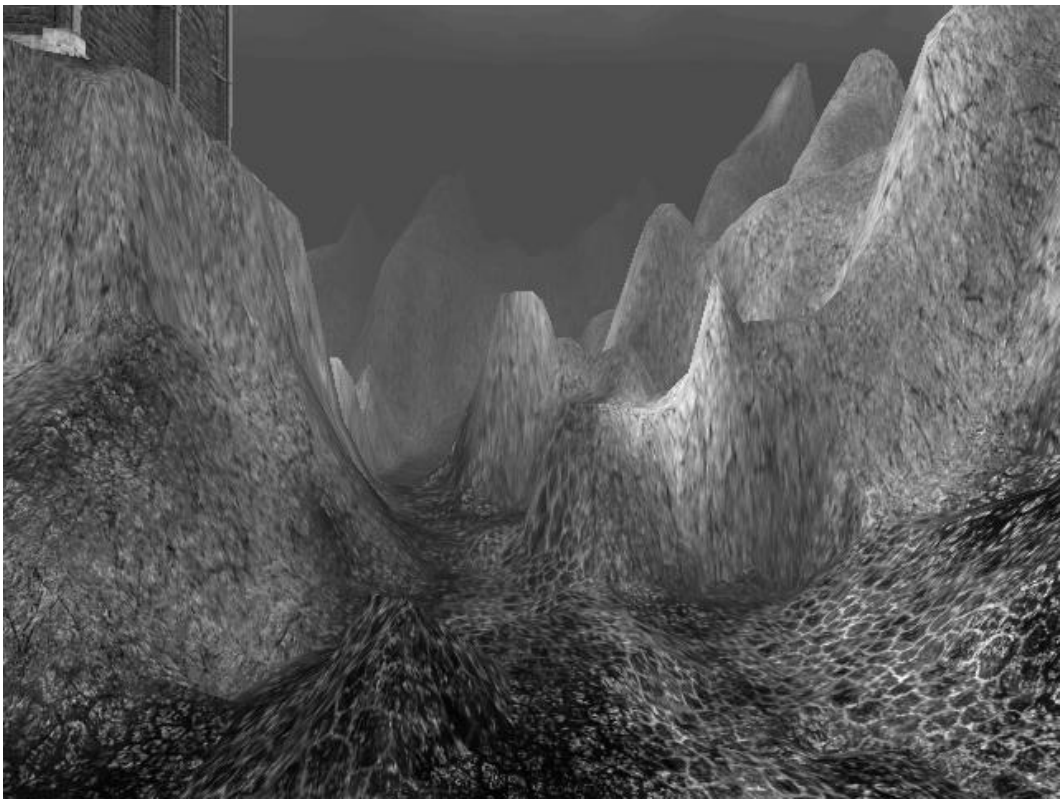
Adding Terrain Rendering Capability to the Starbreeze Engine



Jim Tilander Andreas Brinck

April 23, 2001

Abstract

We have created a terrain rendering plug-in for the Starbreeze 3D-engine. The plug-in uses a static level of detail scheme to reduce the geometrical complexity of the landscapes far away from the viewer. We use the concept of texture layers to enable detailed variations in the appearance of the surface.

We have also created a plug-in for the existing editor to enable simple editing of the landscapes.

Sammanfattning

Vi har skapat en insticks-modul till Starbreeze 3D-motor för att rita terräng. Modulen använder statiska detaljnivåer för att reducera landskapens geometriska komplexitet långt bort från betraktaren. För att möjliggöra detaljerade variationer i ytans utseende använder vi oss av konceptet texturlager.

Vi har också gjort en modul till den befintliga editorn för att enkelt kunna editera landskapen.

Preface

Acknowledgments

We would like to thank Magnus Högdahl and Jens Andersson at Starbreeze Studios for invaluable help and insights into the Starbreeze engine and Ogier.
We would also like to thank the CyberLoop team for hosting us during the work.
Finally we thank our supervisor Magnus Bondesson.

Contents

1 Introduction

Rendering natural looking landscapes has been the focus of a lot of research in recent years. A substantial part of the research has been funded by the U.S. military for developing flight simulators capable of rendering realistic terrain (as in [1]). In the past two years the technology for rendering the huge amount of triangles needed in real time has become available in standard PC's[1], further increasing the interest for terrain rendering.

The game developers Starbreeze Studios have a very advanced game engine in development. The engine didn't however include any capabilities to render large landscapes. They deemed it necessary to have this in their engine, mainly for future projects.

1.1 Organization of the Paper

This paper is divided into three parts: theory, implementation and discussion. In the theory part we will briefly describe some algorithms that are useful for terrain rendering. The implementation part of the paper gives a detailed description of the work done by us in adding terrain rendering capabilities to the Starbreeze engine and in the discussion part finally we will share some of the insights we gained during the project and describe some of the problems we ran into.

2 Theory

Realistic rendering of large scale terrain models poses a lot of interesting problems. In the following section we will discuss two of the main ones: detail reduction and seamless texturing. We will also briefly describe a method to speed up the rendering called *triangle stripping*.

2.1 A Survey of Current Dynamic LOD Algorithms

We will present a small selection of algorithms that are currently being used for terrain rendering and their respective strengths and shortcomings. All of these algorithms share one common characteristics, they work with a regular grid polygonization representation of the surface i.e. the terrains height is sampled on a regular grid and from this information a surface is constructed.

This representation has one serious flaw, it is unable to correctly represent some terrain features such as caves and overhangs, still it is the de facto standard. There is a vast library of real world *Digital Elevation Maps*, or DEM's for short, available in this format. Most of the terrain rendering algorithms were originally aimed at various aircraft simulators and on the spatial scale that these simulators work caves and overhangs are neglectable; most DEM-files are sampled on an interval of 50 meters or more.

The datasets associated with terrain rendering are usually huge, a patch of terrain 50 km on each side will consist of roughly 2 million triangles, which is unfeasible to render at an interactive framerate even on the most high end machines. The purpose of these algorithms is to reduce the triangle count to an acceptable level. From information about the observers position and the terrains topology the algorithms construct an approximate representation of the surface, details are removed far from the viewer and where the curvature of the landscape is low. These calculations are done on a per frame basis, hence the name dynamic LOD. For really large terrains it is necessary to implement a disk paging system for the terrain data, but this is an advanced problem in itself and will not be covered here.

2.1.1 ROAM

ROAM is an abbreviation for *Real-time Optimally Adapting Meshes* and was first presented in [1]. Since it is the foundation of our terrain engine it will be presented in some more detail than the

[1] E.g. the NVidia GeForce family of processors

rest of the algorithms.

Binary triangles ROAM uses a partitioning scheme referred to as binary triangle tree. Each triangle can have two congruent children that are created by bisecting the triangle along its longest edge, see Fig. 1.

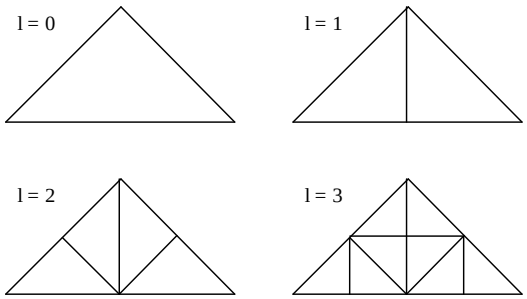


Figure 1: The first four levels of the binary triangle tree.

Each triangle stores five pointers, left, right, base and left and right child, see Fig. 2.

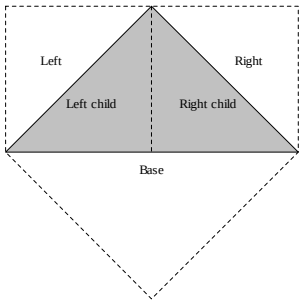


Figure 2: Triangle pointers.

The initial triangulation of the landscape is constructed by creating two binary triangles and linking their bases to each other. All the other pointers are initially set to *NULL*.

Splitting Before we continue we notice that the base neighbor of a binary triangle is either located on the same level as the triangle or one level up, see Fig. 3.

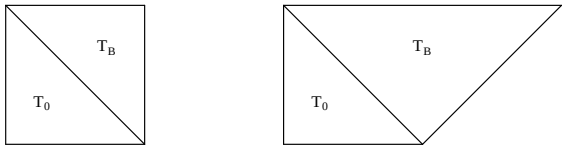


Figure 3: The two possible neighbor arrangements.

If the base neighbor of a triangle we wish to split is located on the same level, the operation is easy. We just create two new children for each of the triangles and initialize their pointers, see Fig. 4.

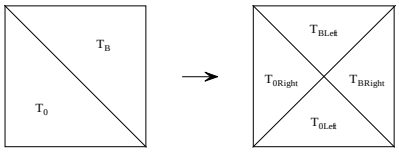


Figure 4: Split operation.

If the base neighbor is not on the same level we first apply the split function on it. Since there is no guarantee that the base neighbors base neighbor is on the same level as the base neighbor further recursive calls to split may have to be made until we reach a triangle who's base neighbor is on the same level as itself, this operation is referred to as *force splitting*, see Fig. 5.

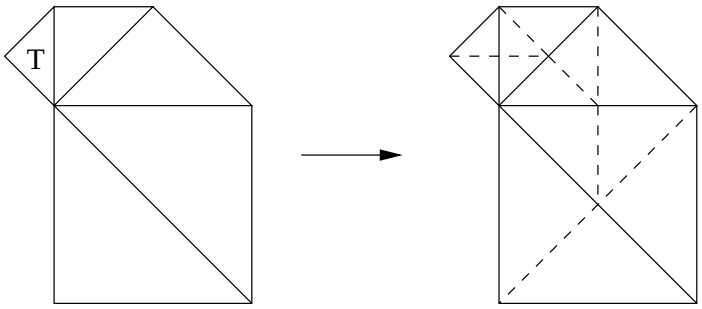


Figure 5: Recursive force splitting of T.

Splitting criteria To determine if a triangle needs splitting we construct a pie shaped bounding volume, a so called *wedgie*. The height of a triangle T's wedgie is set to $2 \cdot \delta = 2 \cdot Max(|f(x, y)|)$ where $x, y \in T$, see fig. 6.

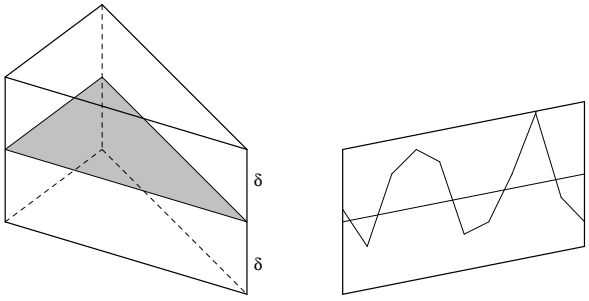


Figure 6: Wedgie.

If we project the wedgie as it is seen by an observer onto the screen we get an error in pixels that is of the magnitude $C \cdot \delta / d$, where d is the distance from the observer to the wedgie and C is a

constant that depends on the screen resolution and the current field of view. The error naturally also depends on the orientation of the wedgie. The error will be more prominent if we look at the wedgie from the side than if we look at it from above. We ignore this fact and assume that the wedgie is oriented in the worst possible way to get an upper limit for the error.

Split Queue The authors of [1] suggest that one maintains a priority queue where the triangles are ordered by screen error. When a triangle is split it is removed from the queue and it's two children are inserted. We keep splitting triangles until the total number of triangles meet some target count, we will then have acquired the optimal representation of the terrain given that number of triangles.

Merge Queue If the viewpoint is moving slowly we can expect the triangulation to be approximately the same between two consecutive frames. To utilize this frame-to-frame coherence we can create a merge priority queue where we keep a list of mergable diamonds[2]. By merging triangles in the merge queue and splitting triangles in the split queue we can obtain a new optimal triangulation. We don't reevaluate the priorities in the merge and split queue between frames but hope that they change so slowly that we can reuse the priorities from the previous frame.

Variations There exists a risk with the priority queue strategy. If the framerate drops for some reason the position of observer will be considerable different between frames, which will in turn disrupt our assumption that the triangulation stays approximately the same. We thus need to make a lot of split and merge operations to correct the triangulation. These time consuming operations will further reduce the framerate. We have entered a vicious circle that will eventually drop the framerate to zero.

A common variation of ROAM that solves this problem is the described in [16] and [3]. Instead of maintaining a priority queue and specifying a target triangle count we can start with the basic triangulation and specify an acceptable error tolerance and keep triangulating until this tolerance is met.

2.1.2 The Lindstrom Algorithm

The Lindstrom algorithm, described in [2], works in nearly the opposite way of the ROAM algorithm. Instead of starting with a coarse triangulation and adding triangles until a specified tolerance or triangle count has been met, Lindstroms algorithm starts with the finest triangulation and then successively removes vertices. For this reason Lindstrom is often referred to as a bottom-up algorithm and ROAM as a top-down.

Since the algorithm involves looking at all the triangles in the finest subdivision of the surface it is computational expensive. To reduce this cost the terrain is divided into subblocks that are each stored in a number of different resolutions. At the beginning of a frame the algorithm makes a rough estimate to decide which resolution is appropriate to start with on each block. The different resolution blocks are not stored explicitly, a coarser resolution block is obtained by removing every other row and column from a finer resolution block.

The Lindstrom algorithm uses a simpler formula than the ROAM algorithm to estimate the error. Where the ROAM algorithm looks at the difference between the current triangulation and the finest possible triangulation, the Lindstrom algorithm only examines how much the terrain is changed by going from the current triangulation to the next. This method has the advantage of not requiring any additional storage. The fact however is that since the introduction of ROAM the use of Lindstroms algorithm has declined severely.

[2] A diamond is the set of four triangles that results from the type of split shown in fig. 4

2.1.3 Other Algorithms

In [15] a method is presented that can be described as geometrical *mip maps*. The terrain is divided into squares and for each of these multiple representations, *levels*, with different level of detail are saved. Each level has dimensions of the form $2^n + 1$ which are used when the square is at certain predetermined distances from the observer. A square can only be adjacent to a square that is on the same or one level less. To connect two squares that differs one level, every other vertex on the edge of the higher level square is removed.

2.2 Texturing

Handling dynamic triangulations and texture mapping involves a couple of problems. Since the triangulation may change at any time we must have some way to adapt the texture coordinates. A naive algorithm is to use a very large texture for the whole landscape. One of the drawbacks is that the resolution of the texture will suffer since the texture is severely constrained by the texture memory available on the graphics card. If one moves the camera relativly close to the terrain the low resolution will result in very large texels stretched across the triangles.

2.2.1 Tile Algorithm

One solution is to use a set of tiles. The tile themselves need to have a decent resolution, but clever use of tiles should result in heavy reuse and thus require less texture memory on the graphics card. This will enable higher resolution tiles, which will improve image quality. However, there are a number of problems with this approach. Consider Fig. 7. In the figure, the grid squares are the tiles. The triangle A is clearly larger than the one individual tile, whereas all triangles in triangle group B are smaller than one individual tile. It's impossible to map a single tile onto the triangle A, and since the tiles covered by the triangle could be any combination, we would be forced to assemble a larger image at runtime and upload it to the graphics card in order to texture the triangle. So the only viable option is to make sure that this never happens, which effectively limits all triangles to fit within a tile. This might not sound so bad, but consider the most extreme case of the ROAM triangulation, a completely flat surface. ROAM should do a pretty good job of triangulating this with simply two triangles. It would then be forced to continue splitting the triangles until they all are smaller than an individual tile, i.e. wasting a lot of triangles because of the texturing scheme which could have been saved if it merely was a matter of geometry.

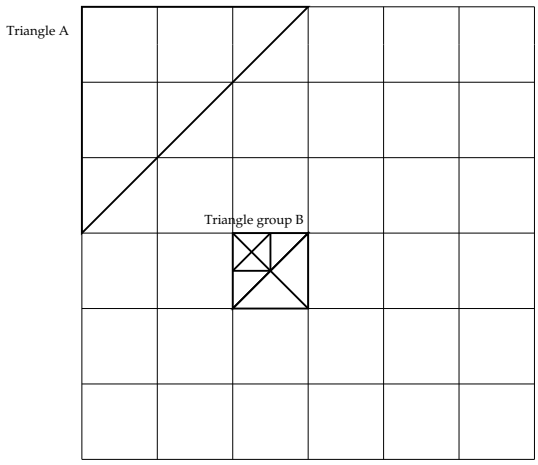


Figure 7: Different triangle sizes in a grid

Another problem with the tiles is how to handle tile transitions. Usually one has a *tile set* which consists of a palette of tiles we can use. Assume that we have n different materials in this tile set. Furthermore, lets assume that we assign materials to the vertices, instead of the tile itself. To handle all transitions we need n^4 different tiles. Of course, we can reduce this number somewhat by disallowing some transitions and using a lookup table which given four materials, one for each corner, returns an appropriate tile.

2.2.2 Multipass Algorithm

By introducing the concepts of layers we can trade speed to solve the texture mapping problems. A layer consists of an alpha mask and a detail texture. The alpha mask covers the whole terrain and the detail texture is repeated across the terrain. The alphamask texture doesn't have to be a high resolution texture to achieve good results. The most important thing is the detail texture that will be visible when you zoom in close to the landscape.

This technique requires a multitexture unit in hardware, in order to blend the detail texture and the alphamask together. One can stack several layers upon each other and blend them together, enabling smooth transitions between sand, grass and stone for instance, thus making it easy to create a beach without resorting to an endless set of different texture tiles, see Fig. 8.

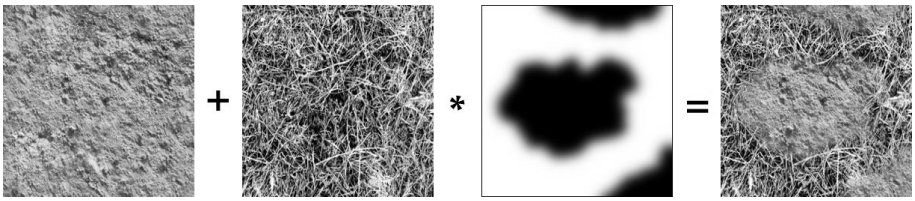


Figure 8: Rock, grass and an alpha channel are blended together.

2.3 Triangle Stripping

One great way to speed up the rendering is to replace the individual triangles with triangle strips. Sets of individual triangles requires $3 \cdot n$ vertices to be transformed, where n is the number of triangles, while triangle strips require only $n+2$ transformations. Most modern rendering API:s such as Microsoft's Direct3D and OpenGL [11] supports triangle stripping.

It is clear that the longer strips we can create the bigger the gain is. Hence we need an algorithm that takes a list of individual triangles and constructs and optimal set of triangle strips from it. Unfortunately it can be shown that the construction of such a set is *NP-Complete*, for a detailed presentation see [10].

2.3.1 Heuristics

Fortunately because of the importance of the problem many heuristics for creating good strips have been developed. We choose the method described in [6]. It is essentially a *greedy* algorithm that chooses the next strip as the one that shares the most vertices with previous strips. The purpose of this is to minimize the number of isolated triangles.

We have made some further optimization that takes the cache on the *Graphics Processing Unit* into account. This will be explained in detail in the implementation section of this report.

3 Implementation

Here we will describe the terrain algorithm that we have implemented and some of the peripheral problems that have come up during the development.

3.1 Engine

The Starbreeze Engine consists of two stages, compilation and game. During compilation information is gathered and transformed into a streamlined format suitable for the actual game. As much precalculation as possible is done in the compilation phase. In the game only a small subset of the code is active, just for rendering and physics.

3.1.1 A Short Description of the Starbreeze Engine

The Starbreeze Engine utilizes the *Object Oriented Programming*, OOP, paradigm, with a strict inheritance tree. Almost everything inherits a single object which keeps track of the class name, enables reference counting and error tracking. The engine was originally designed to handle both PC and the SEGA Dreamcast. On the PC, several graphics API's are supported, GLIDE, OpenGL and DirectX. The engine consists of four major parts, MOS, MCC, XR and Game:

- MCC. Moose Core Classes. These classes provides the basic functionality like matrix handling classes, I/O wrappers, data structures like lists etc.
- MOS. Moose OS. Wrapper around the operating system. Handles interaction with graphics hardware, disk access.
- XR. eXtended Reality. Handles rendering and physics of objects in the world.
- Game. The actual game code, i.e. server and client code.

We've mostly worked against MOS, MCC and XR in our implementation.

3.1.2 LOD management

We use binary triangles [1] to build a triangulation based upon a specified error tolerance. The bintriangle structure has some impact on the overall structure, such as constraints on patch dimensions etc. Bintriangles combines the area covering properties of a quadtree with the familiar properties of a binary tree and all in all proves to be a very useful data structure.

The initial test implementation used a dynamic LOD algorithm in real time based upon each triangle's distance from viewer, see Fig 9. Benchmarking [3] yielded approximately 12000 triangles at 30 fps, i.e. 0.36 MTri/s, which is quite on par with current implementations (e.g. [16]).

[3] Tests were performed on a dual P3 600MHz with a GeForce 32DDR. Implementation was completely unoptimized.

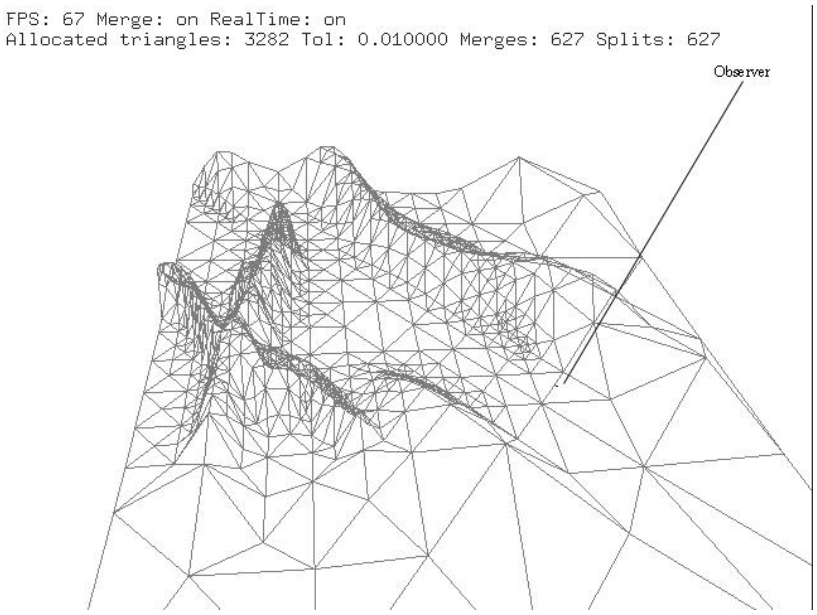


Figure 9: ROAM like dynamic LOD. Notice that the observer is facing northwest, and that the mesh is noticeably sparser outside the frustum.

The ROAM algorithm is almost the direct opposite of a hardware graphics accelerator friendly way of drawing scenes. The scene changes constantly, at worst each frame. Since possibly all triangles are recalculated for each frame, they must also be uploaded to the card each frame, which consumes a lot of precious bandwidth between host and graphics adapter. The test implementation showed this to be all too true. At no time was the triangle pipeline on the graphics card full, the potential of the card was never utilized.

The keyword for today's PC graphics accelerators is *static geometry*, geometry that the card can somehow lock and/or load to faster memory local to the graphics processor. ROAM's approach to change the geometry each frame will simply not do.

In the next phase we moved the triangulation engine we used in real time to the world compiler and precalculated a number of static LOD steps for each patch in the landscape.

The only calculation made at runtime is the distance from patch to viewer that is necessary to determine the appropriate LOD step to use in the current frame.

3.1.3 Physics

Since the terrain engine was implemented as a plug-in it had to support the same functionality as the rest of the Starbreeze engine. One of the things that makes the Starbreeze engine stand out is it's excellent support for realistic rigid body physics. For this to work on the terrain it had to implement three functions:

- Moving point, i.e. line, intersection
- Moving sphere intersection
- Moving OBB, *Oriented Bounding Box*, intersection

Since the landscape contains many hundred thousands of triangles we had to come up with a way to determine which ones we had to check for intersection for a given line, sphere or OBB. Our first approach was to recursively check against the bounding wedgies of the binary tree representing the terrain. If a primitive intersected one of these wedgies it was sent on to the

the two children's wedgies. This process was repeated until the primitive reached one of the tree's leafs. Since the primitive typically reached many leafs a list of the closest intersection was maintained.

When we removed the dynamic LOD algorithm the data for the bounding wedgies of the binary tree was only used for the intersection functions; since this data was typically very large, 16 Mb for a landscape 1000 units in square, we began looking for alternative ways to do the intersection so that we could remove the wedgie data completely. The solution we found turned out to be much simpler and in the end several orders of magnitude faster than the original solution. We used the property that the height of the landscape is sampled on a regular grid in the xy-plane. By projecting the bounding volume of the moving primitive on the xy-plane on which the landscape is sampled we could directly see which triangles that had to be checked, see Fig. 10.

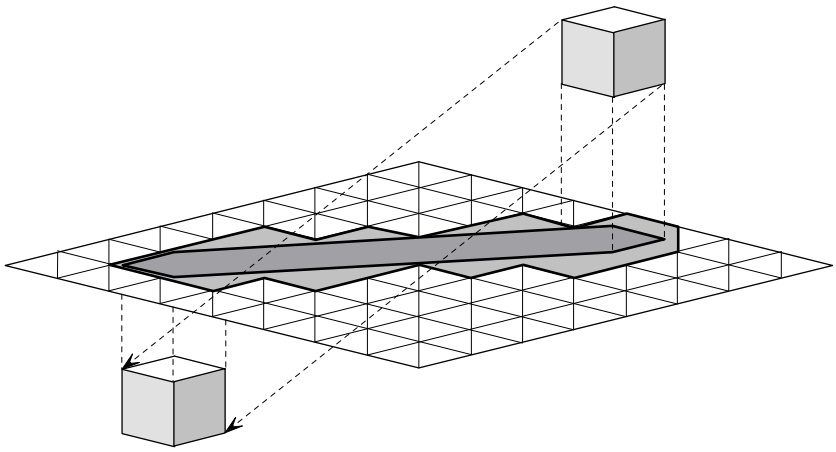


Figure 10: Intersection with moving bounding box. The dark gray area is the projection of the bounding volume and the light gray area is the triangles that need checking.

3.1.4 Texturing

We have used the multipass technique described in the theory section to allow us to use any triangulation on each patch. To minimize the impact of the layers, we've tried to reduce the overhead and excess triangles used in each layer. Triangles that don't change the final image are discarded during the compile phase, e.g. a triangle at layer 0 is completely covered by a triangle at layer 1, and need thus not be drawn at all. In the most extreme case there are no triangles at all in one patch in a particular layer, and we don't need to make any setup (i.e. request that an empty vertex buffer and it's associated textures may be swapped from main memory to the graphics card's memory). This enables an artist to make a layer that represents a trail for example, and only pay the cost of the actual triangles necessary to describe it, which boils down to that we only pay for what we use on the screen.

One problem with texture mapping such a huge landscape is how one assigns texture coordinates to each vertex. We've solved it by allowing the user to specify one of three planes - XY, XZ or YZ - on which the texture coordinates will be projected upon for each layer. A XY mapping means that the X and Y component of the vertex itself are used to calculate the U,V component of the texture coordinate. This extends to the XZ and the YZ case as well.

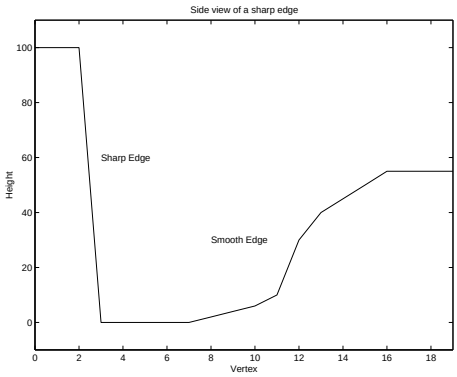


Figure 11: A cross section of a sample landscape with a sharp edge, to the left and a smoother edge to the right.

This enables the user to add a layer with triangles only visible on very steep edges in the landscape to compensate for the stretching effect that a normal XY mapping will produce, i.e. on a very steep edge the texture coordinates of the two adjacent vertices will be very close to each other, even mapping to the same texel in the texture map resulting in only one texel being stretched along the whole edge, see the left edge in fig. 11. The solution is to mark the vertices in a layer and specify a different mapping, either XZ or YZ, to make the wall look good.

3.1.5 Dynamic lighting

Like most other 3D engines on the market the Starbreeze engine uses light maps to display light variations in the virtual world. A light map is essentially a texture that contains information about the luminosity of each point in a surface. Normally the light map is rendered first and afterwards a texture containing information about the color variations of the surface is rendered over it with multiplicative blending, see fig. 12.

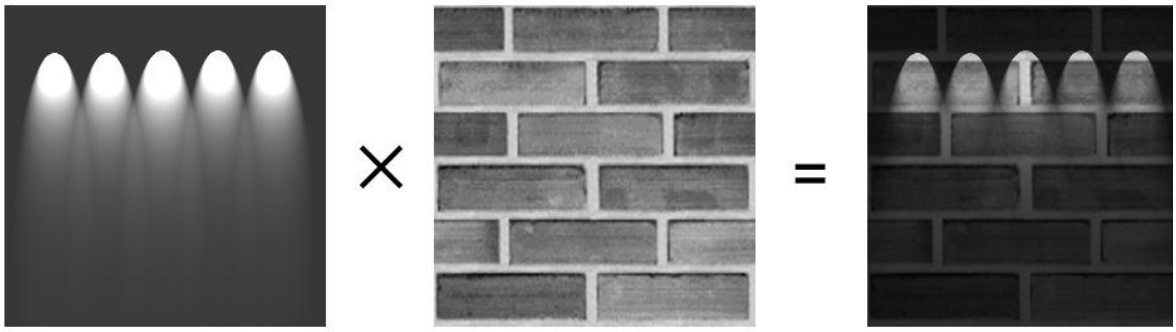


Figure 12: Blending of texture and light map.

Dynamic lighting is created by adding a special texture with the additional light information on top of the existing light map. Unfortunately, this is not possible on the terrain since the operation we would like to perform,

$$\left(\Lambda + \sum_{i=0}^{n} \tilde{\Lambda}_i\right) \cdot \left(\sum_{j=0}^{m} T_j \alpha_j\right)$$

where α_j and T_j represents the texture and alpha mask for stage j and Λ and $\tilde{\Lambda}_i$ represents static and dynamic light maps. Unfortunately this operation is not available on most of the current crop of 3D accelerators. If we remove the dynamic light $\tilde{\Lambda}$ we can solve the problem by first drawing all the textures with additive blending and afterwards draw the lightmap on top with multiplicative blending. To get dynamic light we directly modify the static light map. Since a dynamic light normally affects only a small part of the light map we split it into smaller pieces to minimize the amount of data that has to be sent over the bus to the graphics card when the light situation changes.

3.1.6 Wallmarks

The Starbreeze engine uses a technique commonly known as *wallmarks* to achieve a number of effects such as real time model shadows, bullet holes and blood stains. A wallmark is simply a polygon rendered at the same position (with a slight offset to avoid Z-buffer artifacts) as the underlying surface. With some alpha blended texture mapping it is simple to create effects such as bullet holes.

Since the triangulation can change when switching between LOD levels, the wallmarks must be aware of the different LOD levels, and where exactly the triangles are to be placed. An incoming wallmark is simply a quadrangle. The wallmark is clipped against the triangles it covers in the particular LOD level it will show up in. The clipping is done with the help of a simplified binary triangle tree.

3.1.7 Optimization Techniques

To further speed up the program we have made a number of optimizations that will be described in some detail below.

Hardware Accelerated Geometry A typical landscape needs about 10kTri - 20kTri to describe it on the screen, a huge amount of triangles for today's middle end PC's. These triangles will compete for time and bandwidth on both the CPU and the graphics card with the rest of the Starbreeze engine.

Recently the hardware geometry processors previously only found in high end workstations from Silicon Graphics and Intergraph for instance have found their way to standard IBM-PC compatibles. Graphics programming with an on board processor aka GPU[4] has somewhat different rules. Triangle count is still very important, but equally important is memory management. This includes issues like vertex caching, cache trashing and display lists amongst others.

The onboard GPU has many tasks, one of them is to transform all vertices to view coordinates, which involves a matrix vector multiplication.

The NVidia GeForce chipsets have a cache of transformed vertices, henceforth called vcache, which is about 12 vertices. A cache hit on one of these vertices avoids a full transform, and the need to go through the memory bus. It is thus extremely important to keep the vcache happy. The size of the vcache sets restrictions on the length of any vcache friendly triangle strip. The idea is to make two adjacent triangle strips share as many vertices as possible to make maximum use of the vcache. This fact has been incorporated into our triangle strip generator. As it happens, the structure of the binary triangle tree provides a near optimal drawing order as well if you draw the triangles as you encounter them traversing the tree in depth first order. Our first triangle stripping algorithm didn't take the vcache into account and was thus *slower* than drawing individual triangles as they were encountered in the binary tree. Without consideration of

[4]Graphics Processing Unit, as NVidia calls it

the vcache, rendering with triangle strips instead of triangles actually slows down the rendering process noticeably.

Triangle Removal To reduce the number of triangles in each layer of the terrain we have implemented an algorithm that removes non visible triangles. There are two reasons why a triangle can be removed from a layer:

- The alpha value of the entire triangle is below some threshold value ~ 0.0 and the triangle is thus considered as non visible
- The alpha value of the entire triangle in a layer above the current one is above some threshold value ~ 1.0 and thus obscures the triangle in the lower layer.

To check the alpha coverage of a given triangle we essentially need a *triangle rasterizer*, but to simplify the code we just check the bounding rectangle. This works fine as long as the neighboring triangles have the same alpha value, which will nearly always be the case. The only time where the values are not the same is in the border zone between low and high alpha values in which case the extra border will help to prevent visible seams where the triangles have been removed.

Generation of Texture Coordinates Texture Coordinate Generation enables us to express the texture coordinates as a linear combination of the components of the vertex. This means that we will only have to load the actual vertex from memory, any set of texture coordinates can be calculated internally. With a little help from the memory cache, or smart register usage, the coefficients for the linear combination could be accessed at minimal cost.

This has two large benefits. Firstly, on board memory hit on the graphics card is minimized. Secondly, the system memory requirements are drastically reduced.

3.2 Editor

To make the integration of ordinary geometry and the landscape geometry easier the landscape editing tools were implemented as a plug-in to the existing Starbreeze editor *Ogier*. This choice was also made so that the graphics artists wouldn't have to switch between different programs while they were working.

3.2.1 A Short Description of Ogier

Ogier uses a representation of the scene usually called a *scene graph*. All information about the scene, transformations, geometry, modifiers etc, is stored in the nodes of a *Directed Acyclic Graph* or DAG for short.

If a node contains a transformation matrix for instance this transformation will be applied to all the children of the node.

Ogier shares much of it's code with the actual Starbreeze engine.

3.2.2 Terrain Generation

One of the requirements of the editor was that it should be able to automatically generate believable terrain. When we created the terrain generation module we looked a lot at Matthew Faircloughs excellent shareware scenery rendering program *Terragen* [7]. Although Terragen was designed with a different goal than ours in mind we were able to borrow many ideas from it.

Figure 13: A picture generated with Terragen.

Perlin Noise Just like Terragen we use *Perlin noise* [5] to generate the terrain. Perlin noise was created by Ken Perlin of New York University in the middle of the eighties. It is a way to construct controllable noise that can be used to create two and three dimensional textures. Textures generated in this way are called *procedural* textures. Procedural textures are nowadays the de facto standard for high quality rendering. Unlike regular textures they have infinite resolution and keeps exhibiting new detail as one zooms in on them. Nearly all the computer graphics in major Hollywood pictures are created with *Pixar's* rendering library *Renderman* [8], a library that uses Perlin noise to create textures.

If we look closely at some natural phenomenas we will see that they all exhibit a common property, they are a combination of large and small features. If we zoom in we will see that the features are repeated but on a much smaller scale. This self similarity was studied by Benoit Mandelbrot who coined the word *fractal* [9] to describe it.

It is this property that is the foundation of Perlin noise. Noise is synthesized by adding a sum of random functions with increasing frequency and decreasing amplitude, see Fig. 15. The random functions are constructed by assigning random values to the points lying on multiples of the wavelength and smoothly interpolating between them, see Fig. 14.

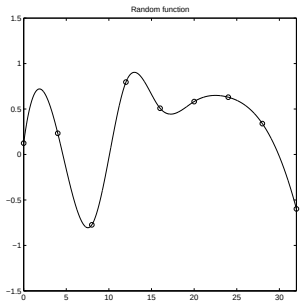


Figure 14: A smooth random function. The circles indicate the random points.

Generally the frequency is doubled between two consecutive functions and each function is hence known as an *octave*. The factor by which the amplitude is decreased is in Terragen defined as *persistency*, a value of 0.5 gives good results.

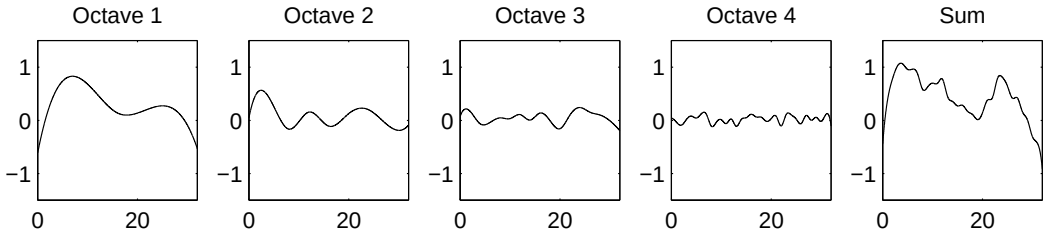


Figure 15: Perlin noise generated with four octaves.

The noise is easily extended into higher dimensions, two and three dimensional noise is constructed by assigning random values on a regular grid or lattice respectively.

Since we are dealing with terrain it is natural to construct a two dimensional texture representing height.

Texturing Now that we have constructed the terrain we would like to assign different materials to it, such as rock, snow and grass. As we explained earlier we use an extra texture stage to mask out where different textures should be used and it is these textures that we would like to create. We do this by introducing two constraints to where a material can exist, a slope constraint and an altitude constraint. Each material is assigned a minimum and maximum altitude and slope where it can exist; we could for example say that snow can only exist if the altitude is greater than 1000 meters and if the slope does not exceed 30 degrees (in which case the snow will slide of).

3.2.3 The Paint Tool

Since a normal landscape contains hundreds of thousands, even millions, of vertices it would be very tedious and time consuming to construct a landscape by adjusting each of them individually. We clearly need a tool that can manipulate the terrain on a larger scale while still maintaining the ability to adjust the smaller details. After looking at the terrain editor for a game called *Earth 2150* [13] and talking with some of the artists that would be using the program we opted for a paintbrush analogy. This way of working is currently making a great impact in the 3D modeling community by the introduction of *Maya Artisan* [14] which also works in this fashion.

By making strokes with the mouse in the 3D view of the editor the user can adjust the height of the landscape. The brush can also be used to paint textures on the landscape in an intuitive manner. The paint tool is very flexible and has several adjustable properties, such as size and sharpness, which are described in detail in the user manual.

4 Results and Discussion

From the beginning our goal was to implement a competitive landscape engine that would be capable of rendering immense landscapes in real time. Since some type of LOD algorithm obviously seemed the way to proceed, we made a test implementation with a ROAM like algorithm and benchmarked different variations. It soon became very apparent that the speed wasn't acceptable. As we came further into the development cycle we had to reconsider whether to use ROAM at all. Some kind of LOD algorithm was necessary, the sheer amount of triangles we would be tossing at the rendering API would be staggering otherwise.

So we simply moved the triangulation phase from the rendering phase to the precompilation phase, thus removing the big CPU hog from the engine. In doing so we accepted a few more triangles than absolutely necessary, but on the other hand we gained the time to render a whole bunch of them and still have time left to spare. As an illustration, the unoptimized ROAM algorithm pushed about 0.36 MTri/s and the algorithm using precomputated static LOD steps very quickly pushed about 5-6 MTri/s, see Fig. 16 and Fig. 17 for two pictures from our implementation.

The nature of the ROAM algorithm makes it an unsuitable choice for any application that needs to take advantage of hardware geometry acceleration.

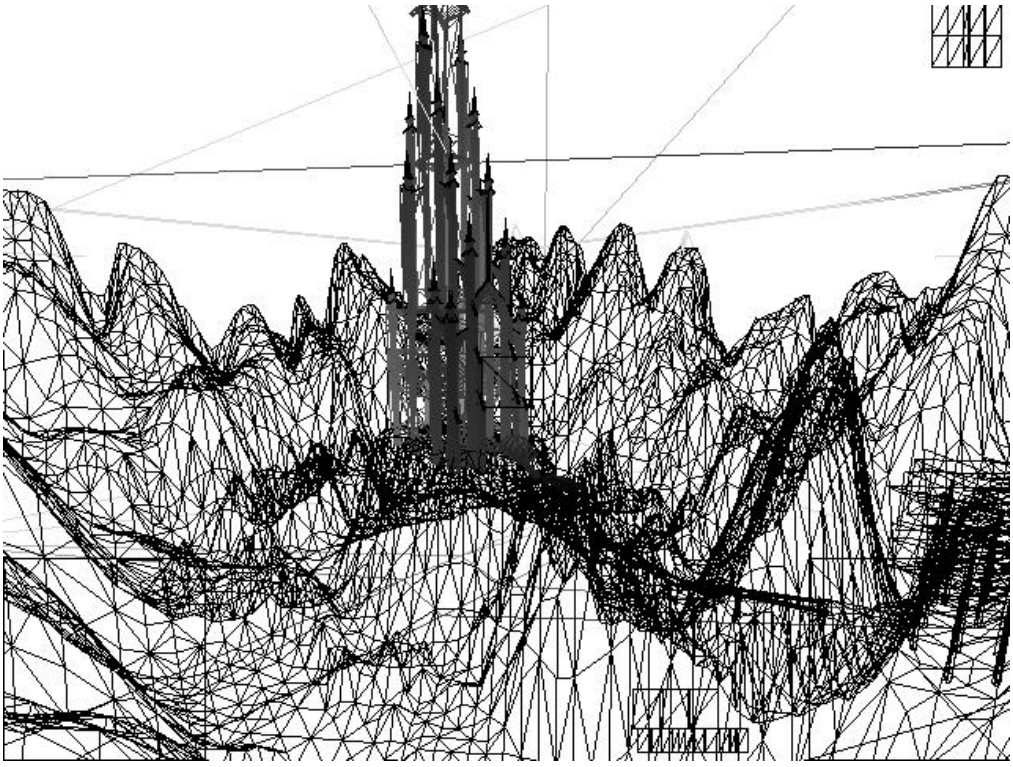


Figure 16: A sample view of a landscape in wireframe mode.

4.1 Related work

Epic Games [18] develops the Unreal Engine, which in version 2 also incorporates a landscape engine. They've solved the texturing problem the same way as we have, with alpha blended layers. There is not much information avaible but the little there is suggests that the engine does not use any LOD at all. This has the benefit that one can make better optimizations of the vcache.

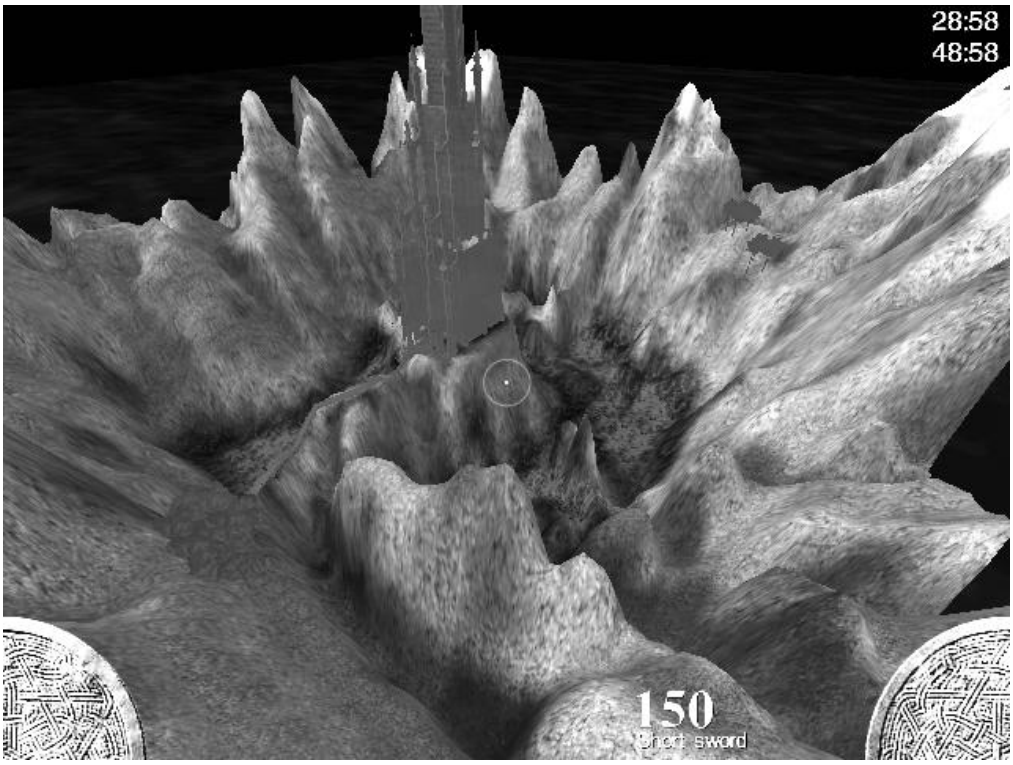


Figure 17: Same view as in Fig. 16 but with textures.

Figure 18: The landscape engine in Unreal 2

4.2 Future work

There are a number of things that we wanted to do but were unable to do because of time and/or hardware constraints. We will give a short description of some of these features and the impact they would have on the engine.

4.2.1 Non Height Field Terrain

Instead of storing the terrain as a two dimensional height field we could have stored it as a three dimensional density field. The terrain surface could then have been implicitly extracted as an isosurface at a certain density value. There exists a number of algorithms for creating surfaces from density data but the most commonly used one is described in [17]. With this approach it would have been possible to create such things as boulders, caves and overhangs.

4.2.2 Deformable Terrain

As it is now the terrain is totally static. It would have been nice to be able to deform the landscape in real time but this isn't possible for a number of reasons, the foremost being that we currently rely heavily on precomputation to construct the different LOD levels. The two dimensional representation of the terrain also severely restricts the kind of deformations that are possible, the only phenomena that can be done are craters and hills.

If we had used a non height field terrain as described above it would have been possible to drill a tunnel through a hill for example.

4.2.3 True Dynamic Lighting

Tightly coupled with deformable terrain is true dynamic lighting. It would also have been nice to be able to make a smooth transition between day and night. The authors of [3] has solved this by precomputing the light maps for a number of times of the day. To reduce the memory consumption these are stored with wavelet compression.

4.2.4 Quad Tree LOD

For patches with sufficiently high tolerance, and thus low triangle count, the cost of handling the patch itself might be more than the actual cost for the drawing of the triangles. The idea with a Quad Tree for LOD levels is to collect a bunch of patches and create a much larger patch. Naturally one have to downsample and combine the alpha masks as well to minimize the texture switches. Since the patches are far away from the observer anyway, the lower resolution will not be that prominent.

4.2.5 Disk paging system

One very real constraint in the system today is the available system memory. If one wants to create truly immense landscapes the avaible system memory will be a hard barrier. Using virtual memory isn't an option, since a) it's not available on the consoles, and b) it's rather non-optimal for this purpose. We have to use a disk paging system tailored for patches. Here we can use a quadtree as well for quick lookup of patches and load them a little before they are needed.

References

[1] M. Duchaineau, M. Wolinsky et al, *ROAMing terrain: Real-time optimally adapting meshes*, Proceedings of the ACM Symposium on Volume Visualization 1997, pp. 81-88, Oct 1997.

[2] P. Lindstrom, D. Koller et al, *Real-Time, Continuous Level of Detail Rendering of Height Fields*, SIGGRAPH 96 Conference Proceedings, pp. 109-118, Aug 1996.

[3] J. Blow, *Terrain Rendering at High Levels of Detail*, Game Developers' Conference 2000, San Jose, California, USA.

[4] S. Röttger, W. Heidrich et al, *Real-Time Generation of Continuous Levels of Detail for Height Fields*, Technical Report 13/1997, Universität Erlangen-Nürnberg.

[5] K. Perlin, *An Image Synthesizer*, Computer Graphics, Vol. 19 No. 3, pp. 287–296, July 1985.

[6] F. Evans, S. Skiena, A. Varshney, *Optimizing Triangle Strips for Fast Rendering*, IEEE Visualization '96, pp. 319–326.

[7] M.P.Fairclough, *Terragen*, www.planetside.co.uk, 2001.

[8] *Renderman*, www.pixar.com/products/renderman/products/index.html.

[9] B. B. Mandelbrot, *The Fractal Geometry of Nature*, New York, NY: W. H. Freeman and Company, 1982.

[10] E. Arkin, M. Held et al, *Hamiltonian Triangulations for Fast Rendering*, Second Annual European Symposium on Algorithms, Vol. 855, pp. 36–47, Springer-Verlag Lecture Notes in Computer Science, 1994.

[11] M. Woo, J. Neider, T. Davis, *OpenGL 1.2 Programming Guide, Third Edition: The Official Guide to Learning OpenGL, Version 1.2*, Addison-Wesley, 1999.

[12] S. Dietrich, *Optimizing for Hardware Transform and Lighting*, Presentation from the Xtreme Game Developers Conference 2000.

[13] The Learning Company, *Earth 2150*, www.earth2150.com.

[14] Alias Wavefront, *Maya*, www.aliaswavefront.com/en/WhatWeDo/maya/index.shtml.

[15] W. H. de Boer, *Fast Terrain Rendering Using Geometrical MipMapping*. E-mersion Project, Oct 2000

[16] B. Turner, *Real-Time Dynamic Level of Detail Terrain Rendering with ROAM*, www.gamasutra.com/features/20000403/turner_01.htm.

[17] W.E. Lorensen, H.E. Cline, "Marching Cubes: a high resolution 3D surface reconstruction algorithm," Computer Graphics, Vol. 21, No. 4, pp 163–169, 1987.

[18] Epic Games, *Unreal*, www.epicgames.com

A Ogier Manual

This manual will guide the user through the process of creating and editing a new landscape from scratch in Ogier. It is assumed that the user is somewhat familiar with Ogier itself, and some basic terminology.

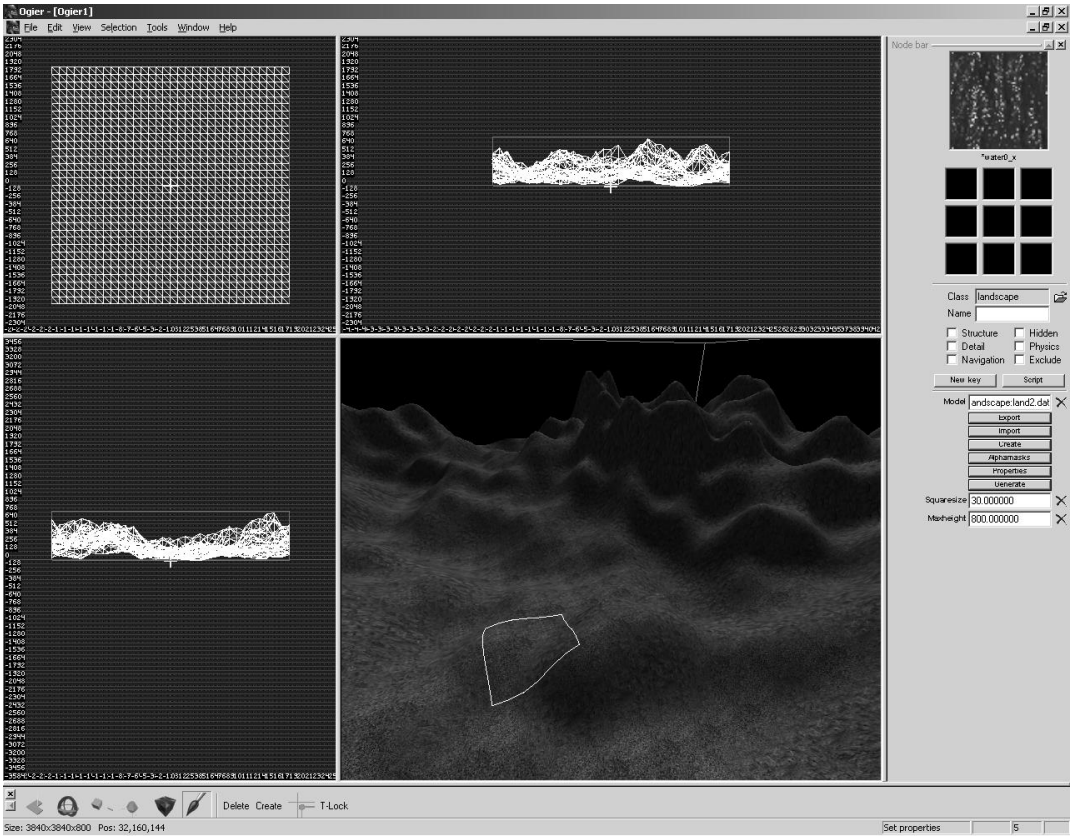


Figure 19: A sample Ogier session with a landscape.

Figure 20: Parameter for the external data file.

First we need to create a new landscape model object. Press the insert key and choose the landscape model. A default arrow will show up to indicate that no landscape has actually been

created. At this point it is a good time to set the auxiliary datafile for the landscape. This is where the world compiler will store all data needed for the engine. Add a colon ':' and the file name after the *Landscape* in the Model Edit box as shown in Fig. 20.

A.1 Create Dialog

Now we need to actually create the landscape. Press the **Create** button to bring up the create dialog. The create dialog contains a number of input fields, which are explained below.

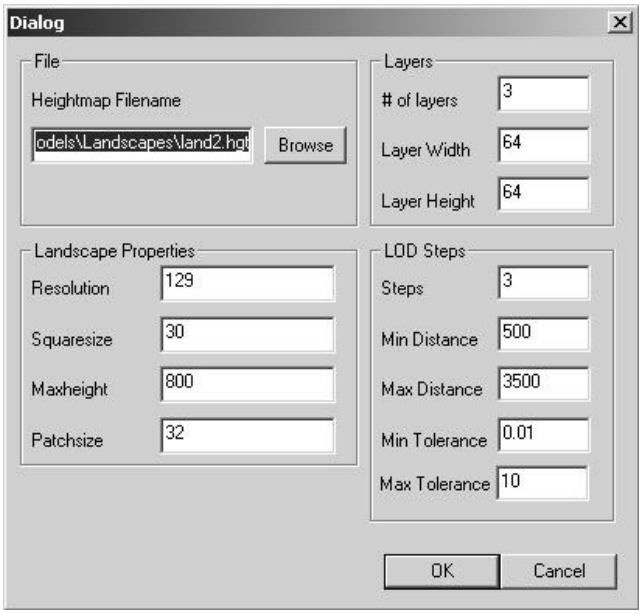


Figure 21: The create dialog.

A.1.1 Heightmap Filename

Indicates the filename relative to the game path[5] where Ogier will save/load binary data such as height field and alpha masks.

A.1.2 Landscape Properties

Resolution Specifies how many height points we want along one edge of the landscape. The number must be of the form $2^n + 1$, where n is a natural number.

Squaresize Specifies how many units there are between two adjacent height points.

Maxheight Set's a "playing ground" for the landscape, any height point exceeding the maxheight is cropped. Negative heights are disallowed.

Patchsize Specifies the size of each patch in Quadrangles. Must be of the form 2^n.

[5]The Starbreeze Engine uses a game path relative to the executable sbzengine.exe to locate all data files

A.1.3 Layers

Enter the number of layers and the initial size of each layer's alphamask. The width and height must result in a texture for each patch that is of the form $(2^n, 2^m)$, where n, m are natural numbers. See the hints and tips section of the manual for more information on how to choose the size of the layers.

A.1.4 Lod Steps

Steps	How many static LOD steps we want.
Min Distance	Where, in units, we want to use the finest LOD step.
Max Distance	Any patch further away will use the coarsest LOD step.
Min Tolerance	A guideline for the tolerance to use at the finest LOD level.
Max Tolerance	This value is currently unused.

A.2 Generation Dialog

To quickly generate a natural looking landscape, this dialog provides an interface to a Perlin generator. Simply put, the Perlin generator will produce natural looking noise between 0 and 1 and rescale it so that it fits in the range [Start Height, Height]. The combine options control how Ogier will use these values on the existing height field.

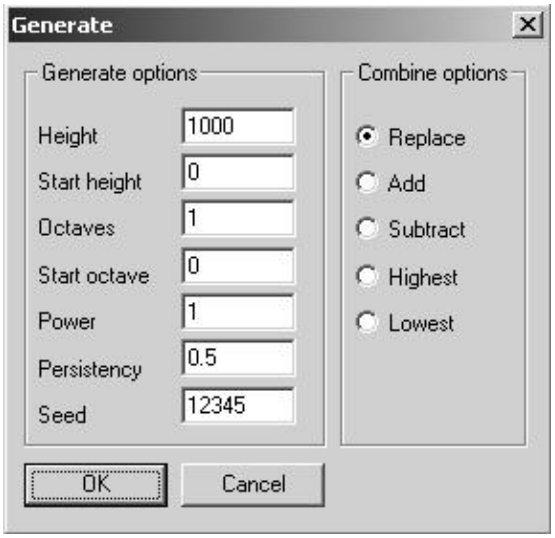


Figure 22: The generate dialog.

The rest of the values control how the height points will be generated. They are:

Octaves	As depicted in Fig.15, the higher order octaves contain finer noise, whereas the lower octaves contains the big smooth curves.
Start Octave	Don't use the octaves preceding this one.
Power	The output from the Perlin generator is raised to this power. A higher value here will result in more prominent peaks.
Persistency	Must lie in the range [0,1]. Controls the sharpness of the noise. A low value will produce smooth hills, a high will generate sharper heights.
Seed	The Perlin generator uses the system random routine, and it can be seeded with an integer to produce a different pseudo random number series.

The interface presented here is very powerful and with a little experimenting, it's rather easy to create the effects that you want.

A.3 Alphamask Dialog

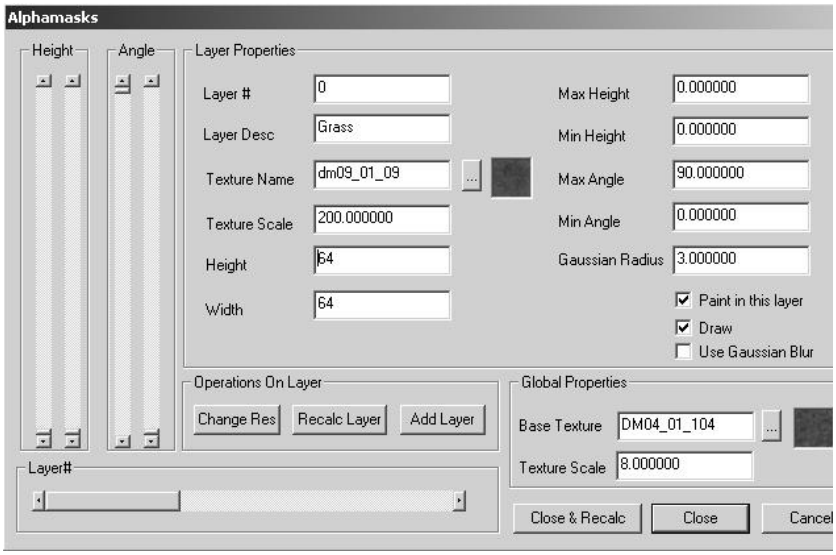


Figure 23: The alphamask dialog.

The different texture layers are basically only RGBA images stacked on top of each other. From this dialog you can control almost every aspect of them. To the right, there is the generation interface. It consists of the following edit fields:

Max Height / Min Height	If the corresponding height point is in this range, possibly let this alphamask value be visible.
Max Angle/Min Angle	If the angle between the normal of the landscape and the up vector (0,0,1) falls between these values, possibly let this alphamask value be visible.
Gaussian Radius	There's an option to apply a Gaussian blur filter to the final image, after the generator has done it's part. This indicates how many pixels that should be used in this filter.
Use Gaussian Blur	Indicates that the extra Gaussian blur filter should be applied.

There are two passes to the generation of the landscape. The first pass just goes through all the pixels in the alpha mask and checks the landscape properties at that point (height, slope) against the specified parameters and writes the appropriate pixel value at that point. The second pass is a Gaussian Blur filter to compensate for any sharp edges in the resulting alphamask. To actually make Ogier apply the the generated values, either press the button **Recalc Layer** to only recalculate the current layer's alphamask, or press the button **Close & Recalc** to close the dialog and recalculate all layers. By recalculating a layer, all changes made with the paint tool are lost.

Besides the generation, there are several properties for each layer that can be adjusted.

Layer Desc	Layer description is a pure editor related attribute, a short description of the purpose of the layer, so that the user does not have to remember each layer's meaning based upon their numbers.
Texture Name	Which texture to use in this layer. The browse button on the right will bring up the standard Ogier texture browser so the user can select the appropriate texture for the layer.
Texture Scale	Specifies how many times the texture will repeat on each patch. 1 means that the texture will be mapped to fit onto one patch.
Height/Width	Indicates the dimensions of the underlying image. These values must be of the form 2^n. Changes to these values must be confirmed by pressing the button **Change Res**. All pixel data in the current layer are lost by pressing **Change Res**.
Paint in this Layer	Selects the current layer to be the destination of the paint tool.
Draw	Toggles wether this layer is visible in Ogier.

A.4 Paint Tool

The paint tool is activated by clicking on the paintbrush icon in the toolbar. By pressing the key 'p' on the keyboard the user can bring up the paint tool dialog, see Fig. 24.

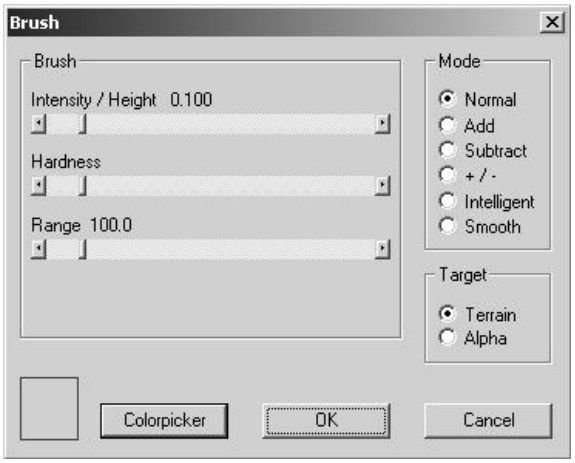


Figure 24: Paint tool parameter selection

The dialog is divided into three parts, brush, target and mode. The target part has the following settings:

Terrain	This indicates that when the paint tool is used it will modify the height of the terrain.
Alpha	This indicates that when the paint tool is used it will modify the alpha mask of the layer selected in the alpha mask dialog.

The brush dialog has the following settings:

Intensity/Height	In alpha mode this is the alpha value that will be used. In terrain mode it is the fraction of the landscapes max height that will be used.
Hardness	This indicates what the intensity profile of the brush should look like. The profile is defined as $(r/Range)^{Hardness}$.
Range	This indicates the size of the brush in Ogier length units.

The different options in the mode section has somewhat different meanings depending on whether terrain or alpha is selected as target. This is their meaning in terrain mode:

Normal	The height of the landscape will be replaced by that of the brush.
Add	The height value of the brush will be added to the landscape.
Subtract	The height value of the brush will be subtracted from the landscape.
+ / -	Left mouse button will activate add mode, right will activate subtract.
Intelligent	Not defined for terrain, the tool will fall back on normal mode.
Smooth	Slowly smoothes out the landscape in the area defined by the brush.

The meanings in alpha mode are:

Normal	The alpha value of the landscape will be replaced by that of the brush.
Add	The alpha value of the brush will be added to the landscape.
Subtract	The alpha value of the brush will be subtracted from the landscape.
+ / -	Left mouse button will activate add mode, right will activate subtract.
Intelligent	The brush works as normal but will at the same time clear the layers above the current layer.
Smooth	Slowly smoothes out the alpha masks in the area defined by the brush.

It is also possible to specify a color that will be used when painting in the alphamasks by clicking on the color picker.

When we return to the editor it's now possible to paint heights or textures in the 3D view. The tool is activated by holding down CTRL and the pressing the left mouse button. Unless the brush is set to + / - the user can pick heights with the right mouse button. A white square indicates the area where the painting will take place.

A.5 Import Dialog

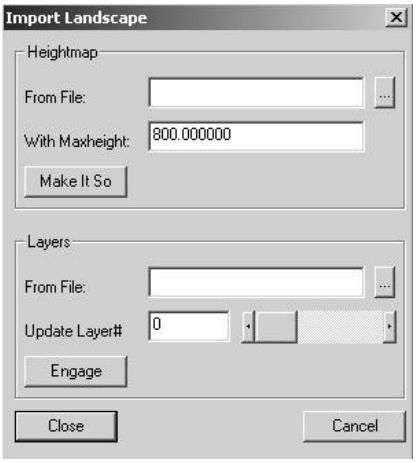


Figure 25: The import dialog.

One can import height maps or layers from standard TGA pictures. The height map import will suffer from the limited range of a 8 bit grayscale image. The "With Maxheight" field specifies the scale factor to use on the incoming height field. Each pixel is scaled with $fMaxHeight/255$.

A.6 Export Dialog

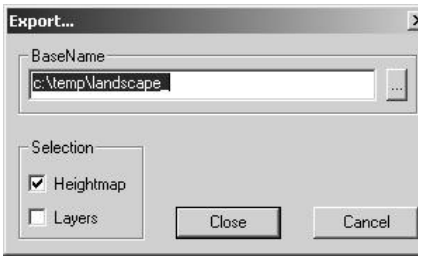


Figure 26: The export dialog.

This saves the height map and the layers as images, the height map as a 8-bit grayscale image and the layers as RGBA images. The input dialog specifies a path and a prefix for the final files. The final picture files will be appended with a UNIX suffix to identify them as heightmap and layers.

Each height point will be rescaled to fit within a single byte, and will thus loose precision.

A.7 Hints and Tips

Since the engine divides the landscape into smaller patches and manages them individually, one has to take this into consideration when choosing the texture sizes in Ogier. All the sizes refer to the landscape as a single unit, i.e. no extra calculations are made to compensate for the patches.

E.g. if we create a landscape with the dimensions 129 by 129 height points and a layer with texture size 64x64 and specify that each patch will be 33 by 33 height points, each patch will have a texture size of 16x16.

Texture compression is disabled for the alphamasks, and thus the sizes of the texture does matter a lot more than for the detail textures.

The actual tolerance used in the world compiler is $T = \frac{MinTol}{MinDist}$ for the finest LOD level. It is desirable to make T less than 1 or at least not too big since it will result in very coarse triangulation.