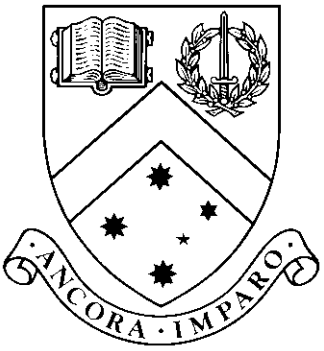
Deformable Terrain Generation for Real-Time Simulation

by

Peter Sbarski, BCompSc
psba1@student.csse.monash.edu.au



Literature Review

Bachelor of Computer Science with Honours (1608)

Supervisors: Jon McCormack & Alan Dorin

2nd Reader: Peter Tischer

School of Computer Science and Software Engineering
Monash University

September, 2004

Deformable Terrain Generation for Real-Time Simulation

Peter Sbarski, BCompSc
psba1@student.csse.monash.edu.au
Monash University, 2004

Abstract

This literature review will present and analyze a number of popular algorithms pertaining to the creation and management of computer generated terrains. This report will outline and discuss popular techniques for the generation of terrain as well as advanced Level-Of-Detail algorithms.

Further to this, a review of basic meteorological processes, such as rain and wind, will be presented. These processes, as well as erosion, will be described in relation to the topography of the terrain.

Contents

List of Figures

Chapter 1

Introduction

The Cambridge dictionary [12] dryly defines terrain as being "an area of land". In truth the terrain is much more than that, it encompasses tall mountains, deep canyons, expansive plains and smooth rolling hills.

Creating realistic artificial terrain can be a daunting task due to two primary problems. The first problem is related to the creation of realistic terrain itself. The terrain (for example see image 2.1) we see around us today has been sculpted by nature over millions of years. Geomorphological processes such as erosion has helped it to create tall and rocky mountains and lustrous plains. Algorithms used for the generation of terrain must therefore take into the account its smoothness and grandiosity. The terrain must look realistic and not be too random or too rough. Section 2.1 deals with the generation of smooth and realistic terrain.

The second problem is performance. A detailed and expansive terrain may look good but perform very poorly in a real-time setting. Section 2.2. deals with advances in this respect.

Section 3 describes various meteorological events and properties such as wind, temperature, evaporation, condensation, precipitation, etc... All of these are, directly or indirectly, influenced by the form and the composition of the terrain. In return, these processes cause erosion which affects the terrain. The overall aim of this literature review is to

Figure 1.1: An example of terrain [13].

present and evaluate terrain generation techniques, terrain Level-Of-Detail algorithms and

to describe natural process which could be used in a simulation to create and continuously modify an existing world.

Chapter 2

Terrain Generation and Management

2.1 Terrain Generation

The generation of algorithmic, as opposed to manual, artificial terrains is a relatively new subset in the field of Computer Graphics. Some current algorithms used for terrain generation have been known for many of years (e.g. Fourier and Poisson filtering) [7]. Other algorithms are not so old, for instance, the summation of band-limited noises algorithm (i.e. noise/turbulence algorithm) only dates back to 1985 [22]. It was originally created for generating efficient and naturalistic textures and only later was it adapted and used for terrain development. Other techniques and algorithms, such as the mid-point displacement algorithm (i.e. fractal algorithms), the fault algorithm, the circles algorithms, and the particle deposition algorithm are even newer [3][16].

There are three main methods used for the creation of terrains: 1) freehand manual modeling and specification of all terrain features and properties, 2) semi-manual modeling of general terrains features and properties and 3) automatic generation based on a number of different algorithms with some limited user-based input.

The first method involves the user explicitly specifying the positions and heights of all vertices making the up the terrain data. Whilst this method gives the user the maximum control over the end result, it is also extremely tedious and long and it makes it hard, for the user, to keep the "big" picture in mind. This method can be done by manually filling out a heightmap or manually perturbing all vertices in a graphics/modeling application.

The second method is very common among most non-professional users. It involves the user approximately specifying the features and the dimensions of terrain by hand using various drawing utilities. The user can interactively specify the overall dimensions of the terrain and draw its various features, such as mountains and canyons, by clicking on the height-map representing the terrain or the terrain itself. Applications such as Geoscape 3D [10] (see figure 2.1) and Leveller [15] work in this way.

The third method allows the user to specify only some details about the terrain, such as the overall size, the level of realism, smoothing, amount of features, etc? The rest is determined by the algorithm used. Certainly, in this method, the user only has very limited control, but unless the user has a specific need to specify the positions of various topographical features and specify some other finer details, this method presents the fastest way of creating terrain. Applications such as TerraGen [27] work on this principle.

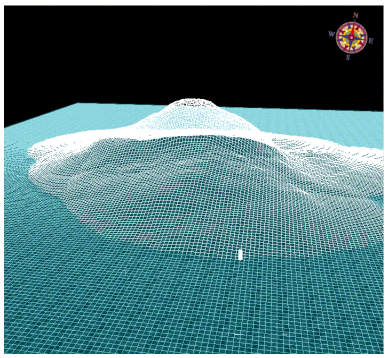


Figure 2.1: GeoScape3D in Action.

2.1.1 Perlin Noise and Turbulence

The third method, which is the domain of the current research into terrain generation, already has a number of established algorithms. Perlin Noise, or noise synthesis [22][7] (pg. 69,73) as it is also known by, is one such algorithm. It works by summing a number of "band-limited frequencies" with randomly varying amplitudes. Ken Perlin, in one of his papers for SIGGRAPH in 1985, called An Image Synthesizer, described a scalar function called noise which produced texture modeling primitives with stochastic properties by taking a three dimensional vector as an argument (the vector may have 1, 2, 3 or more dimensions). This function works in the following way. First, a number of points together with their x, y and z coordinates [x,y,z] (a set of these points is called an integer lattice) [22][7] (pg. 69) are specified. All points in the integer lattice set are assigned a pseudo-random values (a pseudo-random number generator always generates the same output if the input given to it is also the same) and also their own x, y and z gradient values [a,b,c] and value d at[x,y,z] via the noise function [22]. Every point is determined, procedurally, independently of any other point. For example, the following is an example of a very basic pseudo-random noise function; note that it will always produce output between 0 and 1 [8].

$function\ noise(x)$
$x = (x << 13)^x$
$return\ (1.0 - ((x * (x^2 * 15731 + 789221) + 1376312589)\&7fffffff)/1073741824.0)$
$end\ function$

Therefore, if [x,y,z] is on an integer lattice the the noise function is defined as noise([x,y,z]) = d[x,y,z]. Otherwise, a smooth interpolation between "lattice equation coefficients" is first defined. The paper describes a form of noise which is known to occur in natural processes. It is the 1/f signal and it can be described via the following formula (all this does is loop over octaves of noise) [22]: $sum[noise(point * 2^i)]/2^i$

The above formula will create a weighted sum of the output produced by the noise function. It can be used for a number of different octaves. The denominator in the equation can be changed to specify the roughness of the result (a lower value will make the end result more rough, while a higher value will make it less rough). By varying the numerator and the denominator it is thus possible to produce noise of various frequencies.

The noise function is referred to as band-limited because "there is no detail outside of a certain range of size". Another important finding described in, An Image Synthesizer, was the turbulence function [22][8]. The turbulence function simulates the appearance of layers and provides an appearance of a turbulent flow. It takes a set of noisy functions at

different scales and sums them together to produce a relatively complex result, which can also look very smooth. The paper gives the following example of a turbulence function:

```
function turbulence(p)
        t = 0
        scale = 1
        while (scale is less than pixelsize)
                t += abs(noise(p/scale)*scale)
                scale /= 2
        return t
end function
```

The paper specifically points out the *noise(p/scale)*scale* line which specifies that the "amount of noise() added is proportional to its size". The abs() function gives the impression of a "discontinuous flow". Overall, these are the steps the full Perlin Noise function (with turbulence) usually takes:

- Generate a number of sets of noisy values with varying frequencies and amplitudes.
- Using the turbulence function assemble and scale these sets together to produce a smooth end result (see figure 2.2).

As was mentioned before, Perlin's paper An Image Synthesizer proposed these algorithms largely to create a system for the generation of procedural textures [22]. Now, however, they are also used to produce heightmaps which are in turn used to specify terrains. As the turbulence function is able to create smooth patterns it can be used quite successfully in generation of terrain data.

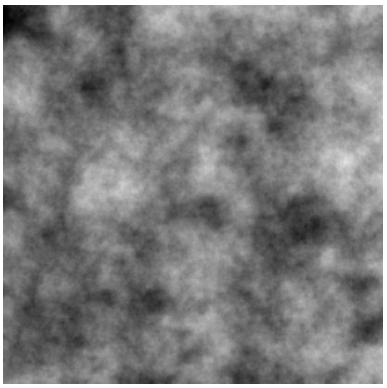


Figure 2.2: A smooth "turbulent" texture created with Perlin Noise.

2.1.2 Mid-Point Displacement Algorithm

Random midpoint displacement algorithm [3][16], a fractal technique, is another popular algorithm. It involves running through a heightmap a number of times and perturbing its various vertices. Diamond-Square is a type of midpoint displacement algorithm which works in the following way. The heightmap is represented as a grid wherein each vertex can be assigned a specific height. The algorithm works by averaging four corner values first in a square and then in a diamond (see figure 2.3).

Iteration 1:

- Square Step. At first the vertex in the middle of the heightmap is displaced by averaging the four corner values (e.g. Mid-Point (E) = A + B + C + D)/4). Where A is x0,y0, B is x1,y0, C is x0,y1 and D is x1,y1 where x0 and y0 represent the top left vertex and x1 and y1 represent the bottom right vertex. The middle point E is calculated as follows:
  E=x1/2, y1/2=(x0+x0+x1+x1)/4, (y0+y0+y1+y1)/4. To the middle point we also add a random value which ranges from -x to x where x represents the maximum allowable displacement for the current iteration (e.g. Mid-Point += rand_value(x)).

- Diamond Step. Select the four corner vertices. Each of these needs to be worked out. Vertex F, for example, is worked out as follows: F = (A + C + E + H)/4 + rand_value(x). The value for H, as we don't really know it yet, can be 0, randomly assigned or E. Otherwise F can be calculated with three vertices only: F = (A + C + E)/3+rand_value(x) This process needs to be repeated for all selected vertices.

Iteration 2:

- Square Step: Repeat the same process for smaller squares. For example, J = (A + G + F + E)/4 + rand_value(x). The x argument can be modified to account for the second iteration.

- Diamond Step: Again, repeat the same process as before.

This algorithm can be looped over a number of iterations, greatly modifying the terrain the longer it is run for. Note that this algorithm can and needs to have a number of extensions not specified before. For instance, a roughness value, in addition to the random displacement value, can come in very useful when generating terrains. This algorithm is certainly processor intensive but it produces good results.

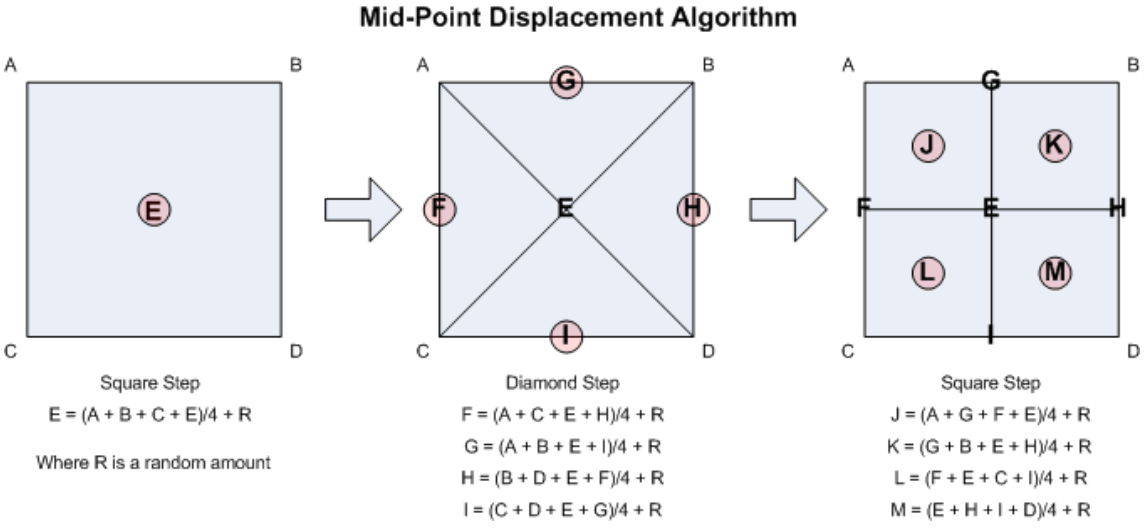


Figure 2.3: Steps the Mid-Point Displacement algorithm takes.

All fractal landscapes work in a similar way to the midpoint displacement algorithm. The terrain is split into a number of grids and points in each grid are displaced a certain value. After a number of iterations this methodology can produce an impressive fractal landscape.

2.1.3 Fault Algorithm

The fault algorithm [23](pg. 27) [16] is another fractal algorithm. The heightmap, which starts with all vertices having the same height, is divided into two random parts

which could be of different size. One part is selected and all vertices in it are uniformly increased, respectively the vertices in the other part are decreased. The terrain is subdivided into another two different parts and the same process occurs (see figure 2.4). After a number of iterations this process is stopped. The problem faced by this technique is that even after many iterations the heightmap could still be erratic, or in other words, look very "noisy". This problem can be solved in a number of steps:

- With each iteration the height by which the vertices go up and down could be decreased. The height can be worked out with this equation:
  *iHeight = iMaxDelta - ((iMaxDelta-iMinDelta)*iCurrentIteration)/iIterations*
  where iMaxDelta and iMinDelta represent the highest and the lowest allowable heights (e.g. iMaxDelta = 255 and iMinDelta = 0), iTerations represents the number of fault passes to be conducted and iCurrentIteration represents the number of the current pass.

- It is very likely that even after the following the first step, the terrain would still be too chaotic. If this is the case a blur or an erosion filter should be applied. The book [23] suggests using a Finite Impulse Respone (FIR) filter to blur and smooth out the heightmap. This filter loops through every vertex and modifies it based in part on the value of the vertex before it and the overall smoothing function. The general formula for this is:
  *ucpBand[j] = fFilter*fVertex + (1 - fFilter) * ucpBand[j]*
  where fFilter is the filter that affects bluring (0 = no blurring, 1 = highest possible blurring). fVertex is the "blurred" value of the previous vertex and ucpBand is the height band. This algorithm doesn't get applied to the whole heightmap at once but is rather done in bands.

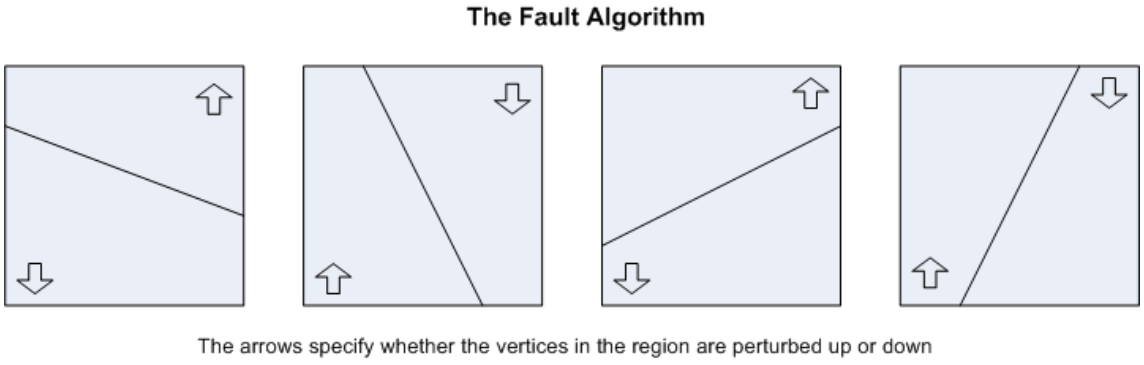


Figure 2.4: The fault algorithm in action.

2.1.4 Particle Deposition Algorithm

The particle deposition algorithm [16][3] can be imagined as dropping particles onto the terrain and through that effect building it up. In reality what the particle deposition algorithm does is vertically displace vertices according to a number of specified rules. One version of this algorithm involves the following steps:

- A random point on the terrain is found and is displaced by a certain amount.

- One of the four points adjacent to the current point is selected and is set as the new point or a random point within a certain radius of the previous point is selected.

- The new point is displaced and then another point adjacent to the current point is found. The process repeats.

A problem with the above approach is that it can create very noisy and rough terrains. One method that can be used to solve this problem is to let particles "roll down" if all adjacent points are too low.

2.2 Continuous Level Of Detail (CLOD)

Level of detail (LOD) algorithms are crucial for effective management of complex 3D scenes. A LOD algorithms main purpose is to calculate a dynamic polygonal mesh [5] which can processed and rendered quickly. A complex scene or an object can take a very long time to be rendered using a brute force approach. A LOD algorithm dynamically reduces (or increases) the amount of detail (e.g. triangles) that is present in the object so it could be rendered quicker. In some aspects Terrain LOD algorithms are more complicated than generic object LOD algorithms [9]. Terrain LOD algorithm must account for "continuous and very large models" that can be viewed "simultaneously very close and far away" [9]. Most LOD algorithms that thus have been created for rendering terrain have two major aims.

The first aim is to create a reasonable polygonal mesh approximation of the terrain which could then be rendered in real-time. The second aim is to preserve enough visual detail for the polygonal mesh to faithfully reproduce the original terrain specifications at reasonable detail.

There are a number of ways to go about achieving these aims. The most common method used by most LOD algorithms is dynamic polygon approximation based on the position of the camera and the view distance [23]. This method dynamically increases the amount of visual detail in the areas (called patches or blocks) close to the camera and decreases the amount of visual detail in the patches that are far away [23]. Therefore the blocks of the terrain that are close to the camera are rendered at the highest possible detail whilst the blocks that are further away are progressively rendered in less detail. Another method is simply concerned with adding more triangles to patches of terrain that are more complex and removing triangles from patches that are less so. In this method only a few triangles are used to approximate a smooth surface whilst a more chaotic patch uses more triangles. Different LOD algorithms use different methods to achieve their aims, e.g. the Quadtree LOD algorithm [24], created by Rottger, only uses the first method while Geomipmapping LOD algorithm [2], created by de Boer, uses both methods.

Unfortunately LOD algorithms have inherit problems such as "cracking" and "popping". Cracking occurs between adjacent blocks of different resolutions and popping occurs when a block has to switch from one detail level to another. It is up to the individual algorithm to solve these problems.

Finally, it should be mentioned that there are two types of LOD algorithms: discrete and continuous [6]. Discrete LOD (DLOD) algorithms use precomputed mesh models of various resolutions which switch depending on the view distance from the camera. The further the camera moves away the less detailed models are used by the system. Continuous LOD algorithms (CLOD) follow a similar idea except for one major difference, the lower and the higher resolution mesh models are calculated on the fly based on the conditions at hand. The advantage of using a Continuous LOD algorithm is that there is no need to pre-compute any models beforehand [9]. With terrain this would also be quite impossible.

The three most popular CLOD algorithms (Geomipmapping[2], Quadtree [24] and ROAM [5]) are outlined, discussed and evaluated next.

2.2.1 ROAM

The ROAM (Real-Time Optimally Adapting Meshes) algorithm [5] originally devised at the Los Alamos National Laboratory uses a Binary Triangle Tree (bintree) data structure and split/merge operations to dynamically modify triangle meshes. The Binary Triangle Tree data structure starts with a root triangle which is a right-isosceles triangle at the coarsest level of subdivision (l = 0). To achieve the first level of tessellation the root triangle needs to be split into two triangles. To properly do this a straight line needs to be drawn from any of the triangle's three vertices to bisect the line opposite the vertex [23].

Two right square triangles should be formed. This process can be repeated (see figure 2.5) and the original root triangle tessellated into more triangles until the resolution of the underlying heightmap is reached. A mesh can be formed by "assigning world-space

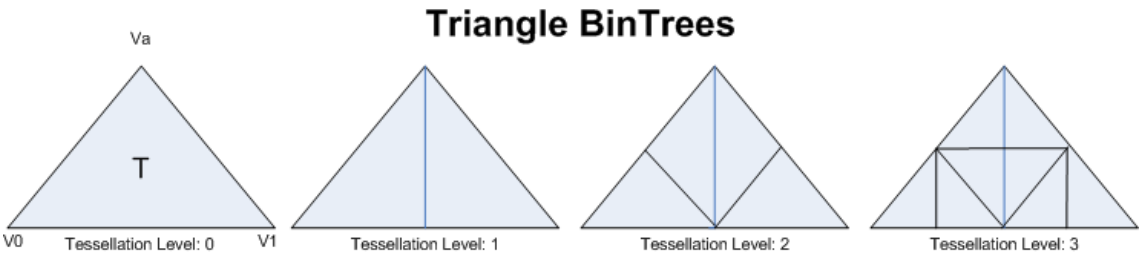


Figure 2.5: A BinTree at different levels of tessellation.

positions w(v) to each bintree vertex. A set of bintree triangles forms a continuous mesh when any two triangles either overlap nowhere, at a common vertex, or at a common edge.". These continuous meshes are called "bintree triangulations" [5][23]. Most triangles (e.g. v0, v1, v2) in a mesh have three neighbors: the bottom base neighbor which shares the base edge (v0, v1), the left neighbor which shares the left edge (v0, v2) and the right neighbor which shares the right edge (v1,v2). The triangles at the edges only have two neighboring triangles. It follows from the algorithm that for a particular triangle with the coarseness level of l, all its neighbors must either have the same coarseness level or have the next finer level (l + 1) or the next coarser level (l - 1). Assuming a triangle T and its base triangle Tb, when they are at the same level of coarseness they form what is called a diamond. Two operations, split and merge (see figure 2.6), can be carried out on a diamond.

The split operation introduces a new vertex in each triangle thus improving the dia-

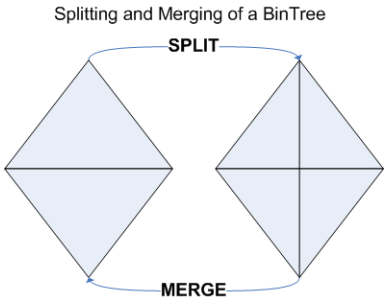


Figure 2.6: Split and Merge Operations

monds overall detail level. Of course, depending on the current level of tessellation more than one vertex could be introduced during the split operation. If T doesn't have a base neighbor then only it is split. A merge operation removes extra triangles put in by the split operation.

If T and Tb originally are at two different levels, e.g. Tb is coarser than T, T can not be split until Tb is split. If this isn't done T-junctions could appear [5]. The paper states that the split/merge approach outlined above doesn't require any special efforts to handle cracks and T-junctions apart from what was mentioned above. The paper offers an example of a greedy algorithm to drive split and merge operations. The idea behind this algorithm is to "keep priorities for every triangle in the triangulation, starting with the base triangulation, and repeatedly do a forced split of the highest-priority triangle" [5]. The algorithm attempts to "create a sequence of triangulations that minimize the maximum priority at every step." [5]. Although, it must be ensured a "child's priority is not larger than its parent's." [5].

Finally this algorithm suggests that by using another priority queue it could be possible to efficiently perform split and merge operations by adding/subtracting detail to/from the mesh from the previous frame rather than starting from scratch every time, thus taking the advantage of "frame-to-frame" coherence. The priority can generally be calculate from: priority = (variance of terrain) / distance [11]. Where the variance of terrain is the measure of its complexity - flat surfaces have a lower complexity variance and rough surfaces have a higher complexity variance. The distance is calculated from the point of view (POV) to the target triangle. The split queue can be implemented as follows as was showed in the paper [5]:

function split()
Let T = the base triangulation
For all T in T, insert T into Qs
While T is too small or inaccurate
    Identify highest-priority T in Qs
    Force-split T
    Update split queue as follows
        Remove T and other split triangles from Qs
        Add any new triangles in T to Qs
end function

This algorithm produces optimal triangulations. The paper [5] has this example, consider a triangulation T' that has a lower priority that T. T' must contain only the descendants of the triangles that were forced to be split while creating T. Because of the way the algorithm works the force-split operation makes necessary refinements (splits) to preserve continuity. Therefore, T' does not and can not contain any ancestors to the triangles in T. T' has a lower priority and therefore it must only contain descendants of triangles in T. The Merge Queue requires that the algorithm keep a second priority queue, Qm, to contain all "mergeable diamonds for the current triangulation". The greedy algorithm for merging, as specified in the original paper, is as follows[5]:

function merge()
If f = 0
    Let T = the base triangulation
    Clear Qs, Qm
    Compute priorities for T's triangles and diamonds, then insert into Qs and Qm respectively
Else
    Continue processing T = Tf-1
    Update priorities for all elements of Qs, Qm.
End if
end function

Where f is a frame. In other words, the split/merge operation can be represented as follows in the pseudocode [11]:

function split/merge()
While terrain triangulation is not of target size or accuracy do
    If triangulation too large then
        Merge lowest priority triangles in merge queue
    Else
        Split highest priority triangles in split queue
    End if
end function

Note that if the overall terrain triangulation is reset each frame then only split operations are needed. ROAM is a greedy algorithm that builds optimal meshes with the least number of split/merge operations.

2.2.2 Geomipmapping

The analysis of this algorithm is mainly based on a paper produced by Willem H. de Boer called, "Fast Terrain Rendering Using Geometrical MipMapping" [2][23](pg. 76). This algorithm assumes a 2D heightmap based terrain representation with exactly $2^n + 1$ horizontal and $2^n + 1$ vertical number of vertices. This allows for 2^n quadrilaterals with each quadrilateral consisting of two triangles. The heightmap must be separated into a series of identical patches (or what the paper calls "blocks" or "GeoMipMaps" [2]) that must have a fixed size of $2^n + 1$. The block size can be chosen arbitrarily by the programmer. The author of paper recommends selecting a 17x17 block for a terrain grid of 257x257. If a block is rendered with all its triangles intact then it provides the highest level of visual detail (see figure 2.7) albeit with a performance penalty. This amount of detail is referred to as LOD of 0 (i.e, maximum amount of detail - default terrain block) or GeoMipMap level 0 [2].

To increase rendering speed the algorithm uses lower resolution block approxima-

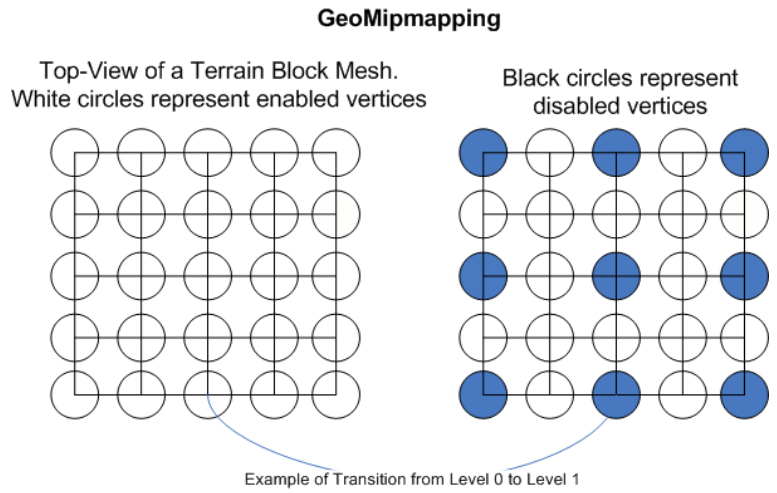


Figure 2.7: GeoMipMap Mesh view.

tions if the block is far away from the camera and higher resolution block approximations if the block is close to the camera. Similarly to texture mipmapping techniques when the camera is a certain distance away from the patch a desired number of triangles are chosen to be rendered. When the camera is close all triangles are rendered and when the camera is far away only the few predominant triangles are sent to the pipeline [2]. The amount

of triangles to be rendered in the patch are selected based on the GeoMipMap level which itself is based on distance (d) from the camera. According to the author of the paper selecting a fixed distance results in popping. Popping is a visual side-effect which occurs when the visual detail of a terrain patch suddenly changes (see figure 2.8). It usually happens when a lower detailed version of a patch is replaced by a more detailed version and vice-versa [23][6](pg. 516). As the GeoMipMap level changes the removal (or addition) of vertices results in a decrease (or increase) in detail and subsequently the change in height. The change in height (h) is less noticeable if the distance is great due to the effects of the perspective.

The paper suggests that a special value (m) be used to specify the value of h in screen space pixels. A special threshold value (t) would also need be used to specify the error rate, i.e. when the change in height is too noticeable. From this follows that the GeoMipMap level should only change when the value of m is smaller than the value of t. Because a change in GeoMipMap level could force a number of heights to change the worst case scenario needs to be looked at. If the worst case height change is within acceptable limits then all height changes are within it too. Next, to find out whether the GeoMipMap level should change for a given d, the algorithm must compare the value of m, which is given by the GeoMipMap level, to t. Each GeoMipMap level has an associated m. If for a selected GeoMipMap level m is more than t then a switch should not go ahead and another GeoMipMap level tried. The value of m, for each GeoMipMap level, is calculated dynamically depending on the slope of the camera's view relative to the patch and the distance d. To reduce CPU usage the author recommends a strategy to precalculate some values of m in advance. This can be done if the camera's direction vector is considered to be "permanently horizontal" [2]. The author admits that this method will not deliver precise results especially when the camera's view vector relative to the GeoMipMap is high. Another way to speedup calculations is to precalculate the minimum value of d (D) at which a particular level (Dn) could be used. The paper offers a small psuedo-code example which shows how an appropriate GeoMipMap level could be selected. This pseudo-code is reproduced faithfully below [2]:

Function GeoMipMapLevel()
For Each GeoMipMap Level N
    Compare L to Dn
    If (L is more than Dn) store to RESULT
End For
Return RESULT
End Function

Where L is the distance from the camera to the center of the GeoMipMap in 3d, and D is Dn. To pre-calculate Dn for each level the following steps could be taken. As shown in the paper[2]:

- $Dn = h * C$,
- $C = A / T$,
- $A = n / t$,
- $T = 2 * t / Vres$

Where n is the near clipping plane, t is the top coordinate of the near clipping plane and Vres is the vertical resolution in pixels. The paper also covers whats called Trillinear GeoMipMapping (or what some call Geomorphing [23][5]) which greatly reduces or even eliminates popping. With the previous method popping can still occur if the value of t is

set too high. The idea here is to adopt something like Trilinear Filtering which blends different mipmaps together. By using a similar approach it is possible to morph two patches of different visual detail together to reduce popping.

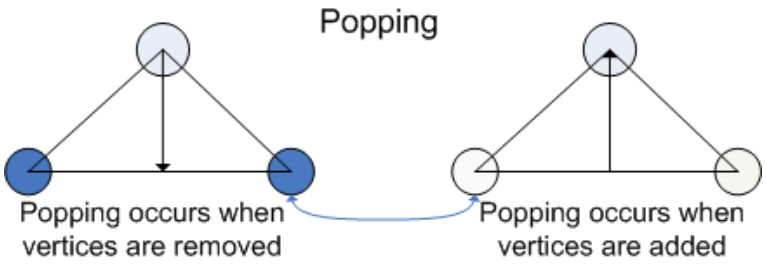


Figure 2.8: The popping that occurs with the removal or addition of vertices.

The second problem faced by this algorithm are the cracks, or geometry gaps, between patches of terrain. Cracks at the edges of two different patches usually occur when they have a different level of detail [23]. There are a number of ways to solve this problem. One way is, as suggested by the author, is to "add extra vertices at the edge of the lower detail level of the two neighbouring GeoMipMaps so that it will fit with the higher detail GeoMipMap's edge". Unfortunately this method is too memory costly and slow. Another solution suggested by the author is to omit extra vertices in the more detailed patch [2][23]. This, alas, lead to T-junctions which have a "missing pixels" effect. To solve this problem some extra vertices which reside on the boundary sometimes need to be omitted when rendering the patch.

The paper also offers a solution if the algorithm takes up too much RAM called "Basic Progressive GeoMipMap streaming" [2]. Here the author argues that in certain applications the algorithm could take up too much primary memory but that there is also a way to get around it by precomputing certain data which can be used to produce terrain blocks on the fly. The steps to do this are quoted in their entirety next:
Generate an initial QuadTree and then for every terrain block do the following [2]:

- Load Terrain Block
- Calculate terrain block's bounding box and store it in appropriate leaf.
- Calculate all GeoMipMap levels for this terrain block.
- Calculate Dn for every GeoMipMap Level and store it in the terrain block's quadtree leaf.
- Erase all GeoMipMap levels and proceed step 1 to 5 for the next terrain block.

On the completion of this method there should be a QuadTree in which every leaf node should contain a collection of Dn values. To quote the author, "At run-time, the quadtree's visible leafs' Dn values are compared to the current d and the appropriate GeoMipMap level is created from disk.".

Geomipmapping is an efficient LOD algorithm, the author of the paper states that he was able to get high frame rates after the introduction of the algorithm with little amount of popping or visual distortions. Geomipmapping have thus far been used very successfully in many proof-of-concept demos [17] and commercial applications.

2.2.3 QuadTrees

The Quadtree [24][23][9] is a data structure generated in the initialization stages of the program before the terrain is rendered (see figure 2.9). The Quadtree algorithm takes

the heightmap as its input and builds a quadtree [6](pg. 386). In a Quadtree each node has four descendants (unless that particular node is a leaf node at the end of the tree). In the context of terrain rendering the parent node has four children square regions (bottom left, bottom right, top left, top right). Child nodes are marked as visible if they lie within the view-frustum [23]. Therefore starting from the root node and traversing down it is possible to find which blocks are visible (i.e. the blocks that lie in the viewing frustum) and which aren't. The blocks that are visible are sent to the graphics pipeline, those blocks that are not visible are discarded. Unfortunately the terrain patches which lie in the viewing frustum can still be very expansive, detailed and slow to render. Therefore often other LOD algorithms, such as Geomipmapping and ROAM, have to be used in conjunction with the Quadtree.
There is a variation of the QuadTree algorithm called the OctTree.

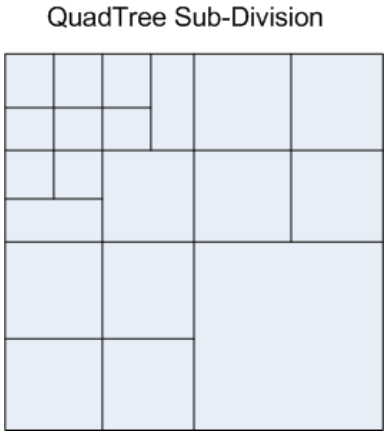


Figure 2.9: A QuadTree representation of a terrain heightmap.

Chapter 3

Meteorology and Geomorphology

Natural meteorological and geomorphological events have an impact on the real-world terrain profile. One can distinguish short-term and long-term events. Short-term events describe relatively instantaneous processes such as earthquakes, tornadoes, tsunami, etc... Long-term events describe processes such as erosion [4] which takes hundreds or even thousands of years to make a serious impact (subject to terrain type and weather conditions). The rate of erosion depends on the soil type, temperature variation, precipitation and wind.

The climate of a particular area strongly depends on the regions location, geography and wind patterns [25](pg. 36-39). The climate itself generally describes air temperature, precipitation and wind. The weather is the state of the atmosphere at any particular location. The weather includes the "temperature, pressure; and humidity of the air; the wind direction and speed; and phenomena such as clouds, rain and snow." [25](pg. 242). The weather is mostly affected by large-scale movements of air.

3.1 Temperature

The air temperature is the measure of the average speed of air molecules, or in other words, the molecular kinetic energy. Changes in air temperature near the surface of the earth are mainly caused by: shifts in climate, seasonal changes, position of the sun and movements (wind) of air masses and fronts [28](pg. 33). Large cold air masses moving from an ocean can instantaneously cool everything down. Whilst warm air masses can quickly raise the overall temperature. The air temperature is greatly affected by topographical features, proximity to large bodies of water, latitude and elevation [28](pg. 34). Temperature changes are usually greater near the surface of the earth rather than higher up in the atmosphere.

The air temperature changes directly as the result of earth heating up (because of the sun) and cooling down. Generally, the temperature is at its warmest during midday when there is maximum possible sun available and coldest at dawn. Clouds and wind have an effect on the temperature. On a cloudy day it is colder as the clouds block sun rays. Wind can also play a part as it can quickly push hot air away from its origin. On average, the temperature at sea level is 15 degrees Celsius [1]. However, this number varies according to the latitude, altitude, time of the day and season. Generally the temperature is the hottest in the tropics (see the first two tables - latitude 0-10) and the coldest near both poles (latitude 70-80) [1]. The coldest temperature ever recorded was -88 degrees Celsius and the hottest temperature was 58 degrees Celsius [1]. The location closer to the poles receive less solar energy from the sun because sun ray's strike the earth "less directly" while the locations near the equator get more solar energy [25]. It must be mentioned that temperature changes cause air pressure changes and as winds move generally from

areas with high pressure to areas with lower pressure, temperature changes affect how the winds flows [25]. The land also affects the temperature, land warms and cools much faster than water. Therefore air over land can heat up much faster than over an ocean. Mean sea level temperatures at a particular region can be determined by the regions latitude.

Lat. Deg.	Summer Temp (deg)	Lapse Rate	Winter Temp (deg)	Lapse Rate
0-10	30	5	25	4
10-30	30	8	20	5
30-40	25	5	15	5
40-50	20	6	5	3
40-50	15	5	-10	3
70-80	5	2	-30	2

Table 1, reveals mean sea level temperatures for different latitudes in the northern hemisphere. The lapse rates reveals the average temperature decrease per km of altitude [18].

Lat. Deg.	Summer Temp (deg)	Lapse Rate	Winter Temp (d)	Lapse Rate
0-10	30	5	30	4
10-30	25	8	20	5
30-40	25	5	15	5
40-50	20	6	10	3
40-50	10	5	-10	3
70-80	5	2	-20	2

Table 2, reveals mean sea level temperatures for different latitudes in the southern hemisphere. The lapse rates reveals the average temperature decrease per km of altitude [18].

3.2 Winds and Pressure

The wind is a collective of many air molecules [25] which move in a particular direction. Air generally moves as the result of difference in air pressure. Air moves from areas with high air pressure to areas with low air pressure. Atmospheric pressure of a point is the weight of a vertical column of air above that point [25](pg. 16). Differences in pressure cause horizontal motion (winds) and vertical motions (convection) [28]. Due to the rotation of the earth, the wind doesn't directly flow from high pressure areas to lower pressure areas. Rather it, in the northern hemisphere winds flow clockwise from high pressure zones to low pressure zones and in the southern hemisphere counterclockwise. This factor that influences the winds direction is called the Coriolis factor or the Coriolis effect [28]. The velocity of winds depends upon pressure gradient and Coriolis forces. The Coriolis force arises from the fact that the movement of masses over the earth's surface depends upon the rotation of earth. In fact, it means that this force is a function of earth's angular velocity of spin and the latitude of the position in question [1]. The velocity of wind (V) in an idealized case (geostrophic wind) is given by the following formula [1]:

$$V = (1 / 2w*(sin(pheta))*raw) * (dp/dn)$$ [14]

Where, w is the angular velocity of earths spin equal to 7.29 x 10-5 radians/sec, pheta is the latitude, raw is the density of air which is, in general case, the function of temperature and dp/dn is the pressure gradient.

The strength of the wind is proportional to the pressure difference (gradient) between high and low pressure zones. When the difference is high the wind is strong and when the difference is low the winds are weak.

There are different types of winds such as Trade Winds (global wind belts), valley winds and diurnal winds (mountain winds) [1][18]. The trade winds blow in one generally

one direction. The direction depends on latitude. If the selected latitude ranges from 0 to 30, these winds blow from east to west (negative direction) for both hemispheres. Otherwise, they blow west to east (positive direction) [1].

Diurnal winds or local slope winds occur as the result of temperature difference between high and low altitudes of the terrain. "Winds blow up the terrain (upslope, up-valley) when surfaces are heated during daytime and down the terrain (downslope, down-valley) when surfaces are cooled during the nighttime. Diurnal mountain winds are driven by horizontal temperature differences." [18].

3.3 Hydrological Cycle and Relative Humidity

The hydrological cycle describes the evaporation, condensation and precipitation cycle that occurs in nature. The first stage, evaporation, occurs when water vapor rises up in the atmosphere owing to vertical air currents [25] and lower than air molecular weight. As vapor rises it cools down and expands. Some vapor molecules stick together forming clouds (condensation). Therefore clouds can form due to convection when warm air, with vapor particles, lifts up and cools down. These clouds are called Cumulus [25].
The temperature at which condensation occurs is called the Dew Point. At this point more condensation than evaporation occurs and water particles start forming into clouds. If enough condensation has occurred the cloud can produce rain or snow (precipitation). The hydrological cycle is a constant exchange of water between the atmosphere and the ground [25].

It is impossible to determine how much evaporation can occur from the ground via a physical formula. However, there have been a number of empirical formulas such as Hargreaves' Equation and Hamon's Equation [19] created to estimate it.

Hargreaves equation uses the temperature, difference between mean monthly maximum and mean monthly minimum temperatures, water equivalent of extraterrestrial radiation, latitude of the site, sunset hour angle, solar declination and relative distance of the earth from the sun [19]. Hamon's Equation uses the average amount of daylight hours per day during the month in which day t falls, saturated vapor pressure at temperature T and temperature itself to estimate the potential monthly evaporation rate [19]. Please see Appendix A for the formula.

The water vapor condenses into water particles at the Dew point which is set according to the following formula [21]:

$$T_d = (b * \alpha(T, RH))/(a - \alpha(T, RH)) \deg$$
$$\alpha(T, RH) = (a * T)/(b + T) + ln(RH)$$

Where a = 17.27, b=237.7C, $0C < T < 60C$ and $0.01 < RH < 1.00$. RH is relative humidity. RH is calculated as the ratio of the volume of evaporated particles to the free volume. Td is the Dew point temperature. Relative humidity can be calculated as a ratio of actual vapor pressure and saturated vapor pressure or actual vapor density and saturated vapor density [21]. The relationship between density and pressure is as follows (assuming model of ideal gas): P/RT = beta/mu where P is actual vapor pressure (in pascals), R is gas constant (8.314 * J/mol*K), T is temperature (in Kelvin), beta is density (kg/m^3) and mu (molecular weight) for H20 is equal to 18 [21].

3.4 Erosion

There are two general types of erosive processes: hydraulic erosion and weathering erosion [4, 20]. These two types of erosive processes can be modeled and used independently.

Terrain deformation can be evaluated using the surface process model (SPM). The SPM calculates long-term (millions of years time scale) changes in topography that result from mass transport processes [26]. Mass transport represents the cumulative effects of hill slope processes and is modeled as linear diffusion [26]. It means that it can be described with a parabolic type partial differential equation. The diffusivity coefficient significantly depends upon climatic influences and the age of terrain. According to Environmental Change and Tropical Geomorphology [4], estimates of soil loss range from 25 to 19,000 kg/ha*yr for tropical regions. Rainfall is one of the major contributors to the erosion process with the linear relationship between the annual erosion and mean annual rainfall [4].

3.4.1 Hydraulic Erosion

Hydraulic erosion involves depositing a quantity of water (from rain or otherwise) on the terrain and depositing a quantity of sediment in it. As the water flows down the terrain it carries the sediment together with it. The erosive power of a specific quantity of water depends on its volume and how much sediment it can carry. The paper, "The Synthesis and Rendering of Eroded Fractal Terrains" [20], gives the following example of a hydraulic erosion model. First the amount of water passed is calculated as: amount of water passed (cw) $= min(w_t^v, ((w_t^v + a_t^v) - (w_t^v + a_t^v)))$
Where v is a vertex at time t with an altitude of a_t^v , volume of water expressed by w_t^v and the amount of sediment expressed by s_t^v . At every time step the excess water and the sediment suspended in it is passed to a neighboring vertex below it.

3.4.2 Thermal Weathering Erosion

Thermal weathering erosion is a process of knocking sediment loose with the wind [20]. This sediment is then collected at the bottom of the incline. According to The Synthesis and Rendering of Eroded Fractal Terrain, the thermal weathering erosion model can work in the following way. At each time step (t+1) the algorithm "compares the difference between the altitude a_t^v at the previous time step t of each vertex v and its neighbors u to the global constant talus angle T. If the computed slope is greater than the talus angle, we then move some fixed percentage c_t of the difference onto the neighbor".

Chapter 4

Conclusion

The four terrain generation algorithms, described in section 2.1, all work reasonably well. The Perlin Noise algorithm is arguably one of the better algorithms as it can be used to produce a heightmap of any dimension. As opposed to, for example, Mid-Point Displacement which has some restrictions. With Perlin Noise it is also easy to control the roughness and the smoothness of the terrain and enforce height restrictions. The Mid-Point Displacement algorithm suffers from a flaw which prevents it from being used with heightmaps not of the same size. For example, MDP will not work with a 128x256 heightmap but only with a 128x128 or a 256x256 heightmap. The MDP algorithm could certainly be modified to account for different dimensions but in its native form it will not work.

The fault formation algorithm deserves a mention as it can be used to produce smooth heightmaps. The problem with it is that it requires a smoothing filter pass on the heightmap to achieve reasonable results. Without using a smoothing filter the heightmap, most of the times, looks too rough and too uneven to be used. Apart from this minor inconvenience this algorithm is technically solid and it has been used with good results. It can also be used for simulation of tectonic plate movement (earthquakes) when large areas rise up or fall down. Finally, the particle deposition algorithm can also be quite effectively but mostly for building worlds made out of islands. In order to generate very large terrain with high mountains a very large number of particles would have to be "dropped" onto the terrain. This process could potentially take longer than other algorithms. On the other hand particle deposition can be controlled very effectively and easily. For worlds with many little islands and terrain with low elevation this algorithm could come in very useful. All-in-all for the generation of terrain Perlin Noise is one of the best tools. Unsurprisingly TerraGen uses Perlin Noise as one of its primary algorithms for constructing terrain. The other algorithms it uses are just basically variations of noise and turbulence functions.

The three CLOD algorithms mentioned in section 2.2 are very effective and fast. The QuadTree algorithm can be and is often used in conjunction with the two other algorithms (GeoMipmapping and ROAM) to dynamically split the terrain and determine the terrain patches visible to the camera at any given moment. The ROAM algorithm is one of the most widely used CLOD algorithms today. It has an advantage over other algorithms because it is fast (in part due to priority queues, bintrees and frame coherency), it yields optimal meshes, it naturally prevents cracks between patches and it can be enhanced further. One problem for the ROAM algorithm is that a mesh created with ROAM can't be rendered using triangle strips due to its "diamond" nature. Thus ROAM's performance suffers in that regards.

GeoMipmapping is another good CLOD algorithm. It uses patches which it dynamically adjusts by enabling and disabling vertices. It is a fast algorithm but it leads itself

to popping (ROAM has this problem too) and cracks between patches. Nonetheless, all these problems are actually preventable. The GeoMipmapping algorithm is fast and it can be successfully used with a QuadTree.

Meteorological conditions, and hence weather, depend upon the following factors: air temperature and pressure, strength and direction of wind and precipitation. The average air temperature is a function of altitude and latitude of the terrain, movement of air masses, season and time of the day. In a simulation, air temperature can be partly set as an initial condition based on the chosen latitude, season and time of the day. Partly, it will be based upon the topography of the terrain and movement of air.

Wind occurs due air pressure difference. The topography of the terrain can affect air flow acting both as a barrier and a "wind generator" (slope winds). In a simulation, two types of wind can be distinguished. Latitude dependent trade winds could be used as an initial condition. Diurnal (slope) winds can be produced in a simulation based on air pressures zones and the underlying terrain.

Evaporation entails the conversion of water into vapor and its rise up into the atmosphere. The vapor gradually cools down and at a certain temperature, at the Dew Point, changes from a gaseous state to a liquid state. Clouds of sufficient size can produce rainfall or snowfall depending on the atmospheric temperature.

Winds, temperature and precipitation affects the erosion rate of the terrain. Therefore weather has a long term effect on the terrain topography. Terrain topography, in turn, has an effect on weather conditions. Thus the terrain and weather constantly exercise influence on each other.

Chapter 5

Appendix A

Hamon's equation:

$$E = (2.1 H_t^2 e_s)/(T_t + 273.2)$$

where:

- E = rate of evaporation, [mm day]
- H_t = average number of daylight hours per day during the month in which day t falls (the function of latitude and season).
- e_s = saturated vapor pressure at temperature T
- T_t = average seasonal temperature, day t [C, deg]

e_s, a function of temperature, is the saturated vapor pressure of water in air at temperature T, is calculated via the following formula:

$$e_s(T) = 0.6108 exp((17.27T)/(237.3 + T))$$

These formulas are valid only for positive temperature. When temperature is negative there is no evaporation.

References

[1] Barry, R.G. & Chorley, R. J. *Atmosphere, Weather and Climate*. Holt, Rinehart and Winston, United States, 1970.

[2] Boer, W.H. Fast Terrain Rendering Using Geometrical MipMapping. *E-mersion project publication*, 2000.

[3] Deloura, M.A. *Game Programming Gems 3*. Charles River Media, United States, 2003.

[4] Douglas, I. & Spencer, T. *Environmental Change and Tropical Geomorphology*. George Allen & Unwin, London, 1985.

[5] Duchaeneau, M., Wolinsky, M., Sigeti, D.E., Miller, M.C., Aldrich, C. & Mineev-Weinstein, M.B. ROAMing Terrain: Real-Time Optimally Adapting Meshes. 1997.

[6] Eberly, David H. *3D Game Engine Design*. Morgan Kaufmann Publishers, United States, 2001.

[7] Ebert, S.D., Musgrave, F.K., Peachey, D., Perlin, K. and Worley, S. *Texturing & Modelling: A Procedural Image*. Morgan Kaufmann Publishing, United States, 2002.

[8] Elias, H. Perlin Noise. *http://freespace.virgin.net/hugo.elias/models/m_perlin.htm*, last visited: 26/4/2004, 2003.

[9] GDC2003 Course. http://lodbook.com/course/. *last visited: 26/7/2004*, 2004.

[10] GeoScape 3D. http://www.geoscape3d.net/. *last visited: 26/7/2004*, 2004.

[11] Haki, Henri. *Diamond Terrain Algorithm*. University of Stellanbosch, 2001.

[12] http://dictionary.cambridge.org. http://dictionary.cambridge.org. *last visited: 26/7/2004*, 2004.

[13] http://geology.usgs.gov. http://geology.usgs.gov. *last visited: 26/7/2004*, 2004.

[14] Institute of Arctic & Alpine Research. http://snobear.colorado.edu/. *last visited: 26/7/2004*, 2004.

[15] Leveller. http://www.daylongraphics.com/products/leveller/. *last visited: 26/7/2004*, 2004.

[16] Lighthouse 3D. http://www.lighthouse3d.com/opengl/terrain/. *last visited: 26/4/2004*, 2004.

[17] lScape. http://www.futurenation.net/glbase/index.php. *last visited: 26/7/2004*, 2004.

[18] Lutgens, F.K. & Tarbuck, E.J. *The Atmosphere: An Introduction to Meteorology.* Prentice Hall, United States, 2001.

[19] Marine Ecosystem Modelling Group. http://data.ecology.su.se/. *last visited: 26/7/2004*, 2004.

[20] Musgrave, F. K., Kolb, E.C., & Mace, S.R. The Synthesis and Rendering of Eroded Fractal Terrains. *Yale University Department of Mathematics*, 2001.

[21] Paroscientific. http://www.paroscientific.com/dewpoint.htm. *last visited: 26/7/2004*, 2004.

[22] Perlin, K. An Image Synthesizer. *Computer Graphics*, 19(3), 1985.

[23] Polack, T. *Focus on 3D Terrain Programming.* Premier Press, United States, 2003.

[24] Rottger, S., Heidrich, W., Slusallek, P & Seidel, H. Real-Time Generation of Continuous Levels of Detail for Height Fields. 1998.

[25] Stein, Paul. *The Macmillan Encyclopedia of Weather.* Rosen Book Works, United States, 2001.

[26] Summerfield, M.A. *Geomorphology and Global Tectonics.* John Wiley & Sons, London, 1999.

[27] Terragen. http://www.planetside.co.uk/terragen/. *last visited: 26/7/2004*, 2004.

[28] Whiteman C. David. *Mountain Meteorology: Fundamentals and Applications.* Oxford University Press, United States, 2000.