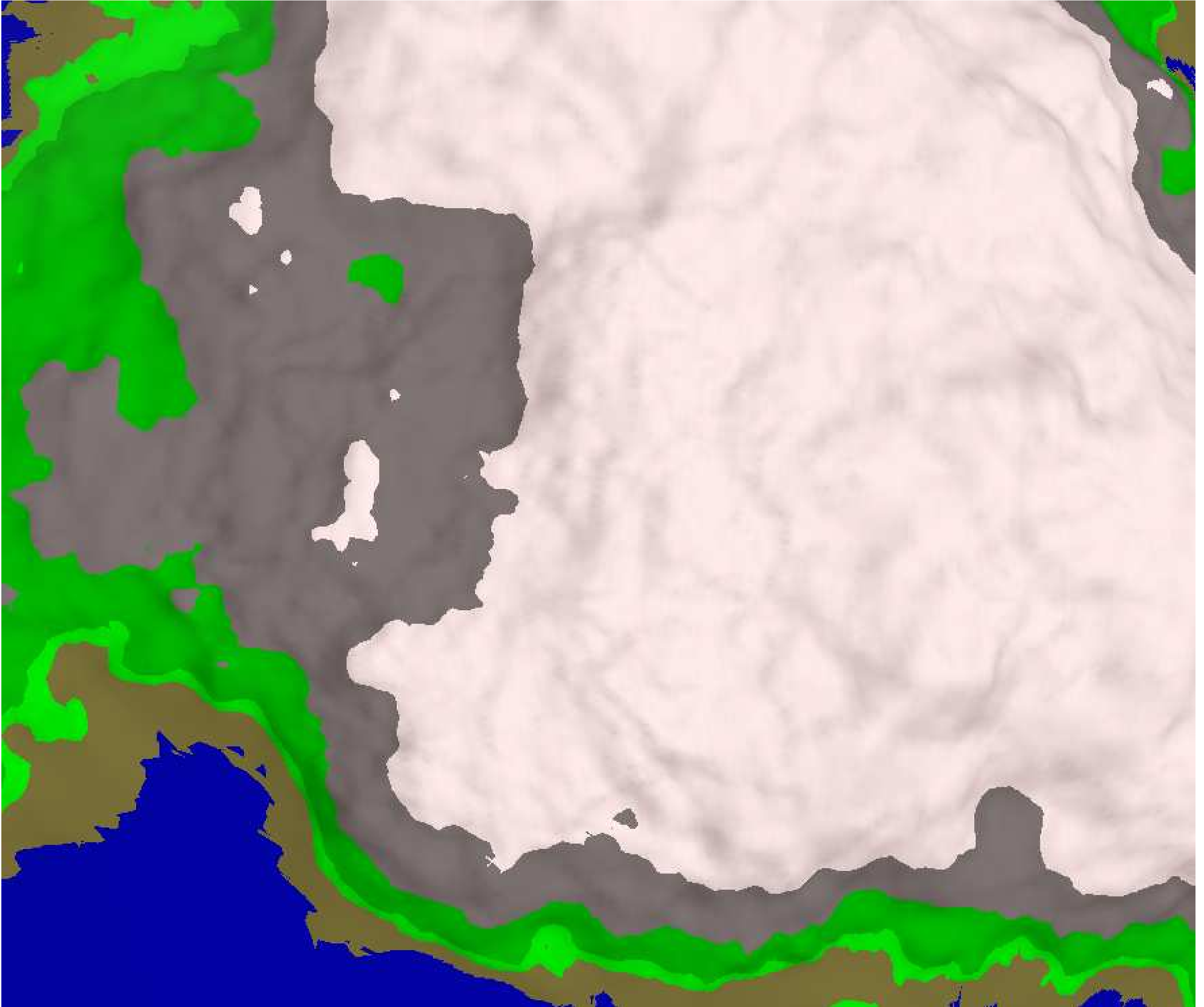


Figure 137: A terrain generated with Perlin noise and not showing any vissible ridges

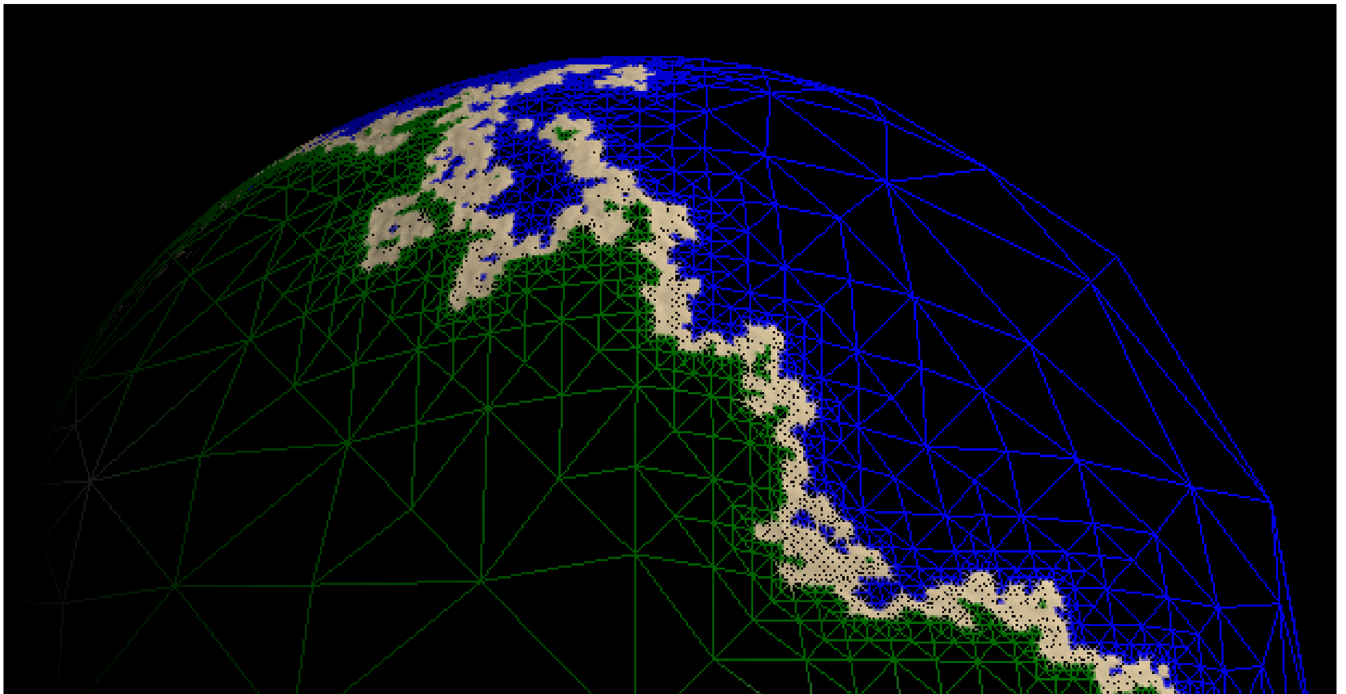


Figure 138: The triangles spanning the coast line have high priority and are subdivided more than triangles which are entirely on land or entirely in the ocean