Internal Report 2013–11 August 2013

Universiteit Leiden
Opleiding Informatica

GPU Based Generation and Real-Time Rendering
of Semi-Procedural Terrain Using Features

Simon Zaaijer

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Generation and real-time rendering of terrain is a complex and multifaceted problem. Besides the obvious trade-offs between performance and quality, many different generation and rendering solutions exist. Different choices in implementation will result in very different visuals, usability and tools for generation. In this thesis, a fast and intuitive terrain generation method based on sketching is proposed alongside a method for real-time generation and rendering of a detailed mesh using the GPU.

Contents

1 Introduction

1.1 Overview

A common agreement in terrain generation is that a combination of a random distribution and model-based, user-provided or previously attained data is necessary for a seemingly realistic and customizable environment. A purely procedural approach will quickly seem artificial or uninspired, while only using model-based or user-provided data will lack detail without a very high generation time or data requirement. A common solution is the use of features [1] [5] [9], in combination with a random generator such as simplex noise (another citatation here).

This semi-procedural approach leaves many options for both the manner of procedural generation as well as model-based or user-provided data. Another important choice is the type of surface that is to be computed. These can generally be divided into three categories: mesh generated from heightmaps [5], mesh generated from a point cloud using a triangulation scheme [1], and a mesh generated from voxel data [8].

Terrain generation and rendering can be useful in any area where a plausible outdoor environment is needed, without caring too much about how it will exactly look. This applies to entertainment, particularly games, as well as architecture and various forms of simulation.

An ideal solution is capable of creating highly detailed terrain with ease, allowing the user to specify the layout, all the while being able to render in real-time. Realistically, every implementation has different requirements, leading to different points of focus and design choices. Intuitive generation for real-time rendering is a relatively unexplored field, mainly due to the difficulty of efficient rendering schemes and the hard time requirement. It is, however, a large feature of many commercial rendering engines.

1.2 Previous work

The work on terrain generation can be subdivided into several categories, based on the methods employed. Procedural methods use a combination of fractals, simulation methods revolve around thermal and hydraulic erosion models, and sketching methods rely heavily on user input.

1.2.1 Procedural methods

The first propositions for generating the surface of a terrain through a fractal started as early as 1982 with random midpoint placement [6]. This is still a popular technique because of its simplicity and relatively high image quality. An alternative fractal method is given by use of Perlin noise [19]. A large downside of fractal based methods is that they generally provide very little control. While the noise can be regenerated, it cannot be controlled directly to place features at desired locations.

More recently, methods have been developed to overcome this limitation. [21] presents a large-scale generation system based on fractals, where the user can interactively modify the shape of the terrain. [2] begins by computing ridge lines and a rivers network after which random midpoint placement is used to generate the resulting mesh.

1.2.2 Simulation methods

Simulation models attempt to generate more realistic and geographically correct terrain, by employing models based on various kinds of erosion. The input for these models is commonly purely procedural, where the physical operations result in a realistic terrain. One of the first models to use erosion is given by [13], where a river network is generated and used to erode the surrounding terrain.

For a long time, erosion models were too expensive to be run on a large-scale terrain, but recent advancements have made large-scale applications more feasible. Methods such as [17] and [26] use the GPU to improve performance and provide a visualization of the hydraulic erosion process. A somewhat different approach is taken by [14], where an erosion method based on Smoothed-Particle Hydrodynamics is used.

A simulation methods based on hydrology is presented in [9] and generates a river network based on user sketches. This approach is very different from erosion based models. Where the simulation of erosion starts with a fractal terrain and carves rivers into it based on its height distribution, this approach using hydrology begins by generating rivers and fills in the rest of the terrain using procedural methods formed around these rivers. The result is excellent and can be used on a large scale.

1.2.3 Sketching methods

Simulation models do not provide a lot of improvement for the user input in terrain editing. While they can provide realistic terrains, it is difficult to get the desired result. Methods based on sketching and other user input aim to resolve this shortcoming.

The method given in [4] combines a sketching method with procedural generation, where brushes are provided that directly create plausible terrain features. This results in a realistic looking terrain, while giving the user complete freedom over its shape. [1] allows the user to provide a sketch of the terrain with several feature types. The terrain is generated using a novel midpoint displacement method according to the user input. Interestingly, the resulting mesh is not based on a fixed grid, but uses an incremental Delaunay triangulation algorithm.

In [7] users can control terrain generation interactively using a sketching interface. The work in [5] provides a generation method using a diffusion algorithm on the GPU. The user can specify vector based features that specify terrain elevation, surface angle or roughness, to be combined with a procedural component.

1.3 Thesis goal

The focus of this thesis is on a large-scale terrain, such that for any reasonable choice of viewing point, a detailed and seemingly endless terrain can be observed. Common limitations such as blockiness on close ups and a visible boundary are to be avoided as much as possible. Furthermore, the terrain should be very simple to generate and the layout easy to control. To accomplish this, a real-time generation solution is presented, using the GPU for performance.

To provide a simple and intuitive generation tool, the user is presented with a very simple painting tool to provide information on where features should be placed. In several layers, where each layer represents a feature type such as mountain or river, the user can specify where this feature should be generated using shapes. Each drawn shape can be given varying settings to specify, for instance, mountain height or a multitude for hills.

This thesis proposes a feature generation model by sketching features with the aim of an intuitive and simple user guided generation system. Secondly, a tiling model for generating and rendering a large surface mesh is proposed with the aim of overcoming large memory requirements and rendering costs while allowing a high level of detail in a large terrain.

1.4 Implementation overview

The terrain solution presented has been implemented using a combination of OpenGL and OpenCL, where OpenGL is used for rendering and OpenCL is used for parts of the terrain generation. Interoperability between these two is very common and is facilitated by both API's. As a result, a computer with a relatively modern GPU is needed to run the program.

2 Mesh tiles

2.1 Tiling rationale

As mentioned earlier, the terrain examined in this thesis should feel like a seemingly endless environment, with sufficient detail for a closer view. Due to storage and rendering limitations, the terrain mesh cannot be generated and shown as a whole at a reasonable desired level of detail. Should we require a terrain of 4 by 4 km, while allowing detail of up to 25 cm, then even when storing only a single floating point number for height, the data storage requirement is already close to 1 GB. This means a level of detail system is needed, which cannot be directly derived from fully detailed data.

Especially if the requirements on size and detail were to increase, a possible solution is for the mesh to be generated on demand during rendering, rather than being generated ahead of time. This allows for varying levels of detail and a large terrain size, while keeping storage costs low. Features could be used as a specification for the generation of the mesh.

Due to the large amounts of data to be generated, the GPU is a sensible platform to perform this computation on. As the GPU is best suited to parallel computation in large datasets, the generation should also be made as parallel as possible. Therefore, the mesh should be split into parts, where each part has a number of vertices to be computed in parallel. Each part should then be generated at the required level of detail.

To further simplify the process, the vertices can be aligned in a grid. Levels of detail can then be made possible using a quadtree datastructure, where each leaf is a square mesh called a *tile*. Each tiles consists of $n \times n$ quads and $(n + 1) \times (n + 1)$ vertices. Since a quadtree subdivides a node in four, the resolution of a tile doubles for increasing levels of detail, but the amount of quads and vertices remains constant.

For the vertices in a tile, the generation of a triangulation to form the mesh is trivial, but leaves the problem of connecting adjacent tiles. When tiles of different levels of detail connect, they will always share a number of vertices along their connecting border. Due to the nature of the quadtree and the levels of detail associated with it, the tile with a higher level of detail will have a number of vertices in between each overlapping pair of vertices. If the level of detail is just 1 level higher, ie. the resolution is double that of the other tile, and n is even, there will be exactly one vertex in between each vertex of the connecting tile. If the generation of a vertex depends solely on its position in the grid, the resulting computation of overlapping vertices will always be the same, so that these will line up nicely.

The remaining problem of connecting adjacent tiles then lies in the difference of levels of detail, resulting in the in-between vertices for the cases where the adjacent level of detail is lower. This can be solved by ensuring the in-between vertices are an exact interpolation of its neighbors, but even then this difference in triangulation is dangerous. Also, it would require the GPU to correct the vertices in a separate step, taking extra time. A better solution is to use a special triangulation near the border of the tile, where the in-between vertices are left out to form a larger connecting triangle (see figure 2.1). This requires no changes to the generated vertices and can easily be accomplished using indexing.

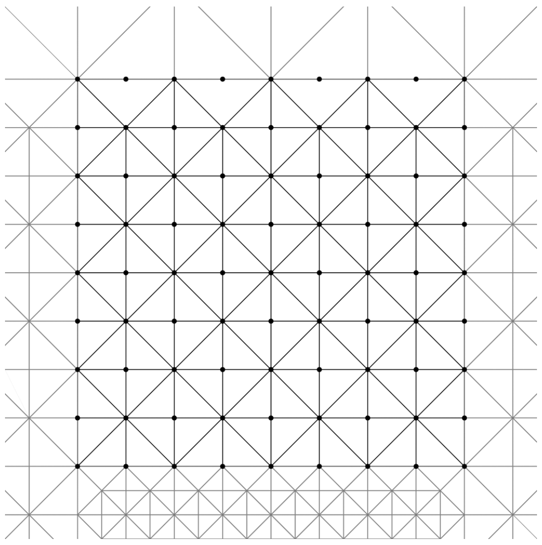


Figure 2.1: A tile of 9 by 9 vertices connected to a tile of lower level of detail (above) and a tile of higher level of detail (below).

In modern rendering, the triangulation is often already a separate entity from the vertices themselves, where the triangulation simply connects vertices by index. Thus, with a constant amount of vertices in each tile and the required connections to different levels of detail, we will need 2^4 different triangulations for the different adjacency cases, a very manageable number. The disadvantage of this solution is that it can only connect tiles that differ at most one level of detail. This will require an increased resolution in places.

Before computing individual vertices, an overall layout of the terrain must be generated. The GPU is less suited to this kind of task, as it is much harder to parallelize. Thus, the layout of the terrain is made on the CPU, from where this data is transferred to the GPU for the final generation. By using features for this information, the data transfer between the CPU and GPU, a common bottleneck in graphics applications, can be kept to a minimum.

2.2 Detail levels

For creating the nodes in the quadtree, a metric for the required level of detail is needed. This metric should determine whether the level of detail of a node is sufficient or the node should be subdivided into four nodes of a higher level of detail.

A good level of detail system is crucial for the performance of real-time terrain rendering. Rendering everything at the same level of detail will either lead to very high rendering times or reduce the visual quality below acceptable levels. While it is possible to get away with a lower level of detail in the distance, changes between these levels of detail should not be too obvious. Also, some finer details that are still visible in the distance can easily be hidden by a too coarse tessellation.

Since different systems will have different rendering capabilities and different terrains can have different demands on detail, the level of detail system should be easy to scale. With a setting for the overall tessellation level, it will still be possible to render a lower quality version of the terrain on an older system, while a high-end system could use a higher quality version.

Not every part of the terrain needs the same level of detail. In the case of a flat stretch of meadow with a mountain in the distance, the tessellation of the mountain should be higher than that of the flat meadow. The presence of features that affect the generation of a tile should influence its level of detail. To account for this, a detail value is chosen for each type of feature, where based on nearby features, the maximum value of those features is chosen.

Combining all this, the metric for the detail levels computes the maximum length of a single quad in the mesh of a tile by:

$$s_{max} = \sqrt{d} \cdot \frac{1}{T \cdot F}$$

where s_{max} is the maximum length of a quad, d is the distance to the tile,

T is the tessellation setting and F is a value based on nearby features. Thus, a lower distance and higher values of T and F result in a finer tessellation.

2.3 Implementation

The mesh of each tile is stored as a Vertex Buffer Object, or VBO, which is essentially an array holding the attributes of all vertices. Each VBO is uploaded to GPU memory, where it can be filled with to be generated data. The tiles are stored in one large set and are referenced from a quadtree, containing the layout of levels of detail in the terrain. Each tile also has a matching position and size attribute to identify it.

Before rendering a frame, the terrain mesh is updated to meet the new required levels of detail. To do this, the quadtree is rebuilt entirely. For each previously used tile, a matching leaf is sought using the position and size attributes. If found, the tile is referenced from the leaf and the tile is labeled as used. Otherwise, the tile is labeled as unused and can be reassigned. Then, for each leaf without a referenced tile, a tile with an unused label is picked and reassigned to match the new position and size. If no more unused tiles remain, new tiles are created as needed.

As long as the camera does not move and the terrain does not change, no updates to the tiles are required. When the camera does move, certain tiles may require a higher or lower level of detail. In this manner, only some of the tiles are updated at a time during camera movement. Ideally, this updating process is spread out equally across frames to prevent sudden peaks in computation time. Simply updating tiles as needed does not seem to result in a very unbalanced computation time across frames.

Using this method, VBO's are created only at the very beginning of the program and can be reused for different tiles. For specific terrains, the maximum number of tiles used could be determined beforehand, but this does not seem necessary in practice.

The creation of the triangulations used for tiles is done only once at the beginning of the program. For all 16 cases of adjacency with varying nearby levels of detail, a separate triangulation is created. Such a triangulation is represented using an Element Array Buffer which, like a VBO, is simply an array of data. In this case it contains indices of vertices. In triangle rendering mode, each triplet of indices represents one triangle. The triangulation pattern used can be seen in figure 2.1.

3 Mesh generation on the GPU

3.1 Generation using OpenCL kernels

Kernels are functions in an OpenCL program, that can be executed on an OpenCL device, such as the GPU. They are the entry points that can be called from the rest of the program. Each kernel is executed in parallel by a number of threads, where one or multiple thread identifiers can be used to divide a task across threads. In the relatively simple use case of this project, the thread identifier is used as an index to the vertex to be computed, so that the vertices are computed in parallel using a single thread each.

Much like regular functions, kernel functions are passed parameters. One such parameter is an array bound to a VBO, where the data can be directly accessed.

3.2 GPU-based Simplex Noise

Besides the functions directly depending on feature data, a lot of the generation is done by using procedural noise. While various classes of procedural noise exist [15], a very good match for terrain is gradient noise, such as Perlin noise [22]. Instead of producing purely random values, this class of noise picks a random gradient at several points laid out in a grid. Interpolation between these points results in the output value. A gradient at each point can then also easily be computed. The space in which these points are placed can be of any dimension, where for computer graphics, 1-dimensional through 4-dimensional noise is practical. Using 4D noise, it is possible to generate a random thickness for fog, while smoothly varying the values across time.

Simplex noise is an alternative to traditional Perlin noise, that uses a simplex grid (triangular in 2D), as opposed to the cubic grid (square in 2D) used in perlin noise [22]. At a slightly higher initial complexity, this reduces the number of interpolations needed in higher dimensions, making it faster to compute. Also, the triangular grid can be more difficult to perceive, giving

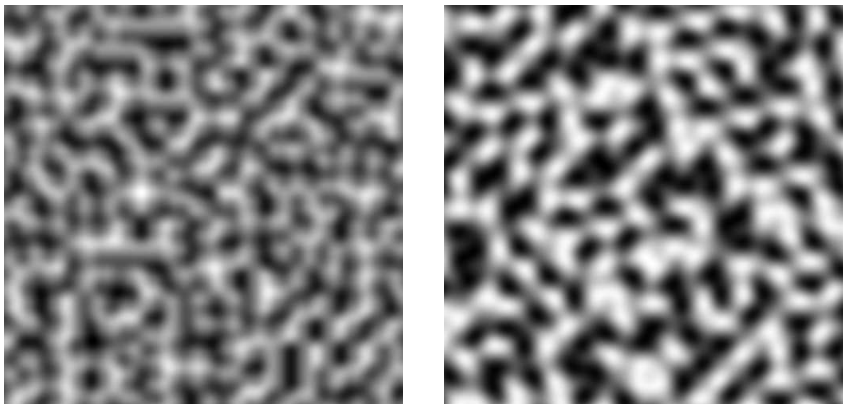


Figure 3.1: Left: traditional Perlin noise. Right: simplex noise.

the noise a more natural look. Figure 3.1 shows a comparison between the two.

The noise values are computed using a hash-function based on a table of integers. For the 2D case, the hashing function has the form $H_{i,j} = P[P[i]+j]$. P is a randomly permuted array of integers 0 through 255, often referenced as a permutation table. This hashing value is then reduced to modulo 16 to pick one of the predefined gradients. The specified i and j usually correspond to coordinates in the computed space and can be offset and scaled to provide different results.

A GPU-based implementation of the algorithm, designed to run in a shader, generally involves a lookup texture. This texture of 256 by 256 pixels, gives a random gradient at each pixel, by evaluating the hashing function beforehand. The shader then simply performs a lookup in the texture, using the position as a wrapping texture coordinate. The idea here is that shaders are not very good at integer arithmetic and array access, but a texture lookup is relatively cheap.

For an OpenCL implementation, using a texture is a much less sensible choice, since integer arithmetic and array access are much less costly. Instead, the permutation table and gradients are implemented directly as constant arrays, avoiding any texture lookups and thus, any external data requirements.

Directly using gradient noise is not that useful. It is commonly used as a summation of several octaves of noise, each with increasing frequency and decreasing amplitude. This is known as Fractional Brownian Motion [16]. The resultant fractal shape can also be manipulated using various functions, to get a wide range of very different effects (see figure 3.2).

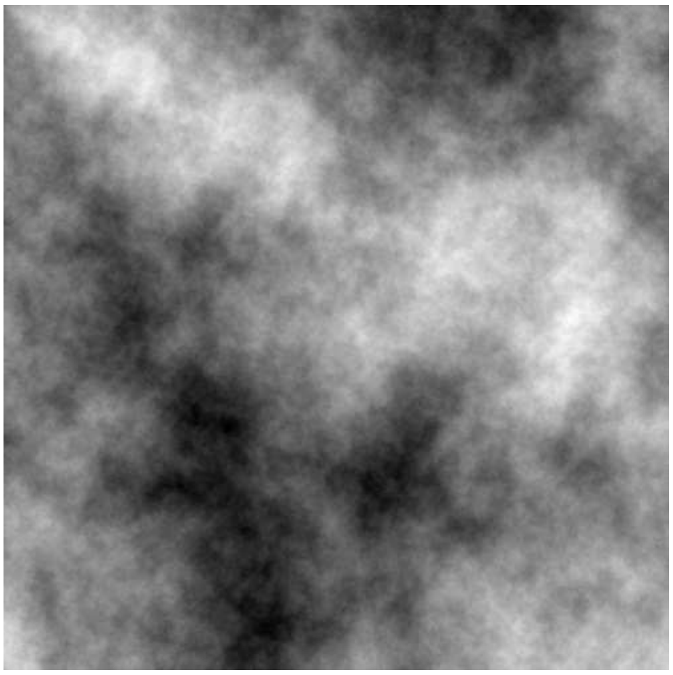


Figure 3.2: Addition of several octaves of noise leading to a fractal.

3.3 Feature representation

Features are represented by a 2 dimensional position, size and strength attribute. Mountain features also have a peakiness attribute. These attributes are passed as an array of structs to the OpenCL kernel. Since not all tiles need to know all features, a subset of nearby features is found for each generated tile, based on the feature position and size.

The effect of features on the generated mesh consists of two factors. First, the features directly adjust the terrain height with a factor dependent on the distance to the feature position. This is done using a combination of simple cubic functions, effectively giving features a circular area of effect. Secondly, a fractal is mixed in with the newly added height, to give each feature a distinct and natural look.

3.4 Spline features

Besides regular features, which are circular in nature, another supported feature type are spline features. Spline features connect several feature points to form a spline curve, which affects the terrain along its path. Because of its simplicity, a Catmull-Rom spline is chosen as the spline implementation. Like regular features, the effect of the spline is based on the distance towards it.

Computing the distance to a spline is not entirely trivial. As detailed in [27], a common approximation is to use Newton's method. This involves specifying a function $D(s)$, the distance towards the spline at position s. The distance to the spline is the minimum of $D(s)$, where its derivate $D'(s)$ equals zero. Newton's method works iteratively to find a root of a function, a point where the function equals zero. A better approximation of the value s^* can be found from a previous iteration using the formula:

$$s^{*,m+1} = s^{*,m} - \frac{D'(s^{*,m})}{D''(s^{*,m})}$$

The initial approximation $s^{*,0}$ can easily be found by taking the nearest control point on the curve. To handle the cases where multiple control points of different parts of the curve are nearby, the three closest control points are selected as initial candidates. Each can then be further refined to find the actual closest point.

Although this should work in theory and handles many cases very well, this does not converge in all cases. [27] Especially in the manner of usage here, even a single incorrect distance is instantly noticeable, producing very obvious artifacts. When combined with a quadratic approximation as proposed in [27], this will still not result in a working solution for catmull-rom splines. This is possibly due to the non-continuous second derivative in between connected spline segments.

Instead of attempting to find a root of $D'(s)$, a course first approximation can be found by searching for a root in $D(s)$ directly, using then formula:

$$s^{*,m+1} = s^{*,m} - \frac{D(s^{*,m})}{D'(s^{*,m})}$$

Although this will not converge completely, the rough approximation allows for a finer second step using just the first derivative itself. While this is reasonable solution, it is difficult to get a stable implementation and will still produce minor artifacts in some cases.

A much safer alternative is to simply compare the distance to a lot of nearby point on the curve. By iteratively decreasingly the spacing between candidate points and taking the closest point at each iteration, a good approximation can be reached. This method requires a lot more computation than Newton's method, but will always find a correct closest point.

3.5 Generating the terrain

Different features have a variety of effects on the terrain. All features use a combination of fixed functions and functions using noise to generate the desired result. There are two often recurring functions, including a cubic function of the form:

$$\text{Cubic}(x) = 2x^3 - 3x^2 + 1$$

This function can be used to get a smooth curve based on a distance value in the range $[0, 1]$.

The other function is a noise based function, generating a fractal by summing several levels of gradient noise:

$$\text{NoiseOctaves}(p, s, a, p, o, n) = \sum_{i=0}^{n-1} (G(p \cdot s \cdot 2^i) + o) * a * p^i$$

Based on a position p and initial scale s, the fractal can be controlled using an amplitude a, persistence p, offset o and number of octaves n.

Hills are generated by using the cubic function as a base, where several noise octaves are added to a constant as a multiplier. Figure 3.3 shows the result of the cubic function and the resulting hill when the noise levels are added. Multiple hill features can also be combined to make a larger hill.

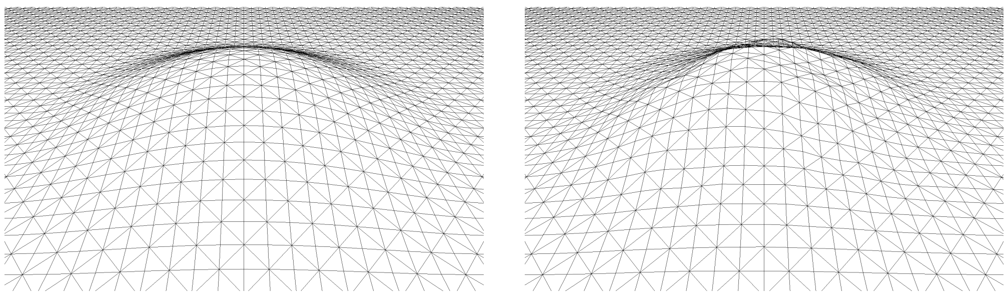


Figure 3.3: A wireframe view of a hill feature generated using only a cubic function (left) and with added noise levels (right).

Mountains are created by a sum of a cubic with another smaller cubic function placed over top to give it a more steep appearance. A similar sum of cubics is done to provide a multiplier for the fractal noise, but the second cubic is given a sharp tip by modifying the input distance, to give the mountain something of a peak.

While mountains and rivers only add height, rivers modify height along a wide range to bring the surface level to a constant height around the river. This is done by using a wide cubic function as a weight to bring the height to a constant level. After that, a narrow cubic function is used to carve the actual river.

Fractal noise can be used alongside other functions in a variety of ways. With a bit of creativity very interesting results can be achieved. An example are sand dunes that appear in a dry environment. Since sand dunes are often created by the wind and are thus commonly oriented in a similar direction, a suitable basic shape to model these is a sine wave along a horizontal direction. By taking the negative absolute value of the sine wave, peaks can be created to form the ridges of the dunes. Several noise levels can then be added to offset the phase of the sine, to make the dunes a little bit wavy. By multiplying the whole by yet another noise fractal, the dunes will appear to begin and end at somewhat consistent intervals. The effect is shown in figure 3.4

Although more sophisticated use of fractals and other functions would be desirable, it is important to remember that a more complex generation kernel will lead to higher generation times. When the generation time becomes too high, the frame rate while moving through the terrain may suffer severely.

3.6 Normal computation

The primary task of the generation kernel is to compute the height values of the terrain. However, to shade the terrain, an object space coordinate system is required at each vertex as an orthonormal basis, consisting of a tangent, bitangent and normal vector. This basis shows the local orientation of the terrain. There are two ways to compute the tangent and bitangent vectors, where the normal vector can be computed from the former two using the outer product: $N = \frac{T \times B}{||T \times B||}$.

The first method is to compute the derivatives, or slope, along with the height, both of the fractal noise and off the cubic functions used. From the derivative, computing the tangent and bitangent vector is trivial. While this only requires some differential calculus in most cases, problems arise when the height function is not continuous. This is an often desired effect in terrains as, for instance, sharp ridges on top of slopes. In these cases, the difference between resultant normal vectors can be too large to interpolate correctly, leading to black spots along the edge.

Another problem of this method is aliasing. When the frequency of the fractal noise is too high compared to the sampling frequency, aliasing will

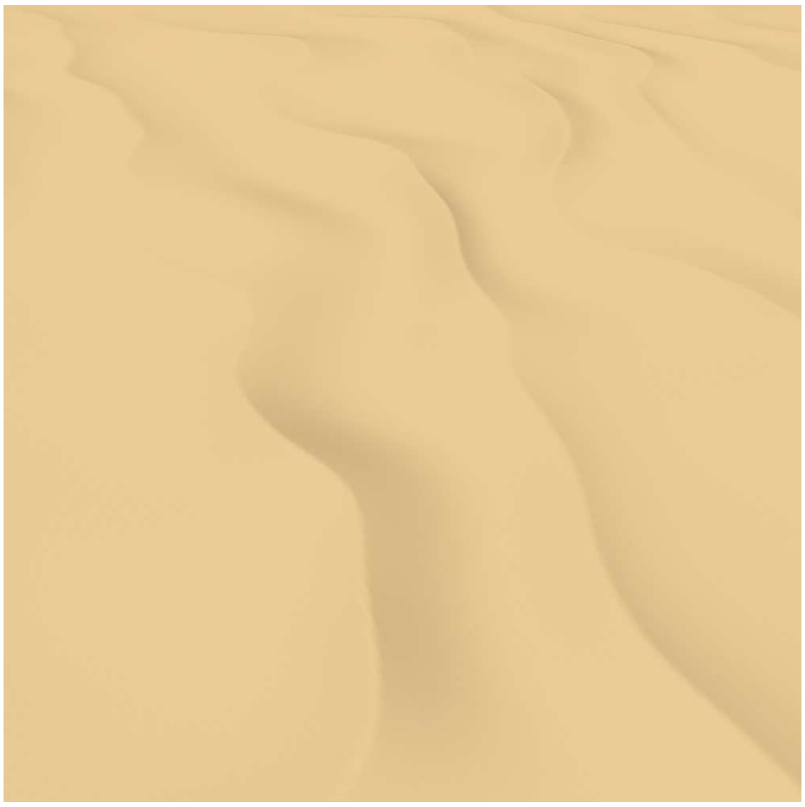


Figure 3.4: A combination of a sine wave with fractal noise can result in sand dunes.

occur, where the derivatives no longer provide an accurate description of the surface orientation. While this not a serious problem for the height itself, it is more visible in the shading because of noisy normal vectors.

An alternative method is to compute the tangent and bitangent vectors in a separate pass, after all the height values have been computed. These can be computed easily using the height values of nearby vertices. This is the common approach for other heightmap based terrain generators. This avoids the aliasing problem and provides a softer appearance of sharp edges.

The latter approach requires some extra effort at the edges of the tiles however. To avoid differences between normals of overlapping vertices in adjacent tiles, it is not sufficient to compute the normals of edges using the vertices inside the tile. Therefore, a border of a single vertex is added outside each tile. This effectively increases the tilesize to $(n+3) \times (n+3)$ vertices, for a tile of $n \times n$ quads. As the tilesize commonly has an n of at least 64, the overhead of this border is negligible.

4 Feature generation

4.1 User-drawn feature shapes

There are several possibilities to allow a user to specify the layout of a terrain. Each has its own level of control, from the very low-level approaches, where the user specifies almost everything manually, to very high-level approaches, where most of the generation is automated. Since a lot of modern terrain editing tools lean towards the low-level approach to allow a user to get exactly the terrain they want, this paper seeks a higher-level solution. The goal is to make a credible terrain, following some approximate layout provided by the user.

A common method of specifying the layout of a terrain is the use of features. A feature gives some abstract information on the shape and possibly the surface type of the terrain. This can be further defined in several ways, such as control curves [5], heightfield brushes [4] or ridge lines, rivers, lakes etc. [1]. In this project, control over three types of features are given to the user: mountains, hills and rivers.

Instead of requiring the user to place features on the terrain manually, the user is provided with a basic set of drawing tools. On three separate image layers, where each layer represents a feature type, simple shapes can be drawn using a circular brush. For each shape, depending on the feature type, some properties can be specified, such as the height of a mountainous area or the number of hills in a hill area.

Features are then generated from these feature shapes in a two-step process. While the features attempt to follow the guidelines set by the user, the user drawn shapes are not an exact representation of what will be generated. Moreover, the generation will fill in parts of the terrain that are not drawn by the user, according to several density settings, even when the user has drawn nothing at all. The drawing is designed to be a very rough view of where everything should be.

4.2 Using feature shapes

The first step of generating features from the feature shapes is the generation of intermediate user features. These will form a point-based representation of the shapes and translate the properties of each shape.

During the drawing of the features, a labeling algorithm is used to separate the feature shapes within a layer. For each shape, the area and weighted center are computed, along with a Euclidean distance map, showing the distance from each pixel to the edge of the shape. From the multitude property, specifying the number and thus the average size of user features, a minimal required distance from the border is computed. Then, a set of candidate pixels is found based on the distance map, where a pixel must either be a local maximum or have a distance greater than the average size. The distance is stored as the weight of the pixel.

Then, a simple dart throwing algorithm generates user features at random, where the position is chosen randomly from the set of candidate pixels and the size depends on the pixel's weight. Dart throwing requires each user feature to be sufficiently far from the other chosen user feature positions. The algorithm stops when the total feature area is high enough or dart throwing fails to find a suitable position after several attempts.

4.3 Two-stage generation

These generated user features provide a very reasonable estimation of what the user has drawn, but these are not satisfactory for direct use in the generation of the terrain. A second stage is required as a more global feature generation solution.

The first reason is that these features follow the shapes specified by the user with decent precision. In fact, the shapes are still clearly visible in the final terrain. Since we only want the user to provide a rough sketch, the exact shapes the user draws should not matter so much. These are merely guidelines for the terrain to follow.

Secondly, the user is not required to draw everything that is generated on the terrain. This mainly means that the system should be able to decide on placements itself. Even when the user draws no mountains whatsoever, if the specified density of mountains is not zero, mountains should be generated at some suitable position.

Finally, the shapes drawn by the user may contain contradictions. If a river is drawn crossing a mountain, this may mean that a river should spring near the mountain top. It may also mean that the user intended for two mountains to be placed alongside the river, with steep cliffs on either side of it, or even two mountains placed further apart, split by a narrow valley. Either way, it certainly does not mean the river should cross directly through the mountain or run overtop it. The system should be able to cope with such situations by choosing a reasonable solution.

4.4 Feature placement

In the second step of generating features, the user features are used as a guideline to place the final features to be used in generation. Features are placed in the most desirable position, where the user features give a strong preference.

To determine the preference of a position, a density metric is used, where the total density of features within a radius around a specified position is computed. This density metric can be used for both user features as well as final features and gives insight into the amount of nearby features, but it can also be used to find a mean property, such as size or height.

The density metric uses a cubic kernel, where features further from the searching position have a lower weight than nearby features. Each feature is also weighted by its size. Besides the Since the number of features can potentially increase greatly, it is efficient to use a datastructure for nearest neighbor queries, such as a kd-tree.

When determining where to place a feature of a certain type, nearby user features of that type will be desirable, whereas existing final features of the same type may not be desirable. Effects between different types are also often reasonable, for instance, for a mountain feature, nearby river features are not desirable. With this kind of system, rules can be specified and adjusted for various kinds of features, in this case for mountains, hills and rivers.

5 Materials and rendering

5.1 Rendering basics

5.1.1 Rasterization

Real-time 3D rendering applications commonly use rasterization as their core rendering technique. Rasterization translates 3D geometry to screen coordinates, after which the covered pixels of the screen are found to draw the shape. The color of each pixel can then be computed freely.

The geometry is provided in the form a mesh, consisting of vertices and triangles that connect these vertices. The surface orientation, which is important for most rendering styles, is stored as normal vectors. These vectors are commonly not defined per triangle, but are instead defined per vertex as the average of the normal vectors of adjacent triangles. During rendering, the normal vectors are interpolated across each triangle to give the illusion of a smooth curved surface. This technique is commonly known as Gouraud Shading. [11]

Rasterization is widely supported by hardware, where the rendering of a surface is controlled by two or more programmable shaders. A vertex shader is responsible for performing the projection that transforms a vertex position from the object space it is defined in, to screen space. This projection commonly depends on the position and manner of projection of a camera object from which the scene is perceived. The result is a 2D coordinate mapping to a position on the screen with an added depth value. It also passes on any properties of the vertices that are needed for determining an output color. Using the output coordinates of each triangle, the hardware rasterizes the triangles to result in a set of pixels, or fragments, each with interpolated properties of the vertices. A fragment shader is then responsible for outputting the color of each fragment, based on the interpolated properties, such as the normal vector, and other data concerning, for instance, light sources and surface color.

The resultant color is finally outputted to a render target. A render target can either be the back buffer, a buffer that is associated with the screen, or a separately defined buffer to be used later in rendering. In the form of a frame buffer object, the result of a render buffer can be used as input for a shader. A common application of this is to define a render buffer to store the light intensity perceived by the camera. In a separate drawing stage, the render buffer is outputted to the back buffer, where each pixel translates the matching light intensity value to a color value. This process is known as tone mapping.

5.1.2 Shading

Shading is the process of assigning a color value to a pixel or fragment. In realistic rendering applications this is a two step process, where first the light intensity is computed, which is then translated to a color value using a tone mapping operator. Tone mapping addresses the limitations of display devices in outputting a wide range of luminous intensity. Many methods exist to find a seemingly correct color associated to a light intensity based on perception. [20]

Computing the light intensity is a very complex subject. Light is emitted from one or more light sources in the scene, subsequently reflected, refracted and absorbed to be perceived by the camera. A basic approach is to only compute a direct reflection term from a light source to the camera. More sophisticated methods also take into account multiple reflections and refractions, but such methods can become very costly. A common approximation is to use a separate ambient term to model any indirect light, which can be made more accurate by the use of Ambient Occlusion, such as [12].

Computing reflections is done by defining a BRDF, a Bidirectional Reflectance Distribution Function. Such a function aims to model the optical characteristics of the surface of a material by returning the amount of light reflecting off a surface in a given direction. The main parameters for a BRDF are the light vector $\mathbf{l}$, the normal vector of the surface $\mathbf{n}$ and a eye vector $\mathbf{e}$. These are defined as unit vectors from the point of reflection to the light source, the outside of the reflecting object and the camera respectively. Color parameters for diffuse and specular reflection are also a vital part in creating recognizable materials. Other parameters are dependent on the chosen model, although many employ a roughness parameter and a refractive index. Besides the BRDF, some radiometry must be computed for the transport of light.

The diffuse color of a material is often stored in a texture map. This is an image where a mapping of the geometry to the image is defined by the use of texture coordinates. The image is sampled at the specified coordinates to return the color of the material at the associated position. Textures can also contain other data, such as normal vectors, displacement heights, specular coefficients or opacity values.

One of the simplest physically based shading models was introduced by Blinn [3]. It defines diffuse and specular reflection as follows:

$$I_{\text{diffuse}} = k_d \, I_L \; \max(0, \mathbf{n} \cdot \mathbf{l})$$

$$I_{\text{specular}} = k_s \, I_L \, (\mathbf{n} \cdot \mathbf{h})^{\mathbf{g}}$$

where $\mathbf{h} = \frac{\mathbf{l}+\mathbf{e}}{||\mathbf{l}+\mathbf{e}||}$, k_d and k_s are colors for diffuse and specular reflection respectively, I_L is the light intensity and g is a glossiness factor.

5.2 Surface Types

A real environment has various kinds of soil and rock, each with its own optical characteristics. Different surface materials are usually implemented by using different texture maps for the color of diffuse reflection.

To show these materials on the terrain, it is important to have a specification on what material is found at various points on the terrain. In this system, a weighting value is used for each surface material. These are computed during the mesh generation and are specified per vertex, from where they are interpolated across the surface. Although surface materials are not expected to change very quickly and the resolution of the vertices is often quite high, this limitation can result in unexpected appearance or disappearance of small patches of materials.

As with the features, various surface types and rules for each of these are imaginable. In the current implementation, steepness and presence of mountains or rivers has an influence on the material. This will often lead to unnatural material distributions. More elaborate systems, for instance, based on groundwater levels, soil types and erosion, could create very realistic material distributions.

5.3 Triplanar texturing

In regular 3D models, a texture coordinate is provided for each vertex. These coordinates map parts of the texture to parts of the model. Generating such texture coordinates is not a trivial task and is commonly the work of a 3D artist with the use of several tools. For more complicated shapes, a perfect mapping of texture coordinates is often impossible, causing seams where the coordinates are not continuous. Textures are created to match the specified coordinates and adjusted so the areas around seams are of the same color.

Terrain is a slightly easier case than the average 3D model, since it can often be reasonably approximated by a horizontal plane. Thus, the texture coordinates can be found as a projection of the vertex positions onto this plane. This will cause problems for steeper parts, where the texture will then be stretched too far. Since a reliable solution to reduce such stretching is not feasible, a different solution is needed.

Triplanar texturing greatly reduces the effects of stretching by using three separate projections, one along each axis [8]. The final texturing result is a weighted blend of the three projected textures. The weights are computed as a vector using the normal vector of the surface, where a smaller angle between the projection axis and the normal results in a higher weight. A blending angle can be chosen to specify how wide the blending range should be. The computation is made easy using the smoothstep function in GLSL (the numbers are the cosine of 35 and 55 degrees respectively):

```
vec3 triplanarweight = smoothstep(vec3(0.57357644), vec3
    (0.81915204), abs(normal));
```

5.4 Texture Splatting

With triplanar texturing and a weight for different surface materials, the manner of blending between materials must be decided. Blending between different textures across a terrain is commonly known as texture splatting and can be done in various ways. The simplest way is to use a blend directly, with the result as the sum of the weights multiplied by the material values. Although this gives a smooth transition, thanks to the interpolation of the weights across vertices, the result is too smooth to seem natural.

An alternative blending technique is given by [18]. The idea is to use an extra channel in the diffuse texture to encode a height. Blending is then performed to bring out the higher parts. This results in a much more natural transition.

The technique is described using two materials, but when using more materials at once, it will require finding the height of all these materials first. In this system, this would require a large amount of temporal bookkeeping, especially with triplanar texturing. To avoid this, the materials are instead layered on top of each other, where a layer is blended with the result of the previous layers using the height value found so far. This greatly simplifies the blending method with no visual loss of quality.

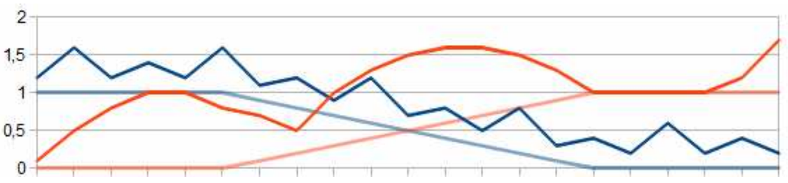


Figure 5.1: Example height values of two materials. Weight values (lighter lines) and added material height (darker lines) are shown.

The height value of the material is added to its blending weight to create the height used for blending (see figure 5.1). This is compared to the previous height value, where the new weight of the layer is found by another smoothstep:

```
float weight = smoothstep(height - 0.4, height + 0.4,
    materialheight);
```

Of course, the quality of the result is strongly affected by the height values in the textures.

5.5 Detail textures and normal maps

Every surface material consists of a diffuse texture map, a detail texture map and a normal map. The combination of the diffuse and detail maps is used to reduce obvious repetition of the textures while maintaining detail at a closer distance, whereas the normal map gives more depth to an otherwise seemingly flat surface.

Detail texture maps can be blended using either a multiplier to the base diffuse texture map, or by taking an average. In some cases, such as a dirt material, choosing the same texture for the detail map as the diffuse map and applying a rotation to the texture coordinates of the detail map before averaging, helps to eliminate repetition (see figure 5.2). This ensures the texture looks smooth from a distance.

In other cases, such as rock, detail textures can be used to add larger patterns, while keeping the finer rock texture (see figure 5.3). This keeps the rock from looking bland from a distance, while also reducing repetition.

Since there is a tangent basis generated in the terrain mesh, normal mapping can be applied without too much overhead computation. The three channels in the normal map encode the vector components along the tangent, bitangent and normal vectors of the tangent basis. The result is an offset normal vector which can be used in the lighting function to simulate a more detailed surface.

5.6 Lighting

For the illumination of the terrain, direct sunlight is combined with an ambient term from the sky. The ambient term adds a constant factor to the diffuse reflection and a very small factor to specular reflection. The sunlight is reflected using a model based on the work by Trowbridge and Reitz as

Figure 5.2: Reducing repetition of a single material using a detail texture. The left and middle images are heavily contrasted to show the difference more clearly. Left: just the base texture. Middle: a detail texture mixed with the base texture. Right: final non-contrasted view.

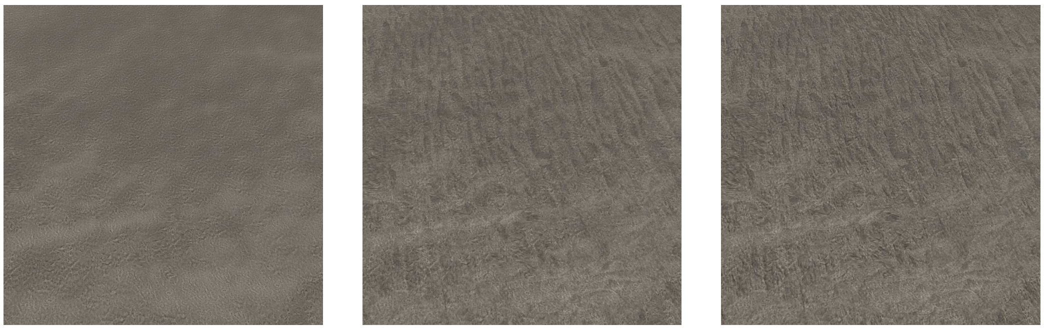


Figure 5.3: Adding far away detail to a material using a detail texture and normal map. Left: just the base texture. Middle: a large-scale detail texture multiplied by the base texture. Right: detail texture and normal map for detail in lighting.

shown in Disney's BRDF Explorer [23]. This model can be used to closely approximate natural materials. Since there are several physical models that can produce high quality visuals and are close to reference measurements, the choice of a specific model is largely an artistic one. The model chosen here is relatively easy to compute, while it results in a natural look.

The model by Trowbridge and Reitz defines diffuse and specular reflection as follows:

$$I_{\text{diffuse}} = k_d \, I_L \, (1 - F) \; \max(0, \mathbf{n} \cdot \mathbf{l})$$

$$I_{\text{specular}} = k_s \, I_L \, F \, \frac{1}{\pi} \frac{g^2}{((\mathbf{n} \cdot \mathbf{h})^\mathbf{2} \cdot (\mathbf{g^2} + \mathbf{1}) - \mathbf{1})^\mathbf{2}}$$

F is the Fresnel function that determines how much of the light is reflected directly and how much will become diffuse reflection. A good approximation to F is given by Schlick:

$$F_{\text{Schlick}} = c_{spec} + (1 - c_{spec})(1 - (\mathbf{l} \cdot \mathbf{n}))^\mathbf{5}$$

where c_{spec} controls the amount of specular lighting at zero incidence.

6 Blue Noise Sampling on the GPU

6.1 Overview

Blue noise sampling is a method for generating an evenly distributed set of points in a predefined space, such that the points exhibit blue noise properties. One method for generating such a set is the generation of a Poisson Disk. The points on a Poisson Disk are required to be at least a distance of r apart. Compared to other sampling methods, a Poisson Disk usually results in superior image quality. [28]

Many algorithms have been developed for generating a Poisson Disk, with varying computational costs. The solution presented in [28] provides a great improvement by using parallelization on modern graphics hardware. This makes on the fly sampling possible by generating several millions samples per second. An example of blue noise is given in figure 6.1.

6.2 Usage

Generating samples with blue noise properties is an important task for many kinds of graphics algorithms, such as ambient occlusion, generating point sets from geometry and ray tracing. For terrain generation, it can be used to generate locations for various types of foliage. Because of the high throughput in modern implementations, it is possible to generate real time positions for strands of grass. This aligns well with the real-time generation of the surface mesh. Using these positions, it is fairly efficient to render grass models by geometry instancing.

6.3 Implementation

The implementation of blue noise generation is written for OpenCL, where several modifications are made from the algorithm in [28]. The goal is to generate a set of points which can directly be used for instancing. The

Figure 6.1: Blue noise generated on the GPU.

original GPU-based algorithm uses shaders for computation and frame buffer objects for storage of the samples.

The main idea of [28] is that the sampling space can be subdivided into cells, such that cells that are sufficiently far away from each other, can be processed in parallel. Any remaining artifacts in the frequency spectrum are eliminated using a multi-resolution approach with a grid partitioned random order of cells.

The most difficult part in generating a poisson disk is an effective distance comparison of a random candidate point to other nearby points. The implementation in [28] uses a frame buffer object to find nearby points through their pixel positions, where each pixel maps to a cell in the multi-resolution grid. This implementation uses a large index array, that functions much like a frame buffer object.

Instead of outputting the results to a frame buffer object, the generated points in this implementation are written sequentially to an array. Since points are generated in parallel and points may be rejected, it is not trivial to output them to an array sequentially. This can be done by using an atomic counter:

```
int pid = atomic_inc(numpoints);
vstore2(newpoint, pid, points);
```

`pid` is the index of the point in the output array, which is a unique value for each thread. The atomic counter increases the value of numpoints by one and returns the old value. `vstore2` then stores the point `newpoint` in array `points` at the specified index.

To choose a random position for a point in a cell, a random function is needed. Since the GPU does not have native random number generation, a separate method is required. A method based on a random seed cannot be easily used, since the number generation should also work in parallel. For the application of procedural generation, this random value should be based on its position, so that it will return the same position each time a point is generated.

A cryptographic hash as proposed in [25] produces such a randomization and is also the method employed in [28]. A different method based on a cipher is presented by [29] and has better performance. With this method a number of iterations can be specified to improve randomness, but with only three iterations the results are already without visible repetitions or artifacts.

6.4 Performance

The performance of the OpenCL implementation has been tested for various values of r and k, where r is the minimal distance between samples and k is the number of attempts to generate a random sample within a cell. The area within which the samples are generated is 512 by 512 units. All measurements were done using a GTX 580 graphics card and record the average generated samples per second over 10 runs. The results are given in table 6.1.

	$r = 1$	$r = 2$	$r = 4$	$r = 8$
$k = 4$	8.03M	6.31M	2.00M	0.57M
$k = 8$	5.51M	4.72M	1.40M	0.40M
$k = 12$	4.29M	3.86M	1.09M	0.32M

Table 6.1: Average number of generated points per second (M is millions) for different values of r and k.

Note that higher values of k reduce the performance, but will output less samples. For applications in procedural terrain, this should not be a big issue. Also, a lower value of r and thus a larger amount of generated samples reduces the relative kernel overhead, improving the performance.

7 Results and discussion

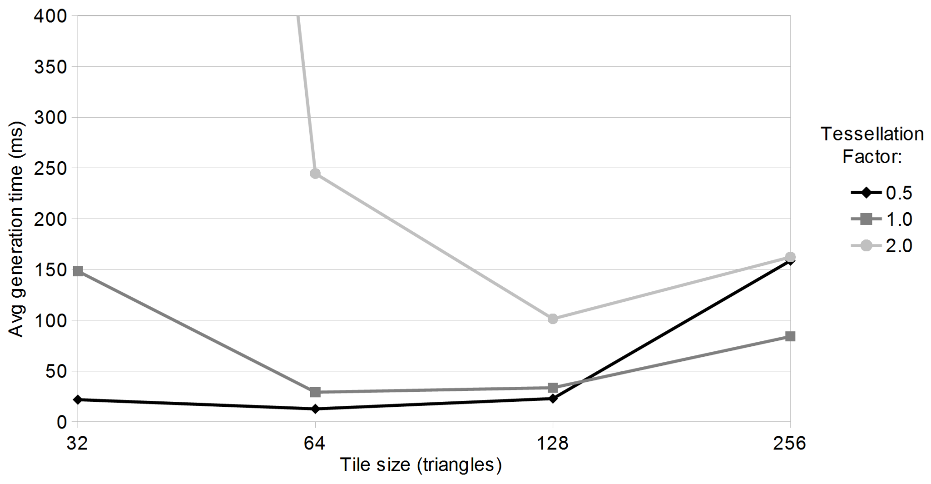
7.1 Generation performance

The performance of the tiling system is an important metric in verifying its worth. The running times of the generation kernel should allow real-time rendering. Since the OpenCL kernels use the same data as OpenGL uses for rendering, they cannot be run in parallel, even if a second GPU would make this worthwhile. Therefore, to maintain a decent framerate in an interactive application, the kernel should use as little time as possible. Commonly, real-time applications attempt to maintain a 60 or 30 frames per second goal, leaving 16 or 33 ms of computation time per frame. Depending on how heavy the rendering is, only a small part of this should be used for generation.

Generation of tiles occurs when the terrain is first loaded and occurs gradually during camera movement. The time taken during initial generation should not be too large, but is certainly not required to be done within one frame. The time taken to update the terrain during camera movement is of more interest. These should be relatively consistent and and should not be too high.

Generation times have been collected from a randomly generated sample terrain with a camera flying overtop at a constant speed. The terrain has a size of 4096 by 4096 meters and has a maximum detail size of 0.2 meters. The generation times are the total kernel execution time per frame and have been measured during 10 seconds, along with the initial generation time for the terrain. The tile size, ie. the length of a tile in triangles, and tessellation factor, ie. the multiplier for determining the level of detail, are varied across different runs. All measurements were done using a GTX 580 graphics card.

Figure 7.1 shows the average and maximum generation times per frame, where the average is of frames when some updating was needed. This gives some idea of how long an update of the mesh takes. Ideally, the times found here should not exceed the part of the frame time reserved for terrain generation. It also provides an idea of what values are optimal for the tile



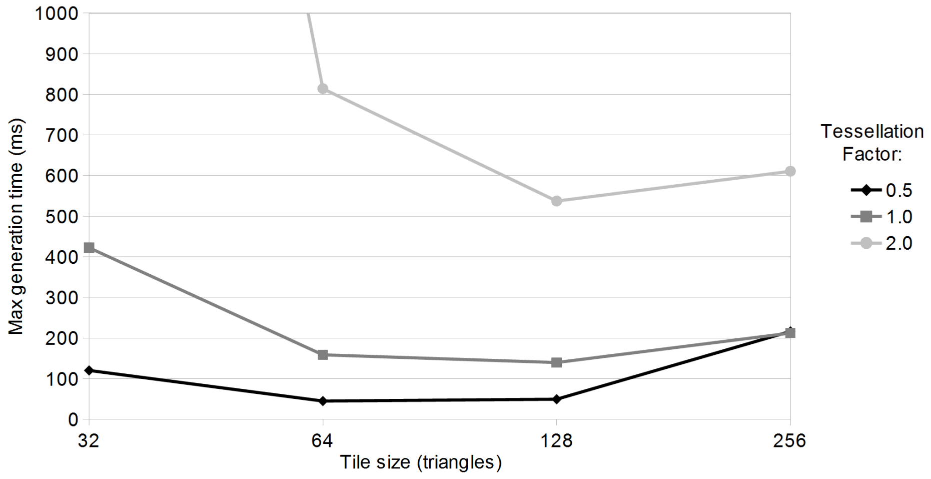
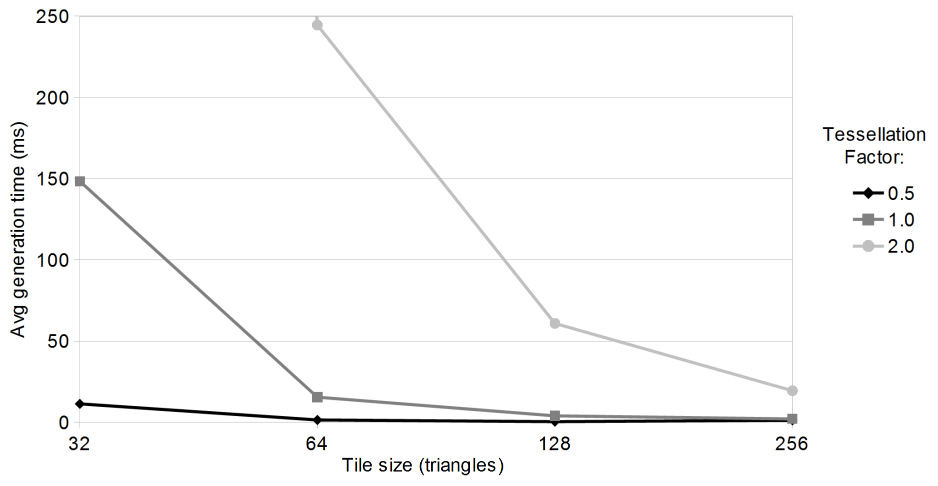


Figure 7.1: Average generation time per frame when the mesh is updated (top) and maximum generation time per frame (bottom) for various tile sizes and tessellation factors.

size and tessellation factor.

A lower tessellation value clearly results in much lower generation times. For a tile size of 64, the generation times remain well below 45 ms, with an average of 13 ms. While these results are still not great, they can be considered sufficient for real-time performance. With a higher tessellation factor, performance quickly drops with a best average of around 100 ms for a tessellation factor of 2.0.

A good of choice of tile size is also an important part in achieving good performance. Smaller tiles allow for more diversity in levels of detail and more gradual updates, but result in a larger kernel execution overhead and generally more updates. Larger tiles decrease the kernel execution overhead, but can be prone to sudden peaks in the number of updates along with less diversity in levels of detail. Figure 7.1 clearly shows the trade-off in choosing the tile size, where either 64 or 128 triangles seem to be good values.



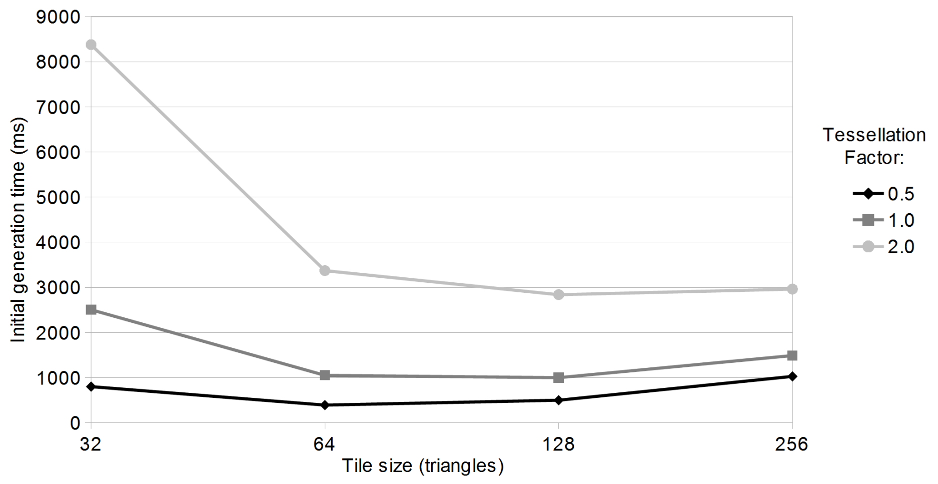


Figure 7.2: Average generation time per frame across all frames (top) and initial generation time of the terrain (bottom) for various tile sizes and tessellation factors.

The average generation time across all frames and the initial generation time are given in figure 7.2. There is a clear gap between this measure of the average and the one given in figure 7.1. This difference gives some insight into how frequent the updates of the terrain tiles are. For larger tile sizes, where the updates are less gradual, the average across all frames is much lower. This shows the sudden peaks in generation time which contribute only little to the average amongst the many frames without updates.

This measure of the generation time across all frames gives a much more positive view of the performance of the system, even coming below a 1 ms average for a tile size of 128. This measure is not of much use however, since for any frames where no updating is performed, the time left over is not used. This does give an idea of the possible performance of the system if the workload of the updates is spread out across frames more evenly. Using a more intelligent updating scheme, it should be possible to decrease the maximum generation time closer to the average.

The initial generation of the whole terrain takes almost half a second at best. For a higher tessellation factor, it can take around 3 seconds. While this amount of time is not a problem for the initial loading of the terrain, a similar update is needed when the viewpoint is suddenly moved to a different position. Although an intelligent updating scheme could begin loading the required tiles in advance, this is not always possible in real world interactive applications. Therefore, this high initial generation time can be seen as a significant weak point of the system.

7.2 User evaluation

To test the usability of the feature generation and sketching, a user study has been done. Several users of various levels of expertise have been asked to create two different terrains in both the terrain editor presented in this thesis, and in the Unity terrain editor.

Unity is a well known game engine that is favored by indie developers for its ease of use, low cost and cross-platform building capability. The terrain editor in Unity is a good example of how terrain modeling is commonly done in commercial applications. The user is given a brush tool with which the terrain can be raised, lowered or smoothed. Additionally, a texture painting brush is available to modify and easily blend surface textures.

In both editors, users were presented with a flat grass area, ready for editing, and were given some basic instructions on how each editor is used. The users were then asked to model a valley with a river, followed by a desert with some distant mountains. Half of the users were first asked to use the

Unity editor to create both terrains and the other half started with the editor from this thesis. Afterwards, the users were asked to fill in some questions.

The users were asked to compare the editor from this thesis to the Unity editor by rating the following on a scale of -4 to 4 (eg. Very difficult to very easy):

- Ease of use: the overall usability of the editor, higher is easier.
- Desired result: the ability to get the desired result, higher is easier.
- Quality of result: the attainable quality of the result, higher is better.
- Overall impression: the overall impression after using the editor, higher is better.

They were also asked to comment on whether something was missing from the editor or whether they were unable to create something they would have liked, and were asked for any final comments.

The results of the rating questions, along with the expertise level of the users, is given in figure 7.3. Users were generally quite pleased with the editor and found the results to be fairly good. A lower score is given to the desired result, showing that users had more difficulty in creating exactly what they wanted.

The comments given by the users agree with this result quite clearly. The most common remark is that the sketch drawn by the user did not match the generated terrain. Some users were slightly frustrated at not being able to control the generation. Also the inability to control the size of the rivers was considered a missing feature.

One user also noted the inability to change the materials used for the terrain. Where in Unity materials can be freely painted, the materials in this editor are decided by the generation method.

Many users remarked that they would like to change or manipulate the features after generation. Clearly, more control over the exact placement and shape of features is desirable.

Finally, some users commented that the controls could be more intuitive, while others stated the editor was easier to use than Unity.

7.3 Tiling model

The subdivision of the mesh into separate tiles for generation on the GPU has several problems that require a change of the system to solve. These problems mainly concern resolution problems, connections between tiles with changing level of detail, the inability to perform erosion and performance.

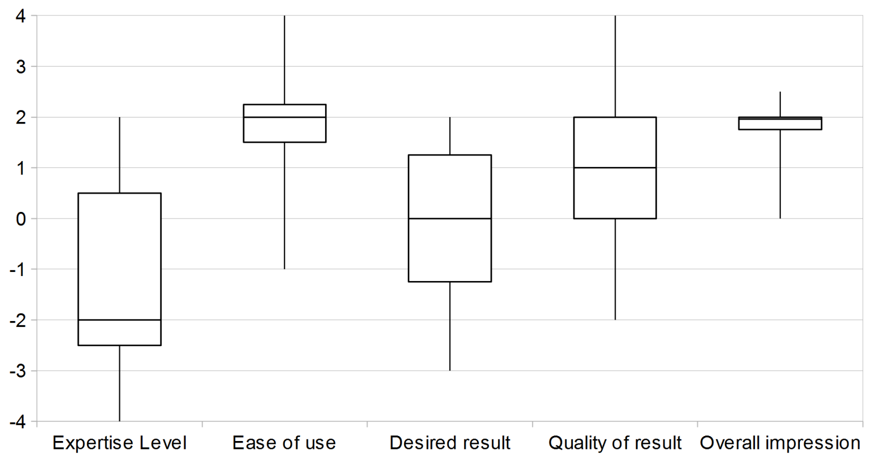


Figure 7.3: Several questions answered by users about the usability of the editor presented in this thesis, in relation to the Unity terrain editor. The boxes show the first and third quartiles and the median. The values range from -4 (very low) to 4 (very high), where a value of 0 means the editors were about the same.

7.3.1 Geometry resolution

One of the main issues with a fixed grid for the vertices is that, unless a very fine tessellation is used, finer features of the terrain will disappear when viewed from afar. This is especially problematic for small rivers and streams, which may be just one or two vertices wide from a distance. Raising the tessellation to solve this, will result in too many vertices to maintain a decent frame rate. The same problem occurs on a rocky surface, where an otherwise rough surface will appear more and more smooth when moving further away.

This issue has been reduced by the adaptive subdivision model, where presence of detail in the form of nearby features increases the subdivision of tiles. In practice, this does little to solve the issue, as the rendering costs are still too high, even with the adaptive subdivision. Also, the restriction where connecting tiles can only differ by a one level of detail makes it difficult to locally enhance the resolution without too much overhead.

An alternative approach that would suffer less from this issue would be to use a varying triangulation. Instead of forcing the generated points to lie on a grid, points could be generated at important geometric locations, from where a Delaunay Triangulation can be generated to form a mesh.

These points could be chosen to preserve important geometric detail, even at lower resolutions. It is easy to imagine a more detailed river where the triangulation follows the river bank. Of course, care must be taken to ensure there are no triangulation artifacts and the normals are generated properly. Also, aligning parts of the terrain could then prove difficult. Should the triangulation overlap generated tiles, or should points be generated on the borders of tiles to connect them easily?

A point-based system could be much closer to a hand-crafted 3D model of terrain and could even allow the generation of overhanging cliffs and caves. Such features are not possible to generate in regular heightmap based approaches, since they can only encode a single height at each 2D position.

An advantage of the tiling model is that, while points are generated using a grid for triangulation, the position of these points can be modified freely. In this system only the height of these points is modified, but by also changing the horizontal coordinates, a lot more freedom in the generated meshes can be achieved. This could allow the generation of the above mentioned overhanging cliffs. However, the limitation of the grid does not make it practical for radical changes such as caves or larger overhangs. Also, reducing levels of detail could have a disastrous effect on such more complicated geometry, where important extruded points can disappear at a lower detail level.

7.3.2 Tile connections and changing levels of detail

Even with the modified triangulation shown in section 2.1, the connections between tiles are sometimes clearly noticeable. In cases with different levels of detail, the generation of the normals across the tiles is not correctly modeled. The border introduced in 3.6 only works when the vertex resolution is the same in the neighboring tile. Otherwise, a different point is used for the interpolation, leading to a discontinuity of the normals along the tile border.

This line accentuates another problem with levels of detail. While moving the viewpoint, levels of detail will change. Ideally, only imperceivable detail is left out to reduce the drawing cost, but for an entire terrain this is not feasible on modern hardware. Therefore, the change between levels of detail is noticeable, resulting in what is commonly known as popping.

Getting rid of these issues once again requires a finer tessellation, making it more difficult to maintain a good frame rate.

7.3.3 Erosion

Generating realistic mountain ranges and rivers using only features and fractal noise is very difficult. Although intelligent use of features makes erosion less necessary, as shown in [9], erosion still shows recognizable patterns which are difficult to reproduce.

Most erosion models, such as [14][17], require the terrain to be stored in a grid. With the tiles presented here, there is no single grid upon which these computations can be run. Although an adaption of these techniques could be possible, this would require further research to be feasible.

7.3.4 Performance

As can be seen from the measurements in section 7.1, the performance of the tiling system is reasonable, but the spikes in generation time are a difficult problem. A more intelligent updating method is needed to balance the generation workload across frames. Also, regenerating more of the terrain at once, as would be needed when the viewpoint is suddenly moved to an entirely different location, takes too long to be usable. For many applications this is unacceptable.

7.4 Feature generation

While the feature sketching system is fast and intuitive, the results are not always as expected. Furthermore, while the generated terrains may look natural, they are not very realistic.

Users had some trouble in obtaining desired results, where their sketch and the resultant terrain were often very different. While the sketching is fast and easy, the rules used for the feature generation made it hard to get the exact desired result. It has proven difficult to define rules that ensure nothing goes wrong, such as a river running through the middle of a mountain, while maintaining the freedom for the user to specify feature positions.

It should be possible to develop a more elaborate system to address these issues. The work presented in [9] does a much better job of creating a realistic terrain with certain specifications. A wider range of supported generation methods may also help to improve results where, for instance, a natural cliff can be used when a river comes close to a mountain side.

One advantage of a rule based system over geologically inspired models is the ability to create a non-realistic or otherworldly environment. The freedom offered by a rule set can create results that are not possible by means of models based on erosion or hydrology.

7.5 Texturing

Creating realistic textures for use in a large-scale terrain is a difficult task. The balance between detail on closer inspection and avoiding repetition and dullness from further away is hard to find. Besides the methods employed here, several alternatives exist, although these too have their limitations.

A fairly recent technique is the use of a so-called ClipMap [24], now commonly known as a MegaTexture. This is a very large dynamic texture map that is uploaded to graphics memory as needed. Of course, no large unique texture is directly available for the generated terrain. Instead, using this kind of technique, parts of several textures could be placed by features, to give a much more diverse textured surface. This can effectively be used as a dynamic texture splatting and decal placement method.

Another alternative is the use of procedural texturing. This can circumvent the use of texture maps entirely and also removes the need for triplanar texturing (see section 5.3). However, procedural texture generation is difficult and is generally only useful for specific types of materials. The work in [10] gives some idea of the state of the art in this field. It is especially suited for textures with small details or a light noisy structure, but has difficulty in realistically generating other types of surfaces. Also, the high computational cost for real-time rendering makes it less suited for such applications.

7.6 Future Work

Much work remains to be done to result in a complete terrain generation system. Besides extending and improving the feature generation model to give more predictable results, and improving the tiling model to increase performance, many aspects of terrain generation have not been discussed here.

Continued work on the feature generation based on user sketches could greatly improve the overall results of generation. By specifying better rules and incorporating geological systems, such as [9], the generated terrain could become much more realistic and closer to what the user has specified. Also, as users have noted in the user study, the ability to manually edit features after the initial generation gives a lot more control to the user, without sacrificing too much of the realism of the terrain or the ease of generation.

Several improvements could be made to the tiling model. By more intelligently updating the tiles as the viewpoint is moved, the generation time could be balanced out more across frames. An alternative to the grid based tessellation should also be researched more thoroughly, as it could provide a much more detailed tessellation with a lower number of triangles. The work in [1] suggests that this is feasible. The use of tiles does seem to have some drawbacks that are inherent to real-time updates of the mesh, particularly the high initial generation time. Although the ever increasing computational power could make this less of an issue in the long term, different methods may be more viable as the expectations for realistic terrain continue to increase.

Besides the generation of the terrain mesh, the generation of vegetation and urban construction is of great importance to a realistic environment. Here as well, procedural and rule based methods could be defined to incorporate them into the feature generation system. It is easy to imagine being able to sketch roads and forests or specify the number of houses in an area. Also, a system could be created for how individual trees or houses are generated, to enable a user to create believable but unique objects spread across the terrain. One could even model wildlife and their behavior in relation to the environment.

8 Conclusion

A feature generation model based on user sketches has been presented. While the method of generation is intuitive and results in a potentially high quality terrain with only minimal effort, the amount of control over placement of the features makes it difficult to use. These issues will need to be addressed before the terrain is useful in practical applications.

Additionally, a tiling model for the generation and rendering of a terrain has been proposed. Although this model allows for larger terrains to be shown at high detail, the computational costs are too high to be able to show this detail well and has other inherent problems. The results of the performance give some insight into the limitations and possibilities of real-time GPU generation, showing that, while at the moment this is not a good option for realistic terrain by itself, real-time generation on the GPU has a lot of potential.

Bibliography

[1] Daniel Adams, Parris Egbert, and Seth Brunner. Feature-based interactively sketched terrain. *Siggraph I3D '12*, page 208, 2012.

[2] Farès Belhadj and Pierre Audibert. Modeling landscapes with ridges and rivers: bottom up approach. In *GRAPHITE*, pages 447–450, 2005.

[3] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, jul 1977.

[4] Giliam J. P. de Carpentier and Rafael Bidarra. Interactive gpu-based procedural heightfield brushes. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 55–62, 2009.

[5] H. Hnaidi et al. Feature based terrain generation using diffusion equation. *Comput. Graph. Forum*, 29(7):2179–2186, 2010.

[6] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25(6):371–384, jun 1982.

[7] James Gain, Patrick Marais, and Wolfgang Straßer. Terrain sketching. In *I3D*, pages 31–38, 2009.

[8] Ryan Geiss. *GPU Gems 3*, chapter Generating Complex Procedural Terrains Using the GPU. NVidia, 2013.

[9] Jean-David Génevaux, Éric Galin, Eric Guérin, Adrien Peytavie, and Bedřich Beneš. Terrain generation using procedural models based on hydrology. *ACM Trans. Graph.*, 32(4):143:1–13, July 2013.

[10] G. Gilet, J-M. Dischler, and D. Ghazanfarpour. Multiple kernels noise for improved procedural texturing. *The Visual Computer*, 28(6-8):679–689, 2012.

[11] H. Gouraud. Continuous shading of curved surfaces. *IEEE Trans. Comput.*, 20(3):623–629, jun 1971.

[12] Thai-Duong Hoang and Kok-Lim Low. Efficient screen-space approach to high-quality multiscale ambient occlusion. *The Visual Computer*, 28(3):289–304, 2012.

[13] Alex D. Kelley, Michael C. Malin, and Gregory M. Nielson. Terrain simulation using a model of stream erosion. In *SIGGRAPH*, volume 22, pages 263–268, jun 1988.

[14] Peter Krivstof, Bedvrich Benevs, Jaroslav Kvrivanek, and Ondvrej vSvtava. Hydraulic erosion using smoothed particle hydrodynamics. *Comput. Graph. Forum*, 28(2):219–228, 2009.

[15] A. Lagae, S. Lefebvre, J.P.Lewis, K.Perlin, M.Zwicker, and et al. State of the art in procedural noise functions. *EuroGraphics*, May 2010.

[16] Benoit B. Mandelbrot and John W. Van Ness. Fractional brownian motions, fractional noises and applications. *SIAM Review*, 10(4):422–437, 1968.

[17] Xing Mei, Philippe Decaudin, and Bao-Gang Hu. Fast hydraulic erosion simulation and visualization on gpu. In *Pacific Conference on Computer Graphics and Applications*, pages 47–56, 2007.

[18] Andrey Mishkinis. Advanced terrain texture splatting. Gamasutra blog, 2013.

[19] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, jul 1985.

[20] Matt Pettineo. A closer look at tone mapping. The Danger Zone - Graphics Blog, 2010.

[21] Jens Schneider, T. Boldte, and Rüdiger Westermann. Real-time editing, synthesis, and rendering of infinite landscapes on gpus. In *Vision, Modeling and Visualization*, 2006.

[22] Josef B. Spjut, Andrew E. Kensler, and Erik Brunvand. Hardware-accelerated gradient noise for graphics. *ACM Great Lakes Symposium on VLSI*, pages 457–462, 2009.

[23] Disney Animation Studios. Physically-based shading at disney, 2012.

[24] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. *SIGGRAPH*, pages 151–158, 1998.

[25] Stanley Tzeng and Li-Yi Wei. Parallel white noise generation on a gpu via cryptographic hash. In *Proceedings of SI3D*, pages 79–87, 2008.

[26] Juraj Vanek, Bedrich Benes, Adam Herout, and Ondrej Stava. Large-scale physics-based terrain editing using adaptive tiles on the gpu. *Computer Graphics and Applications, IEEE*, 31(6):35 –44, 2011.

[27] Hongling Wang, Joseph Kearney, and Kendall Atkinson. Robust and efficient computation of the closest point on a spline curve. In *In Proceedings of the 5th International Conference on Curves and Surfaces*, pages 397–406, 2002.

[28] Li-Yi Wei. Parallel poisson disk sampling. *ACM Trans. Graph.*, 29(6):166:1–166:10, 2008.

[29] Fahad Zafar, Marc Olano, and Aaron Curtis. Gpu random numbers via the tiny encryption algorithm. In *Proceedings of the Conference on High Performance Graphics*, pages 133–141, 2010.