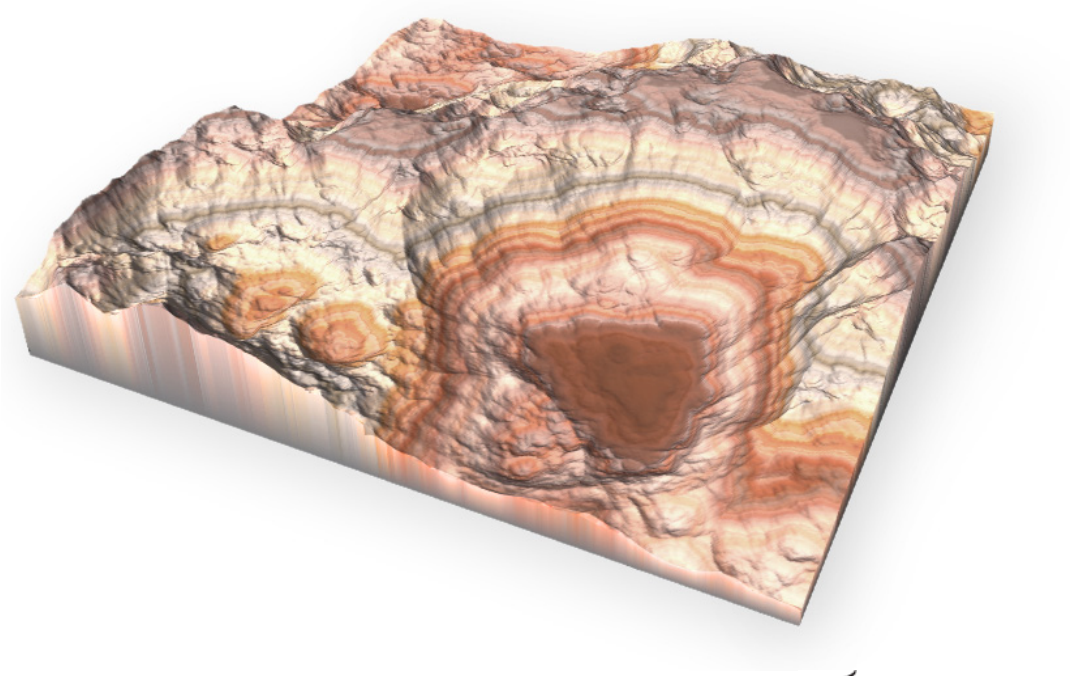
Interactively synthesizing and editing virtual outdoor terrain

Giliam J.P. de Carpentier
giliam@decarpentier.nl

Research assignment report

Submitted in partial fulfillment of the requirements of the degree of
Master of Science in Media and Knowledge Engineering

August 2007



Computer Graphics and CAD/CAM Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

In association with:
W!Games, Amsterdam
The Netherlands

Table of Contents

1 Introduction

Ever since the early days of computer graphics (CG), research has been conducted on modeling and rendering three-dimensional (3D) virtual environments. For a few decades or so, practical applications of this research were limited to offline movie production. Nowadays, even desktop computers have enough processing power to render virtual environments at interactive or even real-time speeds. Consequently, applications like Virtual Reality training simulations and 3D computer games have become feasible.

With the ever increasing processing power of computers, it is possible to create more and more complex worlds at real-time speeds. For example, cutting edge 3D shooting games went from looking like Figure 1.1 to Figure 1.2 in less than fifteen years.

FIGURE 1.1 Wolfenstein 3D (id Software, 1992)

FIGURE 1.2 Gears of War (Microsoft Game Studios, 2006)

From the point of view of a 'gamer', this increased level of detail adds to the realism and immersiveness of these virtual 3D worlds. From the designer's perspective, using this increased processing power can add to the artistic freedom and can give the product a cutting-edge look. However, this graphical complexity comes at a cost. Creating more detail is generally laborious and is, consequently, expensive. [TATA05].

Content creation has been traditionally a manual process to get the most out of the possibilities of a hardware platform. But this situation will not be maintainable for much longer as content creation is increasingly becoming a major bottleneck in game production. Hence, a shift from handcrafted to (semi-automated) generated content is slowly taking place.

This report focuses on only one of the aspects of designing content for virtual environments: outdoor terrain. More specifically, it explores ideas and techniques that are or would be helpful in the process of designing outdoor terrain for 3D computer games. However, general ideas and techniques presented here might be applicable to other areas of the game level design process too. Also, other types of virtual-reality (VR) applications (e.g. 3D simulations and virtual workbenches) that require virtual terrain might benefit from the discussed ideas.

1.1 Problem Statement

The problem addressed in this research report is best described as:

What features would an application need to have to intuitively, effectively and efficiently aid a game level designer in creating and editing virtual terrains, by automating tedious work as much as possible without limiting his/her creativity?

Although this question is not directly answered in this report, recommendations concerning the designer's efficiency are given, together with a survey of techniques aimed at improving different aspects of this design process.

1.2 Motivation

As discussed on page 1, the manual creation of content is not scaling well with the increased technological possibilities and user's expectations. The game industry is currently wrestling with the problem of ever growing artist teams to keep up with the technological possibilities of game platforms. Currently, these artist teams handcraft most of all the geometric models and shading detail required for 3D characters and environments to make it look as realistic as possible. This has become one of the major expenses in all multi-million game productions.

Having 'smarter' techniques available to level designers that partly alleviate this burden by (semi-) automating laborious tasks is slowly becoming indispensable. Yet many of the current tools available to designers leave much to be desired. Therefore, investigating ways to support the workflow of a level designer might be very fruitful.

This report focuses on both commonly available and lesser-known techniques for modeling of terrains for use in computer games. These techniques are investigated from a technological point of view and are assessed based on their effectiveness as an aid to the level designer. Also, ideas will be sought after to create a toolset to maximally support an artist's design workflow, without enforcing a predetermined order of design steps, to maintain maximum flexibility.

Because the report focuses on computer games, the scope is limited to only a (small) part of the field of 3D computer graphics. Namely, to graphics systems that focus on render speed, ruling out many systems that use more advanced off-line rendering techniques.

1.3 Chapter Overview

In Chapter 2 a short analysis of the requirements of game level designers is given and general rules of thumb about usability, efficiency and effectiveness of tools related to level design are described. Chapter 3 describes the different elements of terrain modeling and explains why this report is mostly limited to heightfields.

Chapter 4 explains the basic ideas of both experimental and trialed procedural methods to generate a heightfield landscape. Chapter 5 describes different methods to generate noise, the basic building block of most procedural (terrain) algorithms. Chapter 6 discusses different texture-by-example techniques. These techniques are able to generate images (or equivalently, heightfields) that are like, but not equal to, a given example image/heightfield.

Chapter 7, 8 and 9 focus on different aspects of editing handcrafted or generated terrain. Specifically, Chapter 7 discusses editing heightfields, Chapter 8 covers editing terrain texturing and Chapter 9 discusses foliage placement.

Chapter 10 gives an overview of the currently available editors and generators, focused on heightfields.

Preliminary discussions can be found in chapters 4 through 9, comparing the techniques discussed in each individual chapter from a practical point of view. These discussions complement the overall conclusions that can be found in Chapter 11.

2 Usability

This chapter shortly introduces the reader to the context of terrain design. Then, different aspects of toolbox design for level designers are discussed, aimed at optimally supporting these users in their creative process.

2.1 Terrain Design

A system developed with the techniques mentioned in this report would assist game level designers in creating outdoor terrain for computer games. These terrains could be used as a basis to add level object like roads and buildings to.

In today's outdoor game levels, different areas usually require different amounts of time to design and tweak. This is because storylines and (pre-scripted) actions are usually focused around multiple key points or routes to create pre-designed experiences for the players, thus limiting the need to design all areas on a level down to the smallest scale. Supporting designers at multiple scales is therefore an important factor in maximizing their efficiency. So placing, for example, vast amounts of trees in large areas where less control is needed might be done algorithmically (i.e. procedurally) to make this area more interesting in a few steps. On the other hand, strategically placing a few large trees in (often smaller) key areas for a player to hide behind might still be preferred to be done with complete control over each tree's location.

2.2 Design Loop

Ideally, designing game levels is an iterative process. Often, designers create a fairly detailed level which is then evaluated. See Figure 2.1. Evaluation involves testing a level for the amount of entertainment, which is hard to estimate beforehand. When the level doesn't 'feel finished', the level is tweaked again. Tweaking a level might involve moving only a few objects around or, for example, slightly moving a road. But when that road needs to be adjusted and a large terrain feature like a mountain is in its path, a large area might be affected.

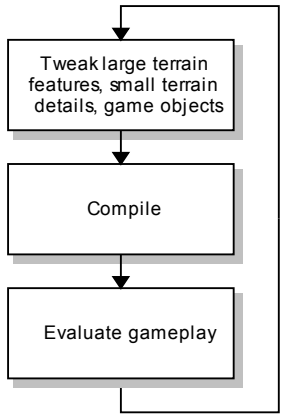


FIGURE 2.1 An ideal level design workflow

However, most applications available to level designers typically are designed around the idea of working from large to small. See Figure 2.2. The arrows indicate the direction of the (enforced) workflow. After the initial idea of a level has been decided, designers have a choice of starting off with a global approximation of this outline. One way of doing this is by searching for a (real-world) example of the type of terrain they need. Another way is to have an application generate a random terrain algorithmically (i.e. procedurally). Techniques used to do this are discussed in Chapter 4. Having a rough first

approximation for a level greatly reduces production time when it is relatively close to the desired end result. Then, large-scale global features can be added, followed by small-scale local editing. The disadvantage of the typical workflow using these tools is that, once an approximation for the whole level is chosen, only lower-level manual editing is possible. This means that only a workflow from left to right in Figure 2.2 is supported, making iterative design of both large and small features, as suggested in Figure 2.1, very difficult.

This report is partly dedicated to editing techniques that would allow higher-level handcrafted or generated features to be mixed and edited at any scale at any time in the design process, to better support the iterative process of a tweak-and-evaluate workflow. Integrating such techniques into the applications available to the designer would, for instance, allow adding a detailed generated mountain in a designated area with minimal effort even after other areas are already tweaked.

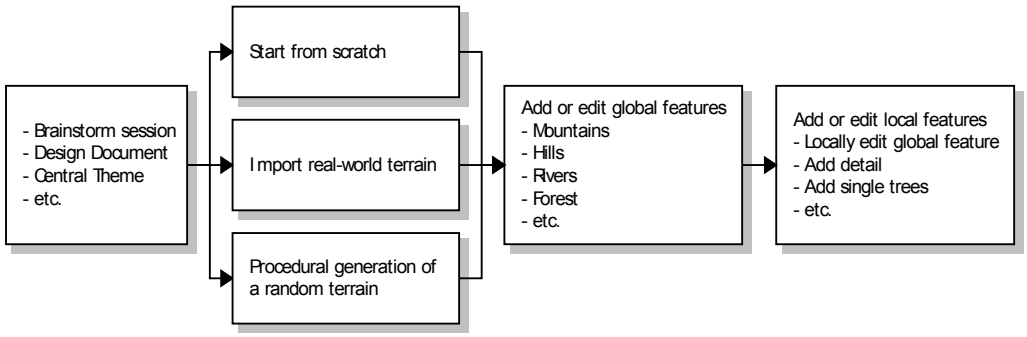


FIGURE 2.2 A typical (left-to-right) workflow supported by current applications

2.3 Parameter Space

Although higher-level tools might be more mathematically involved and harder to code, the user of such a tool should not be required to understand the technical details before he/she can use it proficiently. All that the user should be concerned about is achieving the desired result. This means that tools should behave in an intuitive way that is predictable to a non-technical user. The function of any tools should be unambiguous and easily describable to non-technical users.

One aspect of creating intuitive tools is choosing the right parameter space. Several aspects come into play when designing an intuitive toolset that lends itself to intuitive tweaking:

- Tools that allow parameters to be tweaked to achieve different results should offer an appropriate amount of freedom. Too few parameters, and more advanced users are unable to fully benefit from the technology. Too many, and novice users might get overwhelmed by the possibilities.
- Parameters should have a descriptive name of the effect it has, which is not per se a term used in scientific literature.

- The effect of different parameters should be as independent as possible. Presenting multiple parameters that only have slightly different effects should be prevented where possible.
- The value range of any parameter should be intuitive to the user. Having a value range of 0-1000 with only values between 700 and 800 having a useful effect isn't the best choice.

2.4 Toolbox Consistency

As with any design of a user interface, having a consistent set of tools makes working with it more intuitive. This means the interface of different tools should be as consistent as possible. For instance, having two tools that both need a radius as input should generally offer the same type of interface for this particular parameter. This can be achieved through the consistent use of particular input controls, hotkeys and 3D widgets.

Also, creating a user interface that is consistent with other applications that level designers are familiar with will make a tool easier to work with. For example, implementing (customizable) mouse and key functionality for navigating through a 3D world that is similar to one or more widely used 3D applications is generally appreciated and will increase the overall productivity.

2.5 Flexible Editing

The iterative nature of level design benefits from a powerful multiple-undo function. Having 'Ctrl-Z' functionality greatly helps the designer to experiment with tweaking an effect that requires the execution of a sequence of multiple tools. Undoing multiple actions allows the user to backtrack to any given point in the action history and restart from there.

Having a representation that allows users to tweak a tool that was applied before the most recent operation, without undoing the intermediate operations, further increases this flexibility. One representation that offers this functionality is the separation of (manipulated) data in multiple layers. Many designers already are familiar with this idea from Adobe Photoshop, a well known and powerful 2D image manipulation application. In Photoshop, the user can create multiple layers in a hierarchy and select the layer a tool should be applied to. These layers are internally combined bottom-to-top by the application to render the actual image. Combining a layer with the layers below is done using a user-selectable combine operation per layer, optionally limiting the effect of a layer to a local area using an additional mask image. This allows Photoshop users to separate different elements of a picture and independently apply operations to them (e.g. draw with the selected brush, translate, scale and blur) or apply operations to the relation between a layer and the layers below (e.g. blend mode and opacity setting). A possible drawback of this layered representation is the memory footprint that grows linearly with the number of layers.

Even more flexible and powerful is the representation of operations as a two-dimensional flow graph of operation nodes. This allows the user to apply a tool by connecting the input(s) of a new operation node to any of the already present nodes in a visualized flow network. Tweaking any of the previous steps can be accomplished by changing parameters in any of the nodes and recalculating outputs until all nodes are up to date again. A few powerful high-end content creation and processing applications use this representation. Examples of these are Apple's Shake compositing tool and Side Effects Software's Houdini procedural 3D animation/effects tool. A typical designer might not be used to 'thinking' in flow graphs and operation building blocks, causing a steep learning curve. However, expert users might be very pleased by the, otherwise hard to accomplish, flexibility. A drawback of this system is the amount of recalculation required when applying a change to the flow graph. This can be partly alleviated by reusing cached outputs if none of their inputs was affected by a change. Of course, caching increases the memory footprint considerably for large flow charts.

In short, which of the above representations is the most appropriate depends on the need for flexibility, the available system resources and the expertise of the user. Favoring one above the other requires a clear view of the exact toolset and its use and is therefore beyond the scope of this report.

2.6 Tool Speed

Even complex 'tweakable' tools and parameters become usable when their effect is directly visible. Because optimizing a virtual world for its amount of fun or artistic beauty isn't an exact science, it is often a process of trial-and-error. Shortening the feedback loop gives designers the opportunity to experiment with parameters more freely. Therefore, having tools that can be used at interactive speeds is a valuable asset.

If calculating the effect of a tool is too compute intensive to allow a preview of the effect in the edited world at interactive speeds, previewing the result at a smaller resolution might be a good compromise. This smaller preview might either be a smaller window or a less dense geometry representation the operation is performed upon. Obviously, this is only useful if the preview of the result at reduced resolution is a fair approximation of the final result.

The hardware available to designers typically consists of a stand-alone powerful desktop computer with plenty of RAM and a high-end graphics card with a powerful GPU. Since the early days of hardware-accelerated video cards, the processing power of the GPU has increased dramatically and remains to grow faster than CPU processing power. Factors that contribute to this fact are the increased clock speeds, amount and speed of onboard dedicated memory and the shift from a single special-purpose graphics processing unit to multiple (almost) general-purpose programmable Single Instruction, Multiple Data (SIMD) vector processing units.

Many algorithms that can be implemented for partial or full parallel execution can therefore be sped up somewhere between a half and two magnitudes when work is transferred from the CPU to the GPU. Because terrain manipulation is very data-intensive and for the most part SIMD-like, executing these manipulations on the GPU is very likely to increase the performance. Furthermore, multi-core CPUs are becoming more and more mainstream, increasing the potential of parallelism even further. Having fast tools increases the user's efficiency, which is why all algorithms mentioned in this report are evaluated, among other criteria, for their potential to execute in parallel.

2.7 Tool Quality

Even though procedurally generated and placed geometry, texturing and foliage might look nice at a first glance, level artists/designers easily spot the limitations of most current implementations. Generally, natural terrain has different types of features at different locations. Also, most levels are designed with a clear idea of what type of environment it should be set in. However, most procedural techniques are best suited for creating one or a few terrain types (e.g. ridged mountains, rolling hills, sand dunes, rivers or islands). Therefore, it isn't recommended to have one technique create the terrain for a whole game. Having a plethora of different techniques to choose from enables the designer to pick the right tool for each job.

The quality of any terrain tool is difficult to measure. If all processes involved in the creation of a certain type of landscape are fairly well understood, it is possible to create a model of these physical effects and run a simulation. Although this will result in physically most accurate results, running a full simulation might be impossible due to a limited understanding of a process or impractical due to the vast computational power required for an accurate simulation. Luckily, as an engineer and artist, not as a scientist, a level designer is generally satisfied if a tool is available that has the desired effect, whether such a tool is physically correct or not. For him, and for the typical end-user, subjective beauty of the result is much more important than objective measurements, mathematical elegance or statistical proof. For this exact reason, this report focuses mainly on the effects of different (efficient) methods, not on mathematical backgrounds.

3 Terrain Specification

Different aspects of terrain come into play when designing appealing virtual landscapes. Three of the main aspects are geometry, surface properties (texturing) and placement of natural objects. These three topics are introduced in this chapter and further discussed in the remaining part of this report.

3.1 Terrain Geometry

Before discussing terrain geometry generation and editing, a small overview is given of the different options for specifying terrain geometry, each with its advantages and disadvantages for use in interactive applications. Choosing the type of topology to use for terrain geometry has a large impact on the possible types of creation and editing algorithms. But it also greatly influences what rendering techniques are suitable, how level-of-detail can be implemented in the real-time engine (i.e the core of each real-time graphics application) and whether it is possible to have overhangs, arches and caves. Having such impact, the choice of which types of terrain geometry can be used is often dictated by the graphics engine. Five types of geometry are distinguished and described below. Also, see table 3.1.

	Irregular topology	Regular topology
Solid	Tetrahedrons	Voxels
3D surface	Irregular mesh	Regular mesh
2½D surface		Heightfields

TABLE 3.1 Types of terrain geometry specification

Tetrahedrons

Starting with the most flexible type of terrain specification, tetrahedrons allow variable densities of vertices. Because of this flexibility, tetrahedrons are often used in physics simulations that use finite element techniques. Also, solid modeling can be implemented using tetrahedrons. However, this flexibility comes at the cost of larger storage requirements and more complex algorithms to handle the irregular 3D shapes and densities. Because of this, their use in interactive terrain specification, generation and rendering is limited. For this reason, algorithms working on tetrahedrons are not discussed further in this report.

Voxels

Voxels are values on a regular 3D grid. Like tetrahedrons, voxels are volume based. So, creating holes, overhangs and caves is relatively easy. However, the amount of local detail is limited by global resolution of the regular grid. Also, the same resolution is present (and takes up memory) where less resolution is needed. Because of this, voxels are generally memory inefficient. Furthermore, rendering voxels is generally less efficient than rendering triangle surfaces on today's polygon-based hardware accelerated video cards. Therefore, only few games actually use voxels. So, like tetrahedrons, voxels are not relevant enough to be treated in this report.

Irregular Meshes

Irregular meshes are surface based, have flexible topology and might have varying densities of vertices. Although ideal for surface specification, the implementations of high-level modeling tools are more complex than implementations of equivalent tools for regular mesh representations. Rendering irregular meshes at full resolution is well supported by hardware. However, terrain rendering often requires different LOD (level-of-detail) levels at different parts of the mesh to render terrain at full resolution near the camera while rendering a coarser mesh further from the camera. Accomplishing this for irregular meshes is generally much more complex (and much more compute intensive) than for regular meshes. This is also true for collision detection and response. Both of these issues are serious drawbacks in computer games because almost every 3D game needs fast level-of-detail schemes and collision detection to be able to run at real-time speeds. Because of this, irregular meshes are often only used for objects like characters and trees, where it is generally sufficient to control the level-of-detail for the object as a whole and only require approximate collision detection.

Regular Meshes

Having a regular (grid-like) topology greatly reduces the complexity that is coupled with irregular meshes. Regular meshes are powerful enough to model overhangs and have varying vertex densities, but do not allow specification of arches, connected tunnels or other features that require holes in the surface geometry. Also, most irregular mesh algorithms (procedural generation, editing, level-of-detail and collision detection) can be simplified and optimized for regular meshes. For applications where heightfields are not sufficient because overhangs are needed, regular meshes might be a good choice.

Heightfields

Although the least powerful, most computer games use heightfields to represent terrain. A heightfield (also called heightmap, (digital) elevation map or DEM) represents a discretized height function of 2D coordinates on the horizontal plane, using height samples at regular discrete spacing. A mesh of (vertically displaced) triangulated square quadrilaterals is normally used as its 3D representation, but other topologies are sometimes used also. Rendering and other relevant techniques have been optimized for heightfields. Also, heightfields can be stored very compactly, because only data for the vertical axis needs to be stored.

Because heightfields are discrete functions of 2D space, they can be stored, visualized and even edited as grayscale images. As a 2D grayscale image, the greyvalue represents the local height. Editing techniques for heightfields and digital images are therefore interchangeable. By convention, the maximum altitude is represented by pure white and the minimum altitude by black. Heightfields of considerable detail are publicly available for planet Earth. These can be downloaded and used as a reference or a starting point for anyone interested. For example, see *http://library.usgs.gov*.

Another advantage of heightfields is the ease of texture mapping. A simple vertical orthographic projection of a detailed texture onto the heightfield is generally sufficient. See the next paragraph for an explanation of the term texture. However, when a heightfield contains very steep areas, a simple vertical projection leads to an uneven distribution of texture resolution. Then, a more advanced texturing technique might be required to prevent the otherwise uneven distribution of texture resolution from becoming noticable. Readily available satellite photographs can be used as texture images, which can be found online for the whole planet. For example, see *http://www.truearth.com/*.

When supported by the engine, heightfields can be replaced locally by more powerful representations (e.g. regular meshes) where more resolution or geometry like overhangs or arches is required. For example, see [GAMI01] for a 3D displacement mapping technique to create overhangs with heightfields.

Because of the overall advantages, heightfields are still the most common way to specify terrain for real-time 3D applications. Consequently, this report has limited its geometry-related topics to the use of heightfields. Literature can be found on regular heightfields that are either based on quadrilaterals, triangles or hexagons. For example, see [DIXO94] for procedural terrain generation techniques for different topologies. However, most literate assumes a quadrilateral structure and, moreover, almost all applications use regular quadrilaterals, also known as quads. For this reason, heightfields mentioned in this report are assumed to be based on regular quadrilaterals, unless explicitly stated otherwise.

3.2 Texturing

To render the terrain geometry, all surfaces have surface properties assigned to them. These properties consist of local mapping parameters (i.e. the texture mapping) and a shader. The shader uses the local parameters, the camera direction, the local geometry and possibly other input images (called textures) to calculate the color of a screen pixel. Surface shading might be as simple as outputting an evenly lit projected texture on a surface or as complex as procedurally generating animated natural phenomena (e.g. rendered reflective caustics of a water surface). The rendering process of shading surfaces can be hardware-accelerated by today's high-end desktop computers.

Besides creating the terrain geometry, designers also need to assign these surface properties to different areas (e.g. an image of rocks in one place and grass in another). This report introduces the reader to common texturing techniques that can be found in current applications and discusses a texturing technique called texture splatting in detail in Chapter 8.

3.3 Foliage Placing

Although object placement is supported by most level design tools, placing foliage (modeled grass, bushes, trees, etc.) that looks natural can be tedious if a level designer is creating areas with a lot of vegetation. The distribution density of foliage in the real world depends on the many factors including soil, temperature, humidity, slope, height and even on other species of flora in the area. Having powerful tools that can place vegetation while considering (some of) these factors greatly simplifies the process of natural and balanced placement of different types of vegetation. This is discussed in Chapter 9.

4 Procedural Heightfields

In the following four chapters, algorithms are discussed that are related to generating and editing heightfields, the most common representation of terrain geometry. This chapter discusses procedurally generating heightfields.

4.1 Procedural Synthesis

Procedural synthesis or generation is the term used for techniques that create content algorithmically. These algorithms do not need to be physically correct, elegant or deterministic. They have two advantages in the field of computer graphics. One is the smaller storage requirement. The code needed for procedural algorithms only takes up a fraction of the storage space that is required to store the large (or even infinite) amount of detail it can output. The other advantage is design. Whereas handcrafted data is generally only used once, a carefully designed parameterized algorithm could be reapplied many times to generate varied output of comparable quality. On the other hand, design through the use of procedural algorithms can be complicated if a specific result is desired that cannot easily be expressed in the exposed parameters. This disadvantage can partly be alleviated by the techniques discussed in Chapter 7.

Generating content through procedural algorithms has proven to be fruitful in fields like the generation of plants [PRUS90], cities [PARI01], clouds [VOSS89], complex (fractal) implicit surfaces [PERL89], texture generation [PERL85] and heightfields [MAND82]. Because procedural techniques are very promising in the field of design, a considerable share of this report is dedicated to procedural techniques that are directly or indirectly related to terrain generation and foliage placement. This chapter discusses procedural algorithms related to the generation of natural heightfields.

4.2 Brownian Motion Fractals

The first person who noted mountain-like properties of a mathematical process was Mandelbrot. In [MAND82] he observed the similarity between a trace of the one dimensional fractional Brownian motion over time and the contours of mountain peaks. Extending this idea to two dimensions created a 'Brownian surface' resembling a mountainous scene. This Brownian process was later generalized to fractional Brownian motion (fBm) surfaces with a $1 / f^{\beta}$ power spectrum. β is called the spectral exponent and is directly related to the fractal dimensionality. Although mountains do exhibit some self-similarity, the formation or shape of mountains is not (known to be) quantitatively connected to fractals [LEWI90]. But as a descriptive model, this does not have to be an objection to use it to approximate natural terrain.

FBm surfaces do posses some features that visually distinguish them from real mountainous terrain. The increments of an fBm process have the property of being isotropic and stationary,

creating terrain that is statically invariant under translation and rotation. This will result in terrain that looks too homogeneous when compared to mountainous areas. Also, fBm surfaces have no local spatial relationship between amplitudes of different frequencies. Whereas natural scenes clearly have, as mountain tops are on average locally rough and valleys are locally smooth. Even so, fBm models are still the basis for many procedural terrain generators [MUSG93, p. 33].

By definition, fBm is the integral over time of increments of a pure random process, also called a random walk. This stochastic process can be synthesized by summing over a basis function at multiple discrete frequencies with different amplitudes to create its characteristic $1 / f^{\beta}$ power spectrum. Examples of possible basis functions are band-limited noise functions and sine waves. Varying the basis function and power spectrum has proved to be a powerful method to generate landscapes. Because natural terrain is not per definition best approximated by an fBm surface, exploring different variations that do not yield a true fBm surface, but do have some fBm-like qualities can yield better (more natural) results. Also, approximations can be calculated in several different ways. Most terrain generating applications are based on one of the approaches discussed below.

4.3 Fractal Synthesis

One possible implementation of creating an fBm surface involves the displacement of a plane by summing over the effect of many independent random Gaussian displacements (faults, or step functions) with a Poisson distribution. This was originally employed by B.B. Mandelbrot [MAND82] and R.F. Voss [VOSS85] to create the first procedural landscapes.

Poisson faulting

'Fault formation' and 'particle deposition' are two variants of Poisson faulting. Fault formation is introduced in [KRTE01] and is illustrated in Figure 4.1. Faults are created by repeatedly displacing the heightfield values at one side (i.e. halfspace) of a randomly chosen line through the heightfield by some amount. This process is repeated many times while the amount of displacement per iteration is slowly decreased. Because the result might still be too rough and aliased afterwards, a low-pass filter is normally applied as a final step.

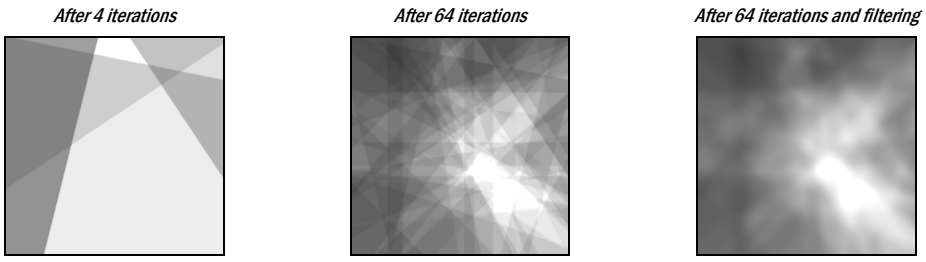


FIGURE 4.1 Creating a fault formation heightfield. Higher areas are lighter

Fault formation can create elongated mountain ridges and faults. However, most fine detail is lost because of the low-pass filtering. Also, the steepness of faults is directly related to the parameters used for the low-pass filter. Furthermore, many iterations are necessary to create a reasonable complex landscape. Creation is mostly fill rate limited because, on average, half the height values are updated for each iteration. It follows that this algorithm has an $O(n^3)$ work complexity, where n is the width or height of the heightfield (expressed in number of vertices) and the number of iterations is related to n. Because of these drawbacks, this technique is seldom used in commercial heightfield applications. One of its merits is the applicability of this idea to primitive shapes other than vertically displaced planes (i.e. heightfield), which might be difficult to do with other techniques. For example, [ELIA01] discusses fault formation on spheres. For a more elaborate discussion of fault formation, see [SHAN00].

Another type of Poisson faulting is called particle deposition, which involves a simple simulation of dropping particles on a flat plane. When a dropped particle touches the heightfield, it will 'roll' further downwards until a local minimum is reached and there it will increase the value of the heightfield with a small value Δ. See Figure 4.2. When enough particles are dropped, the produced pattern will (somewhat) resemble viscous fluid (e.g. lava). Because two adjacent heightfield elements can only differ by Δ, the maximum steepness depends on Δ and the heightfield grid spacing. This 'roll' simulation is a very crude approximation of thermal weathering (See Section 7.3). The shape of the terrain can be controlled by changing the drop pattern. This technique is primarily suited for creating volcanic terrains. Because of its local control and simple implementation, this technique might be useful for interactive editing.

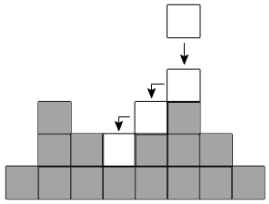


FIGURE 4.2 Flow simulation in particle deposition

Midpoint Displacement

Introduced by Fournier et al. [FOUR82], midpoint displacement has long been the preferred technique to efficiently generate terrains. Heightfields are created by recursively subdividing (i.e. tessellating) a heightfield mesh and randomly perturbing all new vertices. When the perturbation has a Gaussian distribution and a standard deviation of $2^{-\ell H}$, the result will be an approximation of an fBm when ℓ is the subdivision level and H is the self-similarity parameter in the range $[0, 1]$. See the paragraph on noise synthesis on page 17 for more information on the relation between fractal terrain roughness and H. All midpoint displacement schemes have complexity $O(n^2)$, n being the width of the (typically square) heightfield. Because the amount of calculation per vertex is also very limited, midpoint displacement schemes are very efficient.

Different subdivision schemes have been devised for different mesh topologies. [FOUR82] used a triangle subdivision that involves interpolating between the two vertices. Mandelbrot introduced a subdivision scheme specifically for hexagon meshes [MAND88]. However, these topologies are seldom used in terrain specification and will not be discussed in this report.

The widely used diamond-square scheme for quadrilaterals was also presented in [FOUR82]. This two-phase algorithm subdivides a regular square grid at any level by first calculating and perturbing the (new) exact midpoints of each set of four nearest neighbors that together form a square. Then, another set of vertices is interpolated between each set of four nearest neighbors that together form a diamond (two of which were calculated at previous levels and two were calculated in the phase 1 of this subdivision level) and is perturbed. This will create a new regular grid of quadrilaterals. See Figure 4.3.

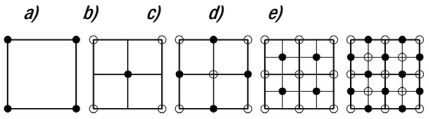


FIGURE 4.3 Square-diamond midpoint displacement. b) and d) are intermediate results after applying the first phase. c) and e) applied phase 2. From [OLSE04]

The diamond-square scheme creates visible anisotropic artifacts along the (eight) directions of interpolation. The square-square scheme presented in [MILL86] subdivides a regular mesh by using its 'input' mesh as a regular mesh of control points for a biquadratic uniform B-splines interpolant. This results in less visible anisotropic artifacts. A disadvantage of this interpolation scheme is the smaller size of the mesh after each subdivision step. Also, the fact that the resulting surface generally doesn't go through the set of control points, but only approximates them, might be a drawback for some applications.

Midpoint subdivision has been used in many simple terrain generation applications. It is generally easy to understand and implement. Furthermore, it is very efficient if a whole patch needs to be subdivided and stored in memory. For example, in square-diamond subdivision, each terrain vertex needs only to calculate one interpolation and perturbation, whereas most other synthesis techniques (see next paragraph) need many interpolations. But because of its nested structure, this method is less suitable for ad-hoc local evaluation and only works on heightfields of $2^k \times 2^k$ vertices.

The principle of interpolating values of neighboring vertices and adding a perturbation was extended to Generalized Stochastic Subdivision in [LEWI87]. There, a larger neighborhood, together with an autocorrelation function for each subdivision level, is used to allow creation of a mix of stationary (noisy) and non-stationary (periodic) patterns. Although flexible, it needs many more parameters than the methods above. For this reason, most terrain generating applications do not support generalized stochastic subdivision. However, it might have some limited use in creating terrain types that are hard to create with other techniques, e.g. (periodic) sand dunes.

Fourier Synthesis

Fourier synthesis can be applied for terrain generation as follows: First, the 2D Fourier transform is calculated of a random Gaussian white noise heightfield. Secondly, the noise in the calculated frequency domain is multiplied with a pre-designer filter to create the desired frequency spectrum. Lastly, the multiplied result is transformed back to the spatial domain using the inverse Fourier transformation. When the right frequency spectrum is chosen, an fBm process is approximated [VOSS89]. An obvious advantage of this approach is the exact control over the frequency content.

Disadvantages are the periodicity of the final surface and the $O(n^2 \log n)$ complexity of 2D FFTs. Also, any heterogeneous extension for local spatial control of detail during construction is less straightforward than for noise synthesis (see below).

Noise Synthesis

Noise synthesis is the iterative summing over band-limited noise functions. The noise functions approximate a band-limited sum of frequencies with random amplitude and phase. By calculating a weighted sum of 2D noise functions of different band-limited frequency ranges, any power spectrum can be composed, including a $1 / f^{\beta}$ spectrum, approximating an fBm surface.

When $G(t)$ is the Fourier transform of a function $g(t)$, $\frac{1}{\|c\|}G(\frac{t}{c})$ is the Fourier transform of $g(ct)$. This means that when the input of a band-limited noise function N is scaled by (a positive) c, the frequency spectrum of N is scaled by $1 / c$. So, having just one band-limited noise function and scaling its input and its output will create another band-limited noise function with a scaled mean frequency. Noise synthesis can therefore be written as:

$$H_{L_{\min}}^{L_{\max}}(x,y) = \sum_{l=L_{\min}}^{L_{\max}} w^l N(\lambda^l x, \lambda^l y)$$

Here, l represents a detail level and λ^{Lmin} and λ^{Lmax} represent the largest resp. smallest scale level any band-limited detail should be visible at. This means that $L_{max} - L_{min} + 1$ is the number of summed noise functions. Increasing the number of calculated levels increases the total range of frequencies covered at the cost of extra computing power. λ, called the lacunarity, is the scale between the mean frequency of each of the successive noise levels. Increasing the lacunarity will increase the gaps between the separate noise evaluations, creating an uneven distribution of represented frequencies, but fewer levels will be needed to cover the same total frequency range. Somewhat like the subdivision scale of midpoint displacement, most noise synthesis implementations use $\lambda = 2$, or a number very close to it, as the optimal tradeoff between accuracy and speed. As a result, the mean frequency of the noise function is roughly doubled at each level. Because of this doubling of frequencies, levels are also called octaves, borrowed from sound theory. The constant w controls the roughness of the synthesized result and can be written as a function of λ and the spectral exponent β, introduced earlier [MUSG93, p. 37]. The relation between these three parameters is as follows: $w = \lambda^{-\beta/2}$. Often, the terrain roughness is specified by the self-similarity factor parameter H, with $\beta = 1 + 2H$. The fractal dimension D_f is $3 - H$. To qualify as a fBm, H must be in the interval $[0,1]$. This means the fractal dimension lies between a 2D surface and a 3D volume (assuming that an infinite amount of levels would be calculated). True (non-fractional) Brownian motion has a $1 / f^2$ power spectrum and has therefore a fractal dimension D_f of 2½. See Figure 4.4.

The actual noise function can be constructed in different ways, each having a different characteristic band-pass quality and construction speed. An overview of these functions is given in Chapter 5.

The above formula can be generalized to create more types of terrains by allowing a function to transform each noise octave before it is added:

$$H_{L_{\min}}^{L_{\max}}(x,y) = \sum_{l=L_{\min}}^{L_{\max}} w^l T(N(\lambda^l x, \lambda^l y))$$

The turbulence function *T(n)* [PERL89] is one of the first algorithms to explore the possibilities of this generalization by defining *T(n)* as *abs(n)*. Taking the absolute value of *[-1, 1]* noise folds it at each zero crossing, creating discontinuities and doubling the number of (positive) peaks. This creates more billowy, turbulent, cloud-like fractal landscapes. See Figure 4.5. Another variant is *T(n) = 1 - abs(n)*. This transform has the opposite effect, creating 'ridges' at the discontinuities around *n = 0*. The results created with non-linear functions are still fractal, but do qualify as fBm surfaces.

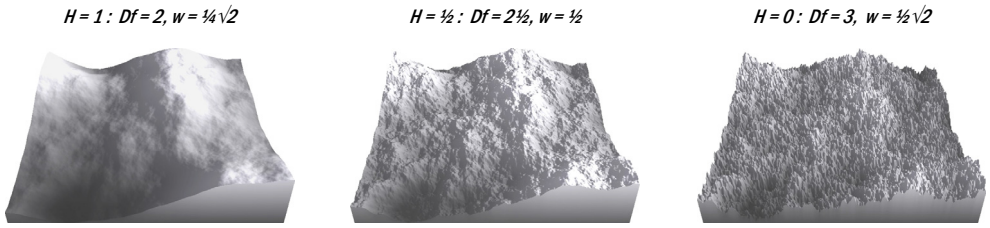


FIGURE 4.4 Heightfield of different fractal dimensions. Perlin noise

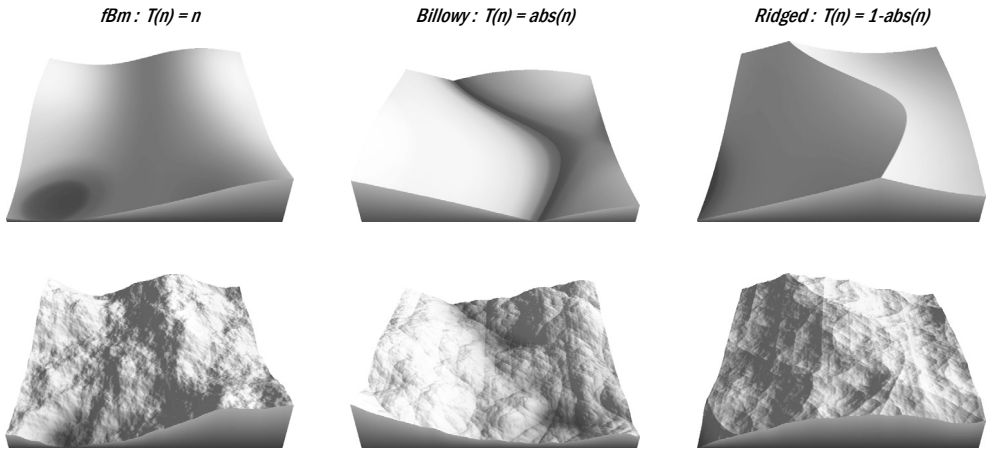


FIGURE 4.5 Heightfields with one octave (top row) and eight octaves (bottom row) of transformed noise. Perlin noise, *H* = ½

Of course, many other functions might prove useful for different types of terrain. One flexible way to give the user the freedom to experiment with this would be to present a simple input/output *T(n)* mapping function as an editable (e.g. drawable) curve.

Local properties of real terrain are not stationary (i.e. statistically translation invariant). Foothills are smoother, while mountain tips are more jagged. The midpoint displacement and noise synthesis approaches can be modified to simulate this observation by controlling the local statistics. To do this, *T* can be defined to depend on the sum of lower frequency octaves, i.e.:

$$T(n) = G(H_{L_{min}}^{L_{max}-1}(x,y)) \cdot n$$

Since higher octaves will have less amplitude (the factor w^l), the sum of all lower octaves $H_{L_{min}}^{L_{max}-1}$ can generally be interpreted as an approximation of $H_{L_{min}}^{L_{max}}$. When *G(n)* is a function that is positively correlated to *n*, *T(n)* will have the effect of locally increasing the noise amplitude at higher altitudes. This has the desired effect of creating rougher terrain (with a higher fractal dimension) at high altitudes and smoother terrain at low altitudes. This type of fractal is called a heterogeneous multifractal. Another way of creating heterogeneous multifractals is by multiplying multiple noise octaves instead of summing them.

$$H_{L_{min}}^{L_{max}}(x,y) = \prod_{l=L_{min}}^{L_{max}} w^l (O + N(\lambda^l x, \lambda^l y))$$

Here *O* is an extra offset parameter that is somewhat reciprocally related to the roughness of the result. The actual range of output values for this type of multifractal is highly unpredictable. Therefore, the output range needs to be measured after creation, so it can be rescaled to a predictable range (e.g. *[0, 1]*). See Figure 4.6 for an example. In [EBER03, p. 498-506], different variants of these multifractal techniques are discussed in detail.

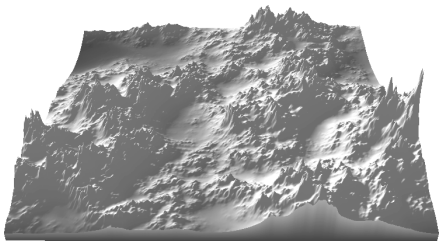


FIGURE 4.6 Height-dependent high frequencies

The octave transformation function *T(n)* can also be made to depend on other inputs. For example, the function *T(n) = M(s x, s y) n*, with *s* being a scaling factor and *M(u, v)* being the local greyvalue of a 2D image at coordinate *(u, v)*. Here, *T(n)* is used to control the local roughness by looking up an amplitude multiplier from another image. The 2D image itself can also be a procedurally generated fractal. This is just one example of cascading, a powerful concept where a procedural algorithm uses other procedural algorithms or complex handcrafted work as input parameters. This idea fits nicely with layers and flow charts, discussed in Section 2.5.

Range Mapping

Another way to create more varying landscapes is to transform the output of *H* as a post-processing step :

$$H'(x,y) = P(H(x,y))$$

To let *P(z)* be as independent as possible of the exact parameters used to construct *H*, *H* is generally rescaled to the range of *[0, 1]* as an intermediate step.

Two functions that are often used for this type of mapping are the *bias* and *gain* [PERL89] functions:

$$bias_b(z) = z^{\log b / \log ½}$$

$$gain_g(z) = \begin{array}{l|l} ½\ bias_{1-g}(2z) & z < ½ \\ 1 - ½\ bias_{1-g}(2-2z) & otherwise \end{array}$$

These functions have the following useful properties:

$$bias_{½}(z) = z$$
$$bias_b(0) = 0, \quad bias_b(1) = 1$$
$$bias_b(½) = b$$

$$gain_{½}(z) = z$$
$$gain_g(0) = 0, \quad gain_g(1) = 1, \quad gain_g(½) = ½$$
$$gain_g(¼) = ½\ (1-g), \quad gain_g(¾) = ½\ (1+g)$$

These simple properties make them transparent and intuitive to a user. See Figure 4.7 for examples of these functions, together with their effect on a heightfield.

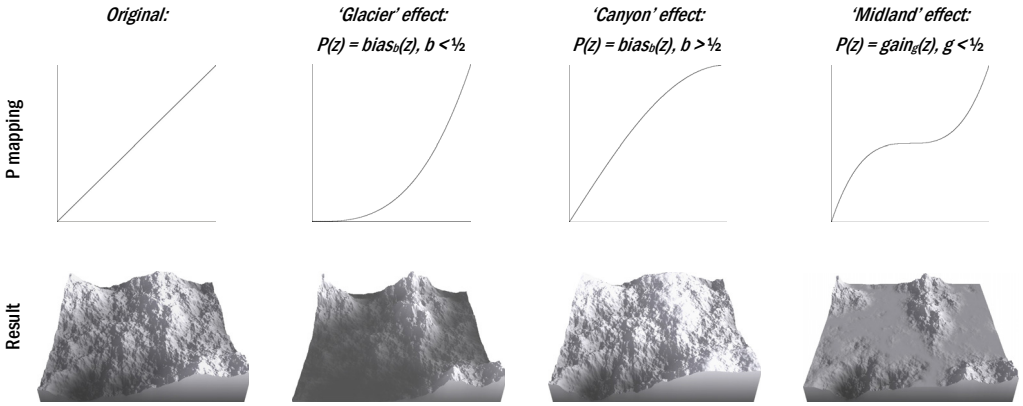


FIGURE 4.7 Heightfields after post-processing. Perlin noise, $H = ½$

Domain Mapping

Range mapping transforms a function's output. Analogously, domain mapping transforms a function's input before the function is evaluated. Besides obvious uses like scaling and rotation, input perturbation is a valuable and flexible tool when defined as:

$$H'(x,y) = H(P(x,y))$$
$$P(x,y) = (x + N_1(x,y), y + N_2(x,y))$$

where N_1 and N_2 can be any (scaled) noise function. As a result, P perturbs the input coordinates of H. See [EBER03, p. 450] for details.

For example, a noise synthesized heightfield that used a Voronoi noise base function (see Section 5.4) will contain many straight ridges. By applying a domain mapping with N_1 and N_2 being (differently translated, rotated and scaled) Perlin noise functions, interesting and natural looking curves and shapes appear. See Figure 4.8. This is another example of cascading different functions to increase the visual complexity of the result.

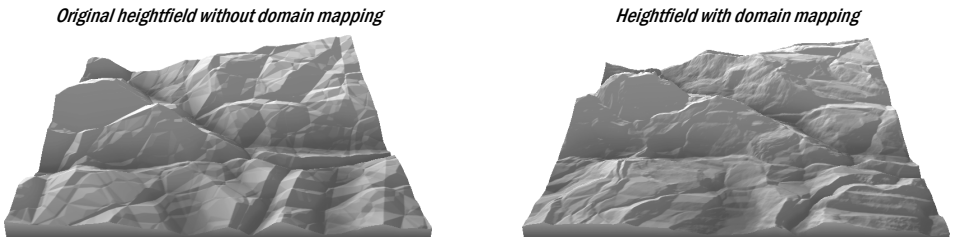


FIGURE 4.8 Voronoi heightfield without (left) and with (right) noise distorted input

4.4 Erosion

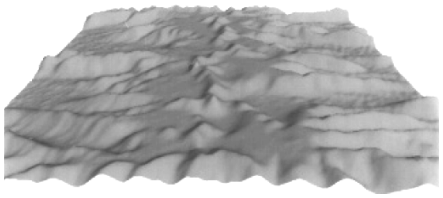


FIGURE 4.9 Result of 100 iterations of fluvial water erosion. From [CHIB98]

In [CHIB98] Chiba et al. describe an algorithm that takes an alternative approach to fractal synthesis by physically simulating fluvial (water) erosion. This algorithm iterates a number of times over two subsequent phases. In the first phase, several erosion-related data fields are calculated from the current (and initially flat) heightfield. Then, the data fields are used to simulate erosive processes on the heightfield. The data fields calculated in the first step are a water quantity field *W*, a water velocity vector field *V* and a collision energy field *C*, which are all discretely sampled using regular grids similar to the terrain heightfield. These fields are estimated using a time-step simulation of many water particles. The water particles are dropped at each grid point and move downhill. At every simulation step, all cells of the data fields that the particles pass are updated. When a particle moves into a grid cell which is steeper, the length of the local vector in *V* is increased. When a particle enters a grid cell which is less steep, the local length of *V* is decreased and the lost kinetic energy is added locally to the collision field *C*. *W* represents the total amount of water that passed through each cell. When all water particles moved outside the terrain heightfield or do not have any kinetic energy left, the first phase is completed. The second phase uses *W*, *V* and *C* to calculate how much sediment will be dissolved, transported and deposited, based on simple empiric rules. See Figure 4.9 for an example of a terrain created by this method.

Physics-based terrain erosion algorithms, like the algorithm described above, often need many compute-intensive iterations before the result becomes valuable, making them generally (much) slower than the fractal synthesis techniques described above. F.K. Musgrave et al. [MUSG89] describe a two-pass approach as a combination of the two different approaches, which is now supported by most of the advanced terrain generation applications. First, one of the above fractal synthesis techniques is used to create a first approximation. Then, an additional erosive pass is run on this approximation. Depending on the type, strength and number of iterations of the erosion process, the erosive pass carves out small gullies and river beds and creates flat sediment planes and talus slopes. When parameters for this second pass are tweaked by a designer, it's unnecessary to recalculate the result from the first pass every time. Therefore, caching the intermediate heightfield might be appropriate. Having such a clear distinction between these phases even allows

a complete separation of the procedural fractal generation step and the erosion step. Hence, it makes sense to offer erosion as an independent tool to the user. Furthermore, having erosion as an interactive tool increases the flexibility to designers who might need to apply erosion only locally. This is why this report further discusses erosion as an editing tool in Section 7.3

4.5 River Networks

One of the drawbacks of all fractal synthesis techniques discussed so far is the lack of explicit river networks in a terrain. Furthermore, adding realistic rivers to a terrain after the terrain already has been generated with one of these techniques has proven difficult. Two alternatives will be discussed here that create river networks before the final heightfield is calculated.

In [KELL88] A.D. Kelley et al. describe a procedure to recursively create drainage networks first that are then used to create the topography of the terrain. The algorithm iteratively inserts tributaries into the drainage network using empirical rules, creating a fractal network of streams. Then, a (smooth, non-fractal) surface is fitted by a surface under tension technique. See Figure 4.10. Although this surface might afterwards be distorted to create rougher terrain, the distortion cannot be too strong, as streams might otherwise end up flowing uphill.

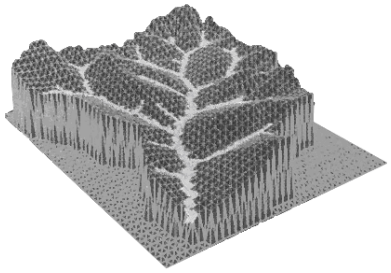


FIGURE 4.10 Drainage network and fitted (non-fractal) surface. From [KELL88]

In [BELH05] F. Belhadj and P. Audibert discuss the idea of modeling outlines of mountain ridges using pairs of 2D Gaussian-shaped particles moving in opposite directions. These particles are randomly translated using fractional Brownian motion. After settling, the trails made by these particle pairs are interpreted as rough outlines of mountain ridges. Then, virtual water particles are placed at these ridge lines and simulated to roll downhill. The trail of these water particles is then interpreted as the shape of a river network. At this point, the heightfield is partly filled with fine ridge lines and river trails. By extending the idea of diamond-square midpoint displacement, all other values of the heightfield are recursively interpolated. See Figure 4.11.

Heightfield after ridge and water particle simulation

Heightfield after midpoint displacement

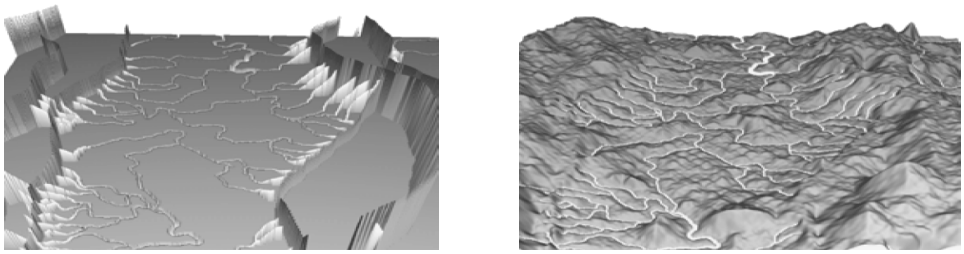


FIGURE 4.11 Fractal landscape with river network. From [BELH05]

4.6 Preliminary Discussion

When attempting to compare the techniques discussed in this chapter, it is evident that the different types of advantages and weaknesses of the algorithms do not allow for simple ranking. Because the types of projects that require outdoor terrain can vary wildly, much of a tool's usefulness depends on whether it is appropriate for a project's style and requirements. As described in Section 2.7, offering a designer a wide variety of tools to choose from allows him to pick the best tool for the job, as long as the toolset is consistent and intuitive. Therefore, expensive (i.e. compute intensive) specialized methods may still have a limited area where these methods are preferred over a method that is generally both better and faster. Supporting a limited number of more expensive methods might therefore be beneficial to the designer. This makes it difficult to compare these algorithms quantitatively.

However, offering all possible algorithms and options to a designer is generally a bad methodology. This would clutter the interface to the user while offering too many overlapping functionality. As a preliminary conclusion, noise synthesis seems to be the best general fractal technique because of its speed, results and flexibility. This is not to say that other techniques are irrelevant as some might be offered in addition. Below, noise synthesis is compared to the other techniques using a number of different criteria.

The complexity of noise synthesis when using a lattice noise basis function is $O(n^2)$, which (in its limit) is constant with respect to the number of vertices in a square heightfield. The constant depends on the number of detail levels calculated. Perlin noise, the basis function that is most used, can be evaluated without requiring explicit information from distant vertices (midpoint displacement) or other random features (Poisson faulting). This last property has the advantage of allowing (practically) infinite seamless growing of the covered area of a heightfield. This is very helpful if it is later decided that a heightfield should be larger than initially anticipated or a domain mapping distorts an area such that height information that would otherwise lie outside the evaluated area is now visible. Another advantageous property of noise synthesis is the flexibility to evaluate areas of arbitrary shape and size. All techniques other than noise synthesis and Poisson faulting work best when (or even require that) a square of a power-of-two size is evaluated. Noise basis functions are described in more detail in Chapter 5.

Parallelism is another important factor and is related to the possibility of evaluating samples independent of values of distant vertices. Recent desktop PCs have a graphical processing unit (GPU) powerful enough to be used as a general-purpose parallel SIMD-like data processor. All above fractal synthesis techniques can be implemented to execute on the GPU to benefit from this parallelism. But the attainable speed-up would heavily depend on the complexity of the specific algorithm and the dependency between the required data structures. Because the execution speed of terrain generation and editing tools is critical in the design loop, as explained in Section 2.2, it is important to be able to use the extra processing power offered by GPUs efficiently. Typically used

noise functions (e.g. Perlin noise, Section 5.2) can be evaluated completely independent of distant sample read-backs and has few data dependency relations. Therefore, this noise function will benefit greatly from this parallelism. Likewise, midpoint displacement might benefit from hierarchical execution on the GPU using intermediate heightfield 'textures', doubling in size at each subdivision level. However, midpoint displacement is only efficient if (and only if) a full evaluation of a square heightfield is calculated. Estimating the speedup that can be attained through the use of a GPU for the different algorithm is difficult, because no papers or terrain generating applications were found that use the (full) capabilities of the GPU. But because noise synthesis (or, more specifically, Perlin noise construction) has been shown to execute efficiently on the GPU, it is safe to say that none of these other approaches would gain preference over noise synthesis.

Other building blocks that can be used for noise synthesis, like Voronoi noise, are more expensive to evaluate and might therefore be less applicable in a general sense. But, offering other noise building blocks in a toolkit offers the designer the freedom to choose whether or not to use it. This fits nicely with the idea of offering different modular building blocks, function mapping and cascading options to let designers compile their own heightfields and reusable functions. When the interface of this modular design is made to be as consistent and transparent as possible, this idea becomes both powerful and intuitive. Because noise synthesis is very flexible and powerful, other fractal synthesis techniques like Poisson faulting, Fourier synthesis and midpoint displacement may not need to be incorporated in a toolset.

By separating erosion from the creation of procedural terrain algorithms, and allowing erosion to be used as a post-processing step as discussed on page 21, erosion can be applied to any terrain. However, erosion techniques that do not adapt a separately created procedural terrain, but create the whole terrain in one step like discussed in [CHIB98], are relatively inflexible and generally not worth considering. See Section 7.3 for a discussion of different post-processing erosion algorithms.

Non-fractal techniques like discussed in Section 4.5 might also be made available to designers. However, the two algorithms presented in that section are of limited quality. The first creates a good looking network of rivers but is not very good in creating natural terrain details. The second algorithm suffers from the same directional artifacts as regular midpoint displacement. However, for the creation of terrain types with a lot of rivers, they might still be the preferred choice.

5 Noise Basis Functions

As described in Section 4.3, generating procedural content through noise synthesis is accomplished by adding band-limited noise functions. Varying the added frequencies (scales) and the characteristics of the noise function will have a large impact on the result. For this reason, different noise functions have been developed as basis functions, almost like building blocks, for the construction of procedural content. For a synthesized result of a specific power spectrum (e.g. fBm surfaces), the ideal noise function would produce narrowly band-limited, stationary (translation invariant) and isotropic (rotation invariant) noise. But as a building block for artistic or natural effects, other 'noise' types might be preferred in order to achieve a desired look. This chapter discusses different noise basis functions for use in noise synthesis-based terrain generation.

5.1 Fourier Synthesis

Fourier synthesis was already discussed in the previous chapter, but separating it in multiple band-limited noise building blocks allows it to be used for noise synthesis, adding to its flexibility. Band-limited noise is easy to define in the frequency domain. The amplitudes of the frequencies are randomly chosen using a probability distribution of the desired band-limited power spectrum. Then, an inverse Fourier transform is performed to get the random noise in the spatial domain using either DFT (Discrete Fourier Transform) or FFT (Fast Fourier Transform) [COOL69]. FFT can be more efficient than DFT when a large 'patch' of noise evaluations is needed all at once (explicit construction). When only single samples are needed, DFT is preferred (implicit evaluation) [EBER03, p. 49]. However, calculating a FFT or DFT is relatively compute intensive, making Fourier synthesis less practical than alternatives.

5.2 Lattice Noise

Lattice noise functions assign uniformly distributed (pseudo)random numbers at every point in space whose coordinates are integers, creating a regular lattice of random numbers. An interpolation scheme that uses the assigned random numbers of nearby neighbors at integer coordinates is applied to calculate the output value for an input coordinate. The interpolation scheme has the effect of a low-pass filter. And because the highest frequency of lattice noise is limited by the lattice density, lattice noise is band limited.

Depending on the application's requirements, the random numbers assigned to every integer coordinate can either be precalculated and stored explicitly, or evaluated at request by hashing the integer coordinate to retrieve a random number. For the hashing technique, two 1D lookup tables are used. The *H* table is a precalculated random permutation of the set of all integers in the input domain of size *n* (typically a power of two). The *V* table is also of size *n* and contains random numbers in the range *[-1, 1]*. Then, a pseudo random value can be calculated by evaluating $V(H((x + H(y)) \bmod n))$ in the case of a 2D integer coordinate [PERL85].

Perlin Gradient Lattice Noise

Perlin noise is perhaps the most well known noise type, introduced in [PERL85]. Here, the random numbers at the integer coordinates do not represent the points through the noise function, but rather, gradients at these points. The returned value at all integer coordinates is per definition zero. All non-integer coordinates are calculated by interpolating between the gradients of the 2^d closest neighbors at the integer coordinates, with d being the dimension of the coordinate space (two in the case of heightfields). For gradient noise, the V table contains random gradients which are random vectors uniformly distributed on the d-dimensional unit (hyper)sphere.

Perlin originally proposed using a linear interpolator [PERL85], but later proposed a cubic [PERL89] and quintic [PERL02] interpolation spline to achieve C^1 respectively C^2 continuity. Higher order interpolation is slightly more compute intensive but, depending on the application, can be worth the extra effort. See Figure 5.1. See [PERL02] and [PERL04] for a more elaborate discussion. The power spectrum of gradient noise has little low-frequency power and is dominated by the frequencies that are near to one-half (on an integer-spaced lattice). In other words, it is fairly well band-limited.

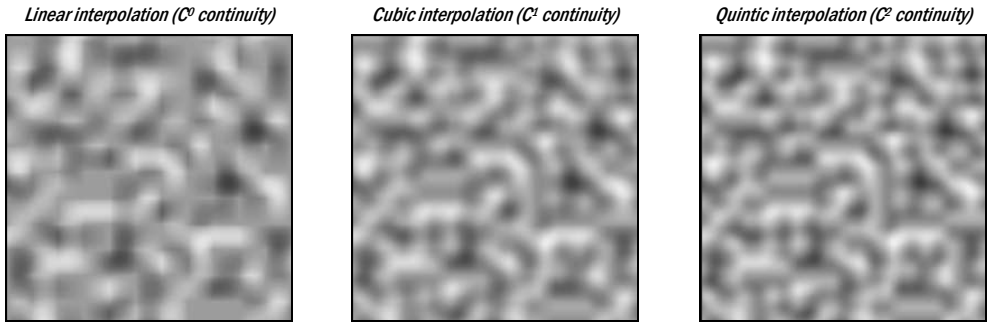


FIGURE 5.1 Different gradient noise interpolation schemes

Wiener Value Lattice Noise

Unlike gradient noise, value noise lets the random numbers assigned to the integer coordinates be the returned noise values at these points. Non-integer coordinates are calculated using an interpolation scheme. Like Perlin Noise, linear interpolation would result in visible 'boxy' artifacts. Interpolation in normally implemented using Catmull-Rom splines. This interpolation scheme needs more samples of the neighboring lattice points (4^d neighbors for d-dimensional lattice space) than gradient lattice noise (2^d neighbors). Value lattice noise has more power in the lower frequencies than gradient noise and is therefore less suitable as a band-limited noise octave. For more information on the value lattice noise, mixing value noise and gradient noise, and other lattice noise functions, see [EBER03, p. 67].

5.3 Sparse Convolution Noise

Lattice noise can have axis-aligned artifacts. To prevent this, sparse convolution noise first places randomly distributed impulses [LEWI89]. Then, filtering is applied using a low-pass convolution kernel. The resulting noise power spectrum can be controlled by the filter kernel and is related to the kernel's power spectrum. A common implementation of the filter kernel is a Catmull-Rom spline. The power spectrum of sparse convolution noise resembles a (scaled) power spectrum of value lattice noise. Even though convolution noise is of higher quality than lattice noise functions, it is (for the non-mathematical purpose of terrain generation) not worth the increased computing time.

5.4 Voronoi Diagrams

Even Voronoi diagrams have been used as band-limited noise functions [WORL96]. Like sparse convolution noise, the first step in constructing this type of noise is picking random points as a Poisson process. Then, a sample's value can be evaluated by calculating the weighted sum of the distances to the top d closest neighbors. That is,

$$N(x,y) = \sum_{d} w_d \left\| N - R_d \right\|$$

with N being the coordinate evaluated, R_d being the random point that is d^{th}-closest to N and w_d the weight for the d^{th}-closest neighbor. See Figure 5.2 for examples of Voronoi noise that are interpreted as heightfields. Although Voronoi noise isn't a very good approximation of band-filtered white noise, its average cell size can be controlled by the random point density. This makes it a noise building block of band-limited feature scale and, therefore, does have its uses in procedural (heightfield) noise synthesis. More natural shapes appear when combined (cascaded) with domain distortion functions. See Figure 4.8.

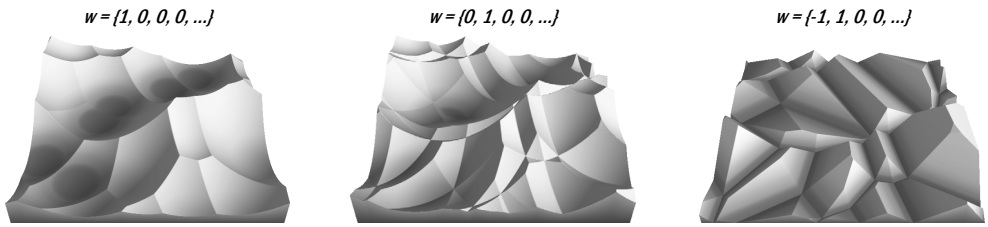


FIGURE 5.2 Voronoi diagram 'noise'

Creating Voronoi noise is relatively compute intensive. However, the shape of its typical features is not easily approximated using less compute intensive techniques. For this reason, it might still be appreciated by designers to offer an option for Voronoi noise in a toolbox.

5.5 Preliminary Discussion

Because Perlin noise is fairly well band limited, has few artifacts and is fast to compute, it is currently the preferred choice of many applications that allow procedural creation of heightfields or other types of content (e.g. textures). Also supporting Voronoi noise can be helpful to create ridged mountains or other sharp-edged smaller features that are difficult to produce with other types of noise. When both of these techniques are available to designers, they create a sufficiently solid base to designs terrains with, when combined with the summing, distortion and mapping techniques discussed in Chapter 4.

6 Heightfields by Example

This chapter discusses an alternative idea for designers to generate heightfields. Instead of generating new terrain by tweaking a number of parameters, the designer is enabled to quickly generate new terrain that is similar to a selected area of already created terrain. A designer would have to select an example area (the exemplar) and start an algorithm that could synthesize similar, but not identical, terrain somewhere else (the destination area). See Figure 6.1. This would allow a designer to reproduce the properties of imported real-world or previously created features, without tweaking any of parameters that would otherwise be required for procedural tools to approximate the desired terrain properties. It also makes it possible to create new terrain based on scanned heightfields (i.e. DEMs) of real terrain. Such a tool would fit nicely between low-level copying tools and purely parameterized procedural heightfield generation.

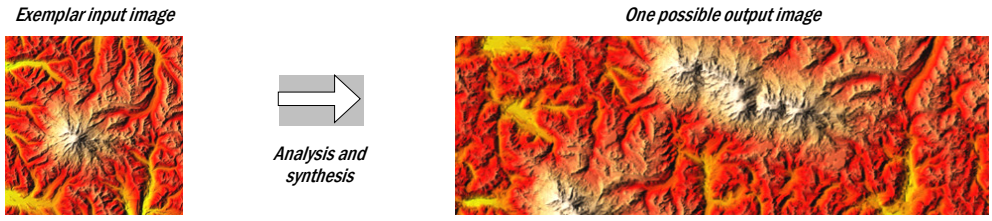


FIGURE 6.1 Texture by example synthesis. From [LEFE05]

A growing set of 2D image synthesis algorithms that can create new images from exemplar images has been developed in recent years. As explained in Section 3.1, heightfields have a direct relation to 2D images. This enables techniques that are aimed at 2D image synthesis to be interpreted as useful terrain creation techniques. So, using these techniques to synthesize heightfields is a natural extension. Note that this chapter adopts the 2D image-related terminology and uses the 2D example images from the original papers. Specifically, the words images and textures are used interchangeably and denote a 2D matrix of color or grayscale values. A pixel represents a local element of this matrix at an integer *(x, y)* coordinate (i.e. column-row pair).

This chapter only discusses a few of the many algorithms available. The quality of the results obtained from these algorithms can vary greatly. See Figure 6.2 for a visual comparison of a number of these algorithms for a scale-like exemplar image. It must be noted that the applicability of these algorithms depends on the type of texture that needs to be synthesized. Algorithms that work fairly well for images that contain different types of features which have sharp edges could perform badly on relatively smooth textures (Figure 6.2, middle row) by creating unwanted seams. Likewise, algorithms that always create seamless results can create results of less quality for exemplar images that contained sharp-edged distinct features [ASHI01].

Terrain is generally smooth and contains only few or no extremely sharp edges. For this reason, only algorithms that are better at synthesizing seamless and smooth textures were chosen to be surveyed in this chapter. The first algorithm is one of the oldest texture synthesis algorithms and is

relatively easy to implement. The two subsequent algorithms describe variants of this algorithm designed to speed up the synthesis process.

But before going into the details of these algorithms, Laplacian and Gaussian image pyramids are explained in Section 6.1. Image pyramids are part of some texture synthesis algorithms and other so-called multi-resolution algorithms with the purpose of speeding up the algorithm and to be able to cope with features on multiple scales. For example, the multi-resolution blending technique that will be discussed in Section 7.4.2 uses multiple pyramids to blend different heightfields together.

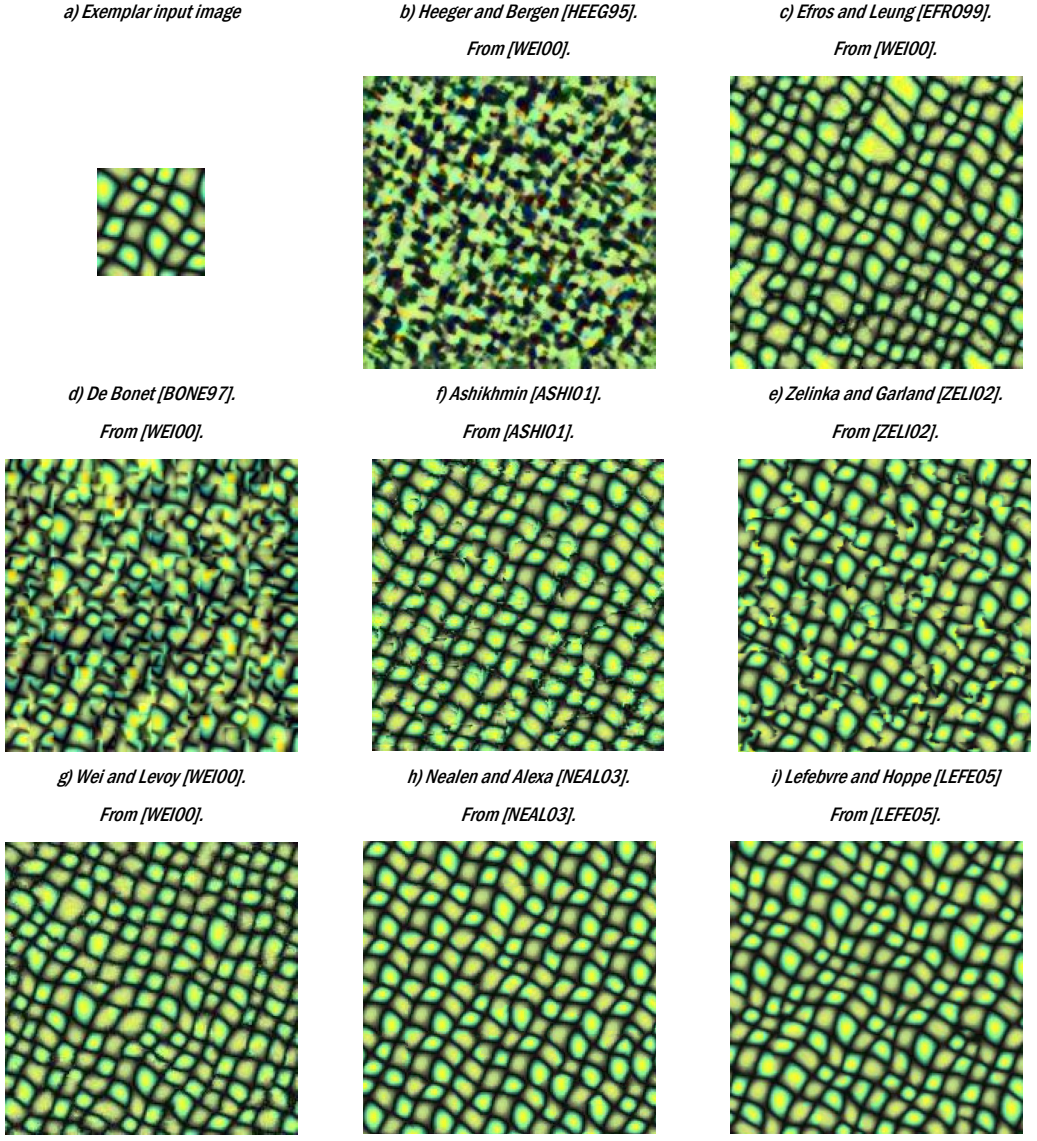


FIGURE 6.2 The topleft image is the exemplar image used to synthesize all other images shown. The other two images on the top row show the result of algorithms that do not correctly copy the structure of this exemplar. The images on the middle row are created by algorithms that produce visible seams. The bottom row shows the result of algorithms that produce perceptually similar textures without visible seams

6.1 Image Pyramids

Comparing different image areas for the amount of similarity is part of all texture-by-example synthesis algorithms. But many images, including 2D heightfield images, have features on varying scales and, therefore, need different window sizes to use for their local similarity measurements. One way to detect all features is to use small, as well as medium and large windows for these measurements. But processing large windows is very compute intensive. Image pyramids [ADEL84] are used often instead. The idea of an image pyramid is not to scale the actual window size of an operation in order to be able to cover different scales, but rather to downscale the input image to multiple power-of-two scales and use these as inputs to an operator that uses a fixed-sized window instead. This idea is the basis for many multi-resolution algorithms.

The image pyramid assumes an input image of size $2^n \times 2^n$ and constructs a pyramid of $n+1$ levels with a $2^l \times 2^l$ image at level l, $0 \leq l \leq n$. The image at level n is the original image. An image at level l can be constructed by downscaling (reducing) the image at level $l+1$ by a factor of two. A filter with a (small) fixed-sized low-pass kernel is convolved before every resolution reduction. This filter filters out all frequencies higher than half the sampling rate, as required by the Nyquist-Shannon sampling theorem, to prevent aliasing. Often, a small 5×5 kernel is used as an approximation to a 2D Gaussian kernel. For a faster, less accurate, implementation, a 2 x 2 averaging kernel is sometimes used. In effect, the different pyramid images can be seen as (scaled) approximations of low-pass Gaussian filtered images with successively doubled radii. For this reason, this type of pyramid is called the Gaussian image pyramid. The construction procedure is depicted in the top half of Figure 6.3. See Figure 6.7 for an example of a Gaussian pyramid.

The images in the Gaussian pyramid are low-pass filtered images. However, the Gaussian pyramid can be processed further to create a band-pass filtered pyramid of images. This band-limited pyramid approximates the Laplacian of Gaussian (LoG), or simply the Laplacian, at different (successively doubling) scales, creating a decomposition into wavelets. The level 0 of the Laplacian pyramid is equal to level 0 of the Gaussian pyramid. The k^{th} Laplacian layer, $1 \leq k \leq n$, can be constructed by subtracting the $(k-1)^{th}$ Gaussian layer from the Gaussian k^{th} layer, after up-scaling (expanding) the $(k-1)^{th}$ Gaussian layer to $2^k \times 2^k$. The interpolation scheme used for expanding can be chosen freely. Construction of the Laplacian pyramid from the Gaussian pyramid is shown in the bottom half of Figure 6.3. Note that the Laplacian pyramid allows lossless reconstruction of the original input image using n cascaded expand-and-sum operations, effectively summing over all Laplacian levels that are recursively rescaled to $n \times n$.

The Laplacian pyramid is not used in this chapter, but it is used in many other computer graphics fields like data compression and multi-resolution editing. Multi-resolution editing of heightfields is discussed in Section 7.4.2.

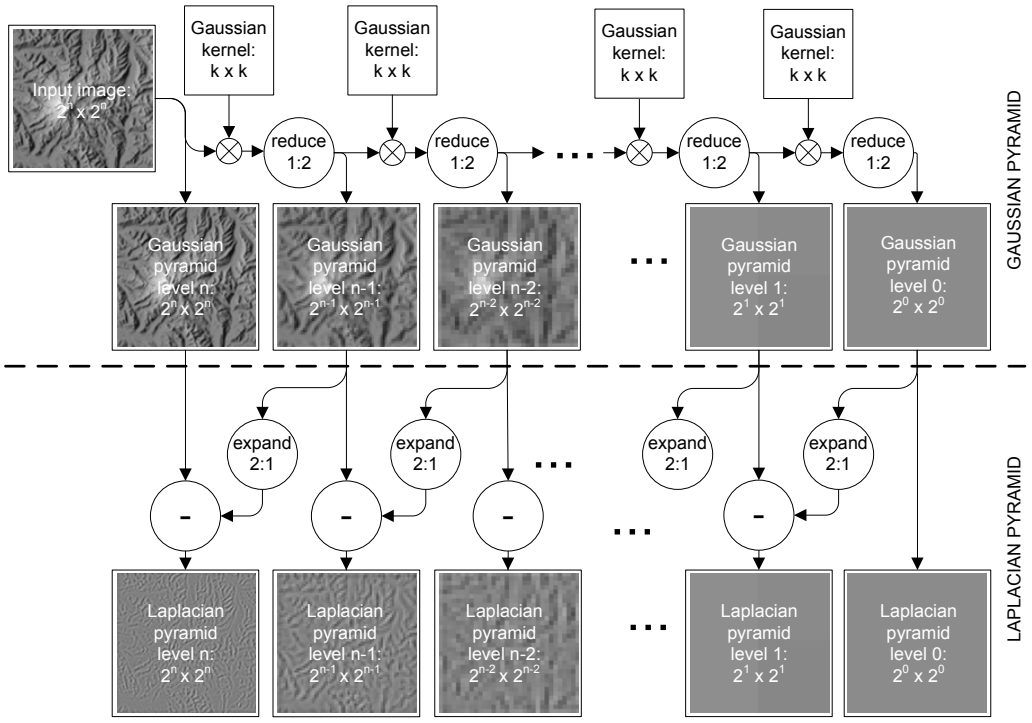


FIGURE 6.3 Construction of the Gaussian and Laplacian image pyramid

6.2 Explicit Neighborhood Window Texture Synthesis

Returning to the topic of texture synthesis, a relatively intuitive and simple algorithm was introduced by Efros et al. that grows a new texture pixel by pixel [EFRO99]. This work models a texture as a Markov Random Field (MRF). Consequently, every pixel value depends statistically on the values of the neighboring pixels for a given neighborhood size. A neighborhood is defined as square window centered around its input pixel coordinate. This relation is strict in the sense that a pixel's value is assumed to be independent of values of all pixels outside the neighborhood. Hence, the neighborhood window size is required to be of a size similar to an image's features in order to effectively detect and reproduce its features and structure. Too small, and the structure is lost. Too large, and the synthesized textures contains features that might be too structured. See Figure 6.4.

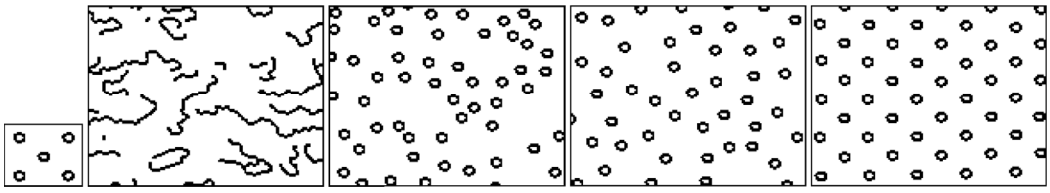


FIGURE 6.4 From left to right: The exemplar and four synthesized textures with a neighborhood window of 5, 11, 15 and 23 pixels wide, respectively. From [EFRO99]

To determine the value of the pixel at each coordinate p in the destination area D, the exemplar E is exhaustively searched for close matches of exemplar neighborhoods $w_e(s)$ with the destination

pixel's neighborhood $w_d(p)$. The amount of similarity between the pixels of two neighborhoods is measured by a similarity distance measure d. These neighborhoods are defined as square windows centered around a coordinate. There is no guarantee that a perfect match will be found (i.e. $d = 0$), because D might start off with areas already partly defined and, also, the algorithm introduces variations itself. A close match is defined as a pair of some s and some p with $d(w_e(s), w_d(p)) < (1 + \varepsilon) \cdot d_{min}$, with d_{min} being the smallest similarity distance found between $w_d(p)$ and all $w_e(s)$. See Figure 6.5. $\Omega(s)$ is the set of coordinates in E that have a closely matching neighborhood when compared to $w_d(p)$. Or, in mathematical notation:

$$d_{\min}(p) = \min_s(d(w_d(p), w_e(s)))$$
$$\Omega(p) = \{s \mid d(w_d(p), w_e(s)) < (1+\varepsilon) \cdot d_{\min}(p)\}$$

ε controls the maximum allowable quality of the elements in $\Omega(p)$, relative to the best match. Consequently, the set size of $\Omega(p)$ will grow with larger values of ε. A larger $\Omega(p)$ set creates less exact but more varying textures. A value of 0.1 is chosen for ε in [EFRO99].

The set $\Omega(p)$ contains coordinates of pixels in S that have a neighborhood that closely matches the neighborhood of D's p. Hence, the (color) value at p is best set to one of the colors at the pixel coordinates in $\Omega(p)$. A histogram of pixel values is created from the pixel values at the $\Omega(p)$ coordinates. This histogram is then sampled uniformly or weighted by d to choose the value at p.

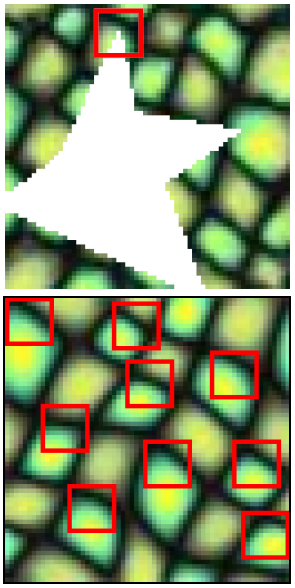


FIGURE 6.5 Nine neighborhoods in E (bottom) that closely match the 9x9 neighborhood in D (top)

The similarity distance measure is taken to be a weighted sum of squared differences between all filled-in individual pixels of $w_d(p)$ and $w_e(s)$ for some p and s. Pixels in a neighborhood that are not filled in yet are not considered in the distance measure. The weights are picked to resemble a 2D Gaussian kernel, centered around the neighborhood window's center, to give differences between neighboring pixels near the center pixel more weight. Consequently, differences in local structures take precedence over distant structures.

The coordinate p is picked at each iteration from the set of all pixels in D that are not yet filled in. The coordinate p from this set that has the most pixels in its neighborhood in D filled in is selected to be filled in next. In effect, the texture is grown outward from areas that are already filled in. As an initialization step, a random pixel can be copied from E to D to function as a growing seed if D was initially completely empty.

The main advantage of this algorithm is its algorithmic simplicity and the decent quality of its results. Its main disadvantage is the time required to synthesize a new image, possibly taking several minutes to synthesize an image of a typical size (e.g. 256 x 256). It is most appropriate for

textures that contain regularly sized features because of its fixed neighborhood window size. In some cases, this algorithm is known to grow garbage (areas of different structures, e.g. noise). Also, the quality of the result depends on the exact sequence of picked *p* coordinates. This is especially true when the algorithm is used to fill gaps in *D* instead of filling *D* completely.

6.3 Multi-resolution Texture Synthesis

In [WEI00], several improvements to the previous algorithm are suggested in order to speed up texture synthesis. For one, it applies multi-resolution techniques to improve the image quality and to be independent of a user-selected neighborhood window size parameter. But first, differences in the traversal order and shape of the neighborhood window shape are discussed.

In contrast to the algorithm discussed in Section 6.2, this algorithm traverses all coordinates *p* in *D* using a fixed raster scan ordering traversal to synthesize *D*. Consequently, it can only be used to fill *D* completely, not to fill gaps in a partly filled *D*. *D* is treated to be toroidal, creating a texture that matches its opposite sides. This allows neighborhoods to 'wrap around' when pixels outside the boundary are needed. To create a random texture, the two rightmost columns and the two bottommost rows are pre-filled with noise to be used for the neighborhood matching at its opposite sides. Hence, by using an L-shaped 5 x 2½ neighborhood window, only these noise pixels and all already synthesized pixels will be used during similarity comparisons. See Figure 6.6. This change makes traversal and similarly comparison simpler without degrading then quality, when compared to a 5 x 5 implementation of [EFRO99].

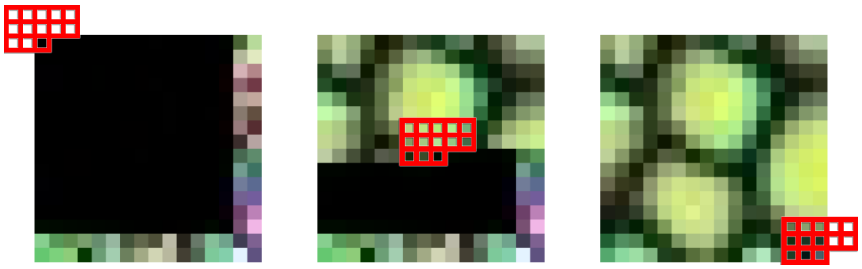


FIGURE 6.6 From left to right: The 5 x 2½ L-neighborhood and the synthesized result at the first, the middle and the last iteration of the algorithm. Note that the red mask uses wrap around to look up a pixel at the opposite side when such a neighborhood's pixel lies outside the image (left image). This wrap around is not visualized here. From [EFRO99]

While the previous algorithm uses a single user-defined neighborhood size, [WEI00] uses a precalculated Gaussian pyramid of *E* to synthesize a pyramid of *D*. During construction, neighborhoods in *E* and *D* are compared on multiple pyramid resolution levels simultaniously. As a result, features of all sizes are automatically detected. Starting with the lowest resolution image in the pyramid, the single-resolution synthesis process is applied similar to [EFRO99] in Section 6.2, now using the raster scan traversal and the L-neighborhood. The used distance measure simply compares the neighborhoods at that first level for both *E* and *D*. Because this level is a downscaled

version of the higher levels, the 5 x 2½ neighborhood would cover a much larger area on the original level, detecting much larger features.

Next, the subsequent higher-resolution layers in the pyramid are synthesized layer by layer, from coarse to fine. But instead of only using the 5 x 2½ neighborhood window at coordinates *s* and *p* at each of these levels, the similarity neighborhood is extended further with a 3 x 3 neighborhood window at each of the previously calculated coarser, lower-resolution levels, accumulating their similarity distances. See Figure 6.7. The *s* and *p* coordinates for the current layer are halved at each subsequent layer to compensate for the resolution reduction. The lower-resolution levels with the fixed 3 x 3 neighborhood windows relatively cover increasingly large window areas when going from the currently synthesized image layer to the top most (coarse) layer. Together, these enforce a close match between the neighborhoods at *s* and *p* at different neighborhood scales.

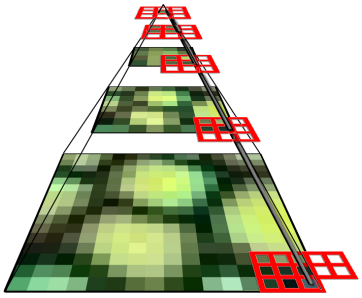


FIGURE 6.7 Neighborhoods used for the calculation of the last pixel in layer 4 of a full Gaussian pyramid

The Cartesian product of all pixel values in a neighborhood can be interpreted as a vector in a high-dimensional domain. This allows each possible neighborhood in *D* or *E* to be seen as a point in this domain. Then, finding the closest match is equivalent to searching the nearest point in this high-dimensional domain. Several search algorithms are available that would speed up such a search. Tree-structured vector quantization (TSVQ) is suggested in [WEI00]. This creates a binary-tree-structured codebook that is trained on the exemplar's neighborhood vectors and allows very efficient traversal to search the approximately closest match to a vector from *D*. The size of the codebook can freely be chosen and is a tradeoff between traversal efficiency, accuracy and memory requirements. Without the TSVQ acceleration, the algorithm described in this section is about 4 times faster than the algorithm proposed in [EFRO99]. With TSVQ acceleration, it is about two magnitudes faster than [EFRO99] and has *O(log N) / O(N)* times the algorithmic complexity, where *N* is the total number of exemplar pixels.

6.4 Parallel Controllable Texture Synthesis

Pixel-based texture synthesis is very data intensive and fairly simple to implement. This would make it ideal for parallel execution on a powerful GPU. However, the algorithms above have the drawback of requiring sequential construction, as the output of one iteration serves as input to the next. In [LEFE05] a texture synthesis algorithm is described that does allow highly parallel execution.

Like [WEI00], it uses multi-resolution levels of the image to work on different scales using a variation of the Gaussian pyramid, called the Gaussian stack. From the lowest-frequency level up, it calculates the next level of *D* in three phases, level by level. First, the previous level is sampled up in order to double its resolution. Secondly, the up-sampled information is jittered to introduce variation. Lastly, the level is iteratively corrected to recreate neighborhoods similar to those found in *E*.

But these steps are not executed on pixel color information in *D*. Instead, another pyramid *S* is used. *S* contains coordinates that point to pixels in the exemplar *E*. This allows *D* to be constructed from *S* by calculating *E[S]*. The advantage of working on a separate coordinate map is that this allows upsampling and jittering coordinates from a lower (coordinate) level, while full-spectrum non-degraded image detail can still be looked up. The 2D coordinates in *S* can be encoded as colors for visualization and fast GPU processing, using the red and green components as X and Y values, respectively. See Figure 6.8.

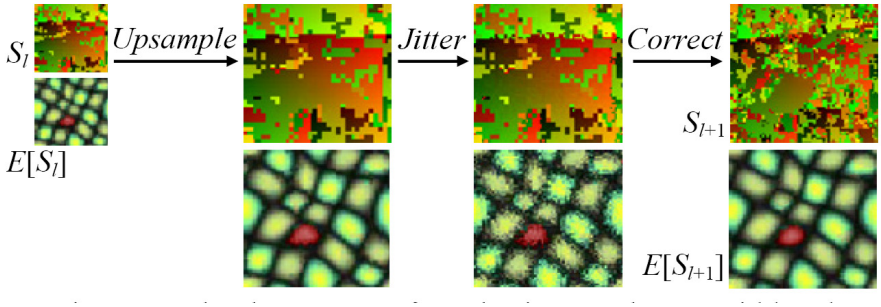


FIGURE 6.8 The three phases of construction of the next layer. The images on the top row are coordinate maps. From [LEFE05]

In the upsampling phase, S_{i+1} is simply calculated from S_i by doubling and interpolating the coordinate values in S_i. The jittering phase introduces randomness by perturbing S_{i+1} using a deterministic pseudo-random hash function (e.g. Perlin noise). Note that the amount of perturbation can be varied per layer, allowing for fine control over the exact type of variation. Also, when the jittering phase is left out, the synthesized image will closely match *E* or even consists of a (multiple of) exact copies of *E*, depending on whether *E* is toroidal. See Figure 6.9.

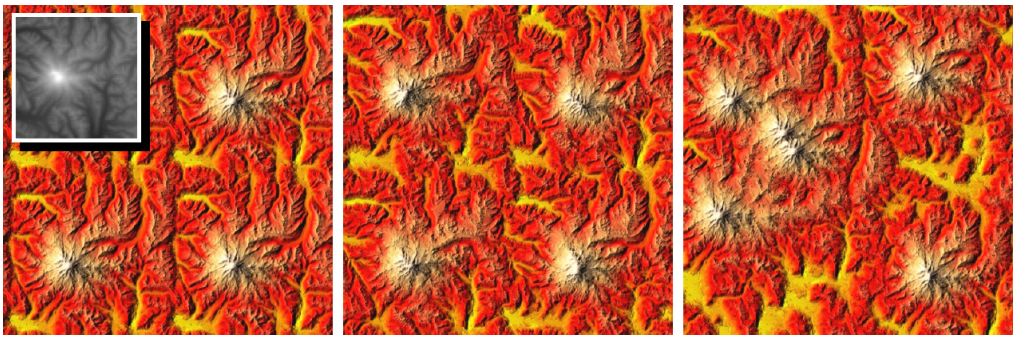


FIGURE 6.9 Synthesizing three versions of *D* of twice the width and height of *E* (the gray image). From left to right: No perturbation, perturbation at the higher (finer) levels and perturbation at the lower (coarser) levels of the image pyramid *S*. From [LEFE05]

These first two phases can be implemented easily and efficiently on parallel architectures. The last phase contains the actual neighborhood matching part for all pixels, which contains many dependencies. Previous algorithms solved this by calculating and updating it sequentially for the different pixels. The algorithm in [LEFE05] introduces an iterative subpass approach that allows highly parallel execution. Each subpass updates an interleaved subset of S_i by searching for 5 x 5 neighborhoods in *E* that closely match the neighborhoods in *E[S]* for the pixels in the current subset of *S*. See Figure 6.10. To do the neighborhood matching efficiently, the exemplar is preprocessed (e.g. TSVQ) to allow a fast lookup of closely matching neighborhoods for all pyramid levels of *E*. In total, k^2 subpasses are used, each responsible for a regular, interleaved subset of *S* of non-(von Neumann) neighboring pixels, with typically *k = 2 or 3*. The partition into subpasses allows neighboring pixels in *S* to be causally dependent on the result of previous subpasses, while the update of non-neighboring pixels is executed in parallel at each subpass. In practice, results from this approach are often better and more isotropic than completely sequential approaches because there is no single explicit sequential construction order. When required, the quality can be further improved by applying the correction phase multiple times.

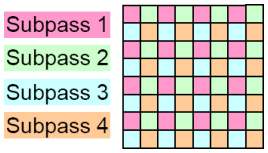


FIGURE 6.10 The interleaved update pattern of 2^2 correction subpasses. From [LEFE05]

A unique and useful control supported by this algorithm is feature drag-and-drop. By letting the user influence the perturbations in the jitter phase, random variation can be locally replaced by exact placement of a feature found in *E*. For example, a mountain top in Figure 6.9 can be relocated from one position to another. To support this, yet another image pyramid can be used to look up the local perturbation. This image pyramid would initially be filled with random values, but can be replaced locally with specific coherent values, forcing a lookup for *D* from the desired area in *E*. And because the correction phase is still applied to *S*, the result remains seamless. However, this control is limited to spatially distant adjustments as earlier adjusted pixels in the perturbation image would otherwise be overwritten by the latest change.

The exact speedup accomplished by this algorithm depends on many factors. But as a rough estimation, the algorithm can be said to be about three magnitudes faster than [WEI00] for typical sizes (128 x 128 and 256 x 256) when executed on GPUs from around 2005. Synthesizing a 256 x 256 image takes about 25 ms.

6.5 Preliminary Discussion

For this literature study report, it was not possible to run different algorithms on terrain heightfields to compare the quality of their results. Having such a comparison would make choosing between algorithms much easier. However, there is good reason to assume that the last two algorithms described above would produce heightfields of fairly good quality. Not only do they produce good results for smooth features, they also search for matching features in the exemplar at multiple scales. Both properties are expected to be needed for good terrain synthesis. The first property is important because terrains are, on average, locally fairly smooth and contain few or no really sharp ridges. The latter property is important because terrain is generally fractal, having features on all scales.

The three algorithms discussed in this chapter were ordered to be increasing both in algorithmic complexity and in synthesis speed. The last algorithm uses the parallel processing capabilities of the GPU to speed up synthesis. Whereas the first algorithm could take up to several minutes to complete the synthesis of an image, the third algorithm does this in a fraction of a second. This makes the third algorithm the only algorithm that could be used as an interactive tool for a level designer on today's hardware.

It is expected that a tool that would allow a designer to copy properties of an exemplar area into a destination area of arbitrary shape and size would be very useful. However, the second and third algorithms discussed in this chapter are only capable of synthesizing a rectangular patch, without considering the neighboring terrain at the patch's boundary. Blending techniques discussed in Section 7.4 can be used to blend new terrain into already existing terrain. However, it might be possible to extend these algorithms to directly support natural transitions between existing and synthesized areas. More research and experiments would be required to verify this statement.

As a last note, the distance measure used by these algorithms was chosen for its usefulness for synthesizing 2D images but might prove to be suboptimal for heightfields. For example, it might be found that the derivative of the height in a heightfield might be more important than the (small) perceptual importance of the derivative in a 2D image. Most algorithms use the squared error measure for the neighborhood comparison, but this often can easily be replaced by other measures. It would require some experimentation to verify that other measures might improve the perceptual quality of synthesized terrains.

7 Terrain Geometry Editing

Chapters 4 through 6 discuss the procedural synthesis of new terrain. Some of the currently available level edit tools already allow some form of procedural terrain synthesis. Having such a tool helps a designer to create a rough outline of the whole terrain required for a game level. However, these tools only offer global, high-level parameters, making it hard to control exact placement of different desired landscape features (e.g. mountains and lakes) throughout the landscape. Even if one feature (e.g. a mountain) is generated to the liking of the designer by tweaking procedural parameters, it is very unlikely that all other simultaneously generated features in that generated landscape are more or less exactly as planned. Therefore, when a designer requires somewhat exact placement of specific features at specific locations he has no other choice than to use the only other set of tools that is typically available to further sculpt the procedurally generated rough outline. This alternative set of tools typically allow for low-level operations that only make simple local adjustments to the heightfield. Examples of these low-level tools are mouse-controlled local vertical heightfield pushing, pulling and leveling operations that operate at a specified location within a specified radius. However, once manual changes have been made to a terrain, the high-level synthesis tools are no longer of use; applying synthesis algorithms would otherwise overwrite all manual changes.

Low-level operations can be ideal when only small changes are needed. And indeed, every type of terrain can be created with these tools by a good level designer given enough time. But it is clear that tools that fit somewhere between the high-level procedural terrain synthesis tools and the low-level local operation tools certainly would find their use in level design.

For this purpose, four types of editing tools are surveyed in this chapter. First, the terrain editing tools that are typically the only non-procedural tools available to today's designers are covered. Secondly, simple extensions that allow terrain warping in uncommon ways are discussed. Thirdly, erosion algorithms are introduced in Section 7.3. These complement the other tools by offering the creation of more physically correct features that can easily be carved out where the designer desires to. Algorithms that are capable of integrating an area of one terrain into another are discussed in Section 7.4. Such algorithms make it possible to reuse terrain synthesis tools at later stages of the level design, as the combinations of these tools can be used to synthesize and blend in terrain in designated areas of a level that still need work.

7.1 Simple Editing

Starting with low-level editing, this section gives an overview of the (only) terrain editing tools that are commonly available in today's level editor applications. These are typically used inside an application environment that is able to render a 3D preview of the level at real-time. The mouse is used to designate the circular area a tool should work on. Typically, a tool radius can be chosen to vary the size of the selected area. Other options include the tool strength (e.g. amount of change

per time unit) and the shape of any strength falloff towards the boundary of the circular area. Then, the terrain is edited by repeatedly changing the editing tool type and its options and then 'painting' or 'brushing' with these tools by dragging the mouse. Of course, mouse simulating hardware like drawing tablets can transparently be used instead if preferred. Typical tool brushes are:

Vertical push and pull These two tools simply slowly decrease and increase the height values that are currently under the selected circular area, respectively.

Smoothing A simple low-pass filter is slowly applied to the height values inside the the selected area over time. Smoothing can be used to smooth out areas that are too rough.

Leveling This drag tool sets all height values inside the (dragged) selected area to the height value that lied at the center of the selected area when the tool was activated (e.g. the left mouse button was first pressed). This is typically used to level (i.e. bulldoze) streets and the areas surrounding buildings.

Contrasting An (unsharp mask) sharpening filter is slowly applied to the selected area over time. As the opposite of smoothing, it can be used to roughen areas.

Noising Small random displacements are added to all height values inside the selected area over time. This is typically used to introduce some variation into terrain.

Like applying simple painting strokes, these tools can be used to create any type of terrain that is required. But of course, it takes skills to use these tools effectively. Also, creating levels this way is very time consuming. Nevertheless, this is all that is offered by most level editors.

7.2 Warping Tools

As discussed in Section 4.3, domain and range mapping support stretching and warping of landscape features. Examples of range mapping are simple glacial-like and canyon-like range adjustments. Domain mapping allows irregular and naturally flowing horizontal warping when coupled to a (Perlin) noise distortion field. These techniques could be offered as editing tools to the designer to simplify the creation of certain types of features, or simply to move a feature horizontally or vertically. Like the other proposed editing tools, a brush with a user-defined radius and falloff curve could be offered as a local interactive tool, adjusting the terrain while brushing with simple mouse strokes. The amount and variation of distortion could be made adjustable through the use of sliders and presets or could be coupled to a (cascaded) noise source.

Two different methods can be used for many of these brushes. The first is straightforward and consists of direct editing of the selected heightfield. The second is indirect editing, where the designer can paint an (invisible) mask field specifying the local strength of a tool's effect, similar to an alpha mask. Then, this mask field is used to locally (re)apply any of the operations discussed throughout this chapter to create a separate output heightfield. This has the advantage of supporting a simple effect eraser brush where the effect mask can locally be cleared with. Another advantage is mask scaling, globally amplifying or fading away the effect. Also, more advanced, non-linear techniques could use this mask to reapply the operation to the complete input instead of reacting to the latest change. Results created this way would be independent of the exact sequence of brush strokes.

When this idea of indirect editing is generalized, heightfield operations can be seen as a flow graph of operation and data nodes (e.g. blend nodes, file inputs, procedural heightfields and painted mask layers). Although this is a powerful paradigm, it is also difficult to implement efficiently in terms of memory and computational power, as explained in Section 2.5. It is especially difficult to do so when an operation requires multiple heightfield inputs. By allowing the designer to choose between direct editing and indirect editing through the use of mask layers, it is left up to the designer to choose the type that is most appropriate. Direct editing is fast and is less flexible. Indirect editing is more memory intensive and compute intensive, especially when many layers are used during editing. Collapsing a layer (i.e. applying the operator using the mask field, explicitly storing the result as a new heightfield and deleting the mask field and any other input fields) after being done with it might keep indirect editing workable at interactive speeds.

Because range and domain mapping derive a new heightfield from an original heightfield, it is expected that any direct feedback loop of the effects into the same heightfield by editing this heightfield will render these tools possibly less useful. For example, keeping your brush too long at the same location while using direct domain warping will result in a fully horizontally smeared patch under your brush, loosing all detail due to the repeated use. In contrast, by using a layered, indirect version, all original detail is maintained, as it effectively is a perturbed lookup into the original unaltered heightfield. And again, when the designer is content, the layer could be finalized and collapsed to preserve memory and improve performance.

7.3 Erosion Tools

Although the tools that are described above are very simple, the concept of brushing to edit terrain is not necessarily too primitive to be efficient for a designer. When the set of brush tools is extended to include more powerful and natural effects, this intuitive interface allows creation of more natural effects in less time. In this subsection, different terrain erosion brushes are suggested to simplify the creation of geological phenomena that would otherwise be laborious to achieve.

These brushes use simplified models of geological laws and observations to simulate different aspects of the real-world ongoing process of terrain erosion. Because it is essential to have tools working at interactive rates, as discussed in Section 2.6, many of the simulations mentioned in this subsection are only simple approximations of the actual geological processes. But nevertheless, impressive results can be created quickly with these algorithms.

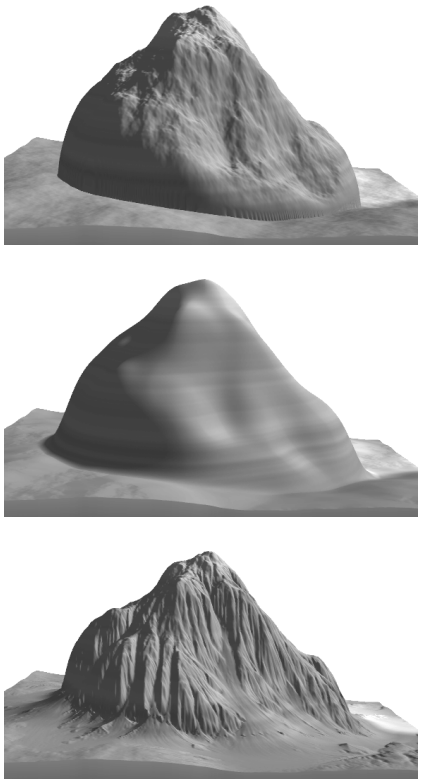


FIGURE 7.1 Different types of erosion. From top to bottom: unaltered procedural heightfield, thermal erosion and fluvial erosion.

Note that the algorithms discussed here were originally proposed as operations that are applied to the whole heightfield as an additional phase in the construction of procedural heightfields, as discussed in Section 4.4. But these algorithms are easily adapted to allow them to be applied only locally.

The erosion algorithms can be divided into two categories. The first simulates thermal erosion. This is the geological term used for the process of rock crumbling due to temperature changes, and the piling up of fallen crumbled rock at the bottom of an incline. The second type of erosion discussed is fluvial erosion. This type of erosion is caused by running water (e.g. rain) that dissolves, transports and deposits sediment on its path. See Figure 7.1.

7.3.1 Thermal Erosion

Thermal erosion, or thermal weathering, is the computationally least intensive type of erosion. However, the results created with this type of erosion are also less interesting. It simulates the process of loosening substrate which falls down and piles up at the base of an incline. This process is responsible for the creation of talus slopes at the base of mountains.

A simple thermal erosion algorithm is proposed in [MUSG89]. There, the heightfield is scanned for differences between neighboring height values that are larger than a threshold *T*. When found, the higher of the two neighbors deposits some material to the lower neighbor. If a height value has multiple lower neighbors, it distributes the deposition according to

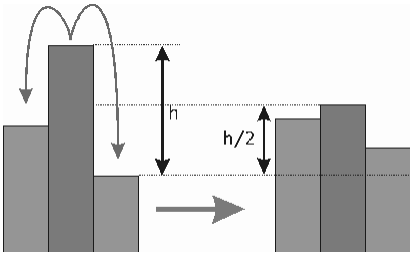


FIGURE 7.2 Thermal erosion deposition with $c = 0.5$, $T = 0$. From [BENE01b]

the relative differences. The amount of material deposited is a fraction *c* times the height difference between the neighbors minus *T*. See Figure 7.2. In effect, a maximal slope is enforced after enough iterations are executed.

The whole heightfield is updated at each iteration for these types of algorithms. Typically, the height values are read from the heightfield from the previous iteration, processed independently and stored to the new heightfield. As causal dependencies of interactions between values are not solved for but set independently for each height value instead, fluctuations in total mass and oscillatory heights can occur. But when the fraction *c* of deposited material is chosen small enough (e.g. 0.5), these effects will be sufficiently damped and barely noticeable. The advantage of such an implementation is that it allows parallel execution of all height updates within one iteration.

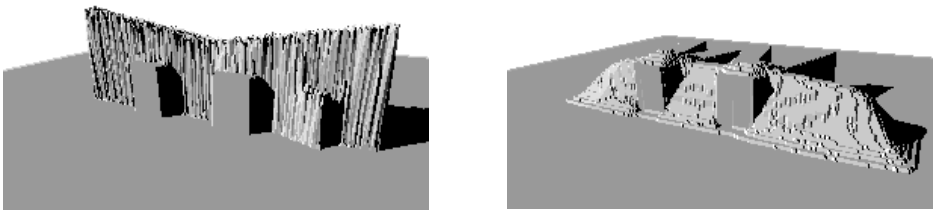


FIGURE 7.3 Before (left) and after (right) erosion was applied to the letter W consisting of a hard material and a layer of soft material on top.

A layered representation of heightfields was presented in [BENE01a] in order to cope with a different rock hardness at different earth layers. This allows different erosion rates at different locations and at different depths. The layers are represented as the relative height of different stacked material layers in a vertical geological core sample from the surface down to an absolute zero height. See Figure 7.4. Therefore, the height at the surface is the sum of the different layer lengths. Erosion is only applied to the surface, using the erosion parameters of the top layer. After this layer has locally been worn away, the next layer is exposed and so on. This can result in more varied results when the layers have been defined usefully. The experiment shown in Figure 7.3 shows a result that would be difficult to achieve with non-layered erosion.

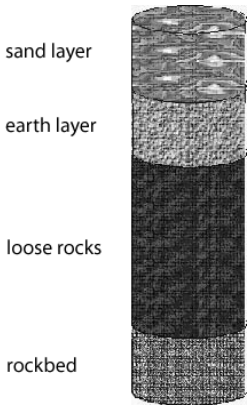


FIGURE 7.4 Example of a layered core sample. From [BENE01a]

7.3.2 Fluvial Erosion

Fluvial erosion, or hydraulic erosion, involves depositing water that can dissolve, transport and deposit suspended material on its way downhill. Examples of its effects are gullies and alluvial planes. But also the effects of alpine glacial erosion can be simulated if the right settings are used. A simulation of such a process is generally computationally more involved than thermal erosion.

These erosion algorithms can roughly be divided into two approaches. One is the simulation of individual water particles using a particle system, eroding the terrain under their individual paths. Simple physics rules are used to calculate the trajectory as it 'rolls' down and picks up and deposits sediment. The other approach uses a set of additional 'height'-fields that store the amount of water and the amount of suspended sediment within each grid cell. Then, a simulation step consists of updating these fields after locally exchanging the necessary information between neighboring cells. This type of grid-based local interaction is typical for all cellular automata algorithms.

A summary of [CHIB98] was already given in Section 4.4, where individual water particles are used to calculate water quantity, velocity and collision energy data fields which are on their turn used to update the heightfield. This process is repeated as many times as needed. Although the original paper used it to create new heightfields, it can be used to adapt a (previously generated) existing heightfield without any modifications.

One of the first grid-based fluvial erosion algorithms can be found in [MUSG89]. Each grid point *v* in the heightfield *H(v)* contains an additional water volume *W(v)* and a suspended sediment amount *S(v)*. Initially, a uniformly distributed amount of water is dropped (i.e. all of W is set to a non-zero value). When the local altitude plus the local water level is higher than the neighboring levels, the difference is transferred to the lower neighbors. See Figure 7.5. Flowing water will dissolve material and carry this sediment to its lower neighbors, up to a given sediment capacity constant times the (steepness-dependent) volume of the transferred water. Dissolving material is implemented by locally increasing the value in S(v) by the same (small amount) as decreasing *H(v)*. Likewise, depositing material increases *H(v)* at the cost of *S(v)*. When the local steepness-dependent sediment transfer capacity is larger than the amount of local sediment, more sediment is dissolved from *H(v)* and transferred. Likewise, when the capacity is smaller than the local amount of dissolved sediment, some of the sediment is deposited back to *H(v)*. Because the capacity is zero when the water level has reached a (local) equilibrium, all dissolved sediment is eventually returned to *H(v)*.

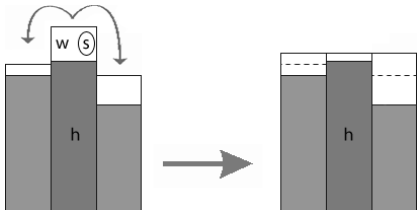


FIGURE 7.5 Fluvial erosion water transfer

In effect, this process will dissolve material from steep areas where relatively more water will flow and deposits the dissolved material again at flat areas downhill. As the geometry will force water to flow down non-uniformly, certain areas will be deepened and smoothed more than average. Areas that are deeper than their surrounding areas will receive even more water in the next iteration, amplifying this effect. As a result, distinguishable water streams are sculpted into the original heightfield. Note that water velocity, impact and evaporation are not considered here. Nonetheless, impressive result can be obtained with this algorithm given the right parameters and enough iterations. See Figure 7.1.

Several variations have been devised. In [BENE02b], water evaporation is included to limit the distance sediment can travel. Olsen suggests several tradeoffs between accuracy and speed in [OLSE04]. There, only the four neighbors in the von Neumann neighborhood are considered instead of the original eight neighbors in the Moore neighborhood. See Figure 7.6. Also, water is only transported from a high grid cell to its single lowest neighbor instead of being distribution among all its lower neighbors. Furthermore, it is assumed that water is fully saturated with sediment at all times and thus no separate *S(v)* sediment map is required. Although physically less correct, the results are still visually plausible.

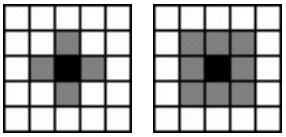


FIGURE 7.6 Neighboring cells (grey) of in the Von Neumann neighborhood (left) and Moore neighborhood (right)

A more physically correct model has been proposed in [BENE06] by discretely solving the Navier-Stokes equations to simulate water more realistically. Sediment transportation equations are added to simulate erosion. The equations are applied to voxelized (terrain) patches instead of heightfields to allow for a standard Finite Element Modeling approach to solve these equations. See Figure 7.7. Although results are impressive, calculation time currently prohibits its use in interactive applications.

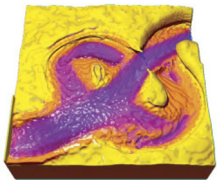


FIGURE 7.7 Oxbow lake-like features carved out by water simulation in a terrain patch. From [BENE06]

7.4 Terrain Blending

Another useful type of brush would be a copy brush. This would enable a designer to locally 'paint' a terrain from a different source heightfield onto the destination work terrain. Consequently, procedural techniques might be used in later stages by blending any desired parts of newly generated terrain into a project. Such a copy brush could be accomplished in different ways, varying from the simple copy-pasting of all height value within a (circular) brush area, up to seamless copying and blending of brush areas using more advanced algorithms.

As discussed in Section 7.2, brushes can be applied by directly modifying the original area or can be applied indirectly by transparently (re)applying an algorithm to the separately kept original area while using a brushed influence mask. The latter has the advantage of supporting eraser brushes (locally clearing the influence mask) and global scaling and tweaking of the effect at any time. Terrain blending would benefit from this latter approach as it presumably requires iterative tweaking of the exact blend area and other blend parameters.

The simplest type of blend would be mere copy-pasting of the selected source terrain into the destination terrain. One difficulty with this idea would be the resulting seams at the border of the selected area. Unless the height at the source and the destination area match up at the borders of the brush(ed) area, a shift in average height will be noticeable. This is generally not desirable as you

most likely would like to copy features within the brush areas from the source area to the destination area, not create new features (i.e. the sudden change in height). The following subsections discuss different techniques of increasing complexity to blend two heightfields. As with many algorithms discussed before, these techniques were developed as image editing techniques, but can transparently be applied to heightfields as well.

7.4.1 Simple Boundary Feathering

A common technique in image editing is feathering. A soft brush (with a falloff curve towards its edge) is used to blend in the result. A simple *dst' = lerp(dst, src, mask)* (i.e. linear interpolation blend of *src* into *dst* where indicated by *mask*) can be used to calculate the local height value of the blended result. Here, *mask* is a temporary mask field (i.e. a scalar field similar to a heightfield) where the local value determines the blending strength. It is typically zero for all height values outside the brush's radius and is increasing up to one towards the brush's center. This will limit the hardness of the brush's border, but will not completely alleviate the problem, as Figure 7.8 demonstrates for a synthetic example. In that figure, a 'mountain' is created while it might be the designer's intent only to locally replace the square wave with the triangular wave where he or she brushed. The problem here is the large difference in the mean of the source and destination terrain. In this particular case, one could normalize both the source and destination terrain by subtracting their respective mean value before blending them and then add the old mean value of the destination again. This can be seen as separating the terrains into a DC (i.e. zero frequency) component and a non-DC (i.e. all non-zero frequencies) component, blending the source and destination terrain per component using a weighted strength mask and calculate the sum of these blended components. This is a special case of the algorithm discussed next.

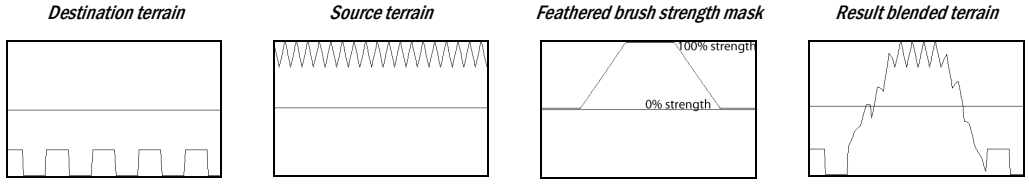


FIGURE 7.8 Terrain heightfield cross section

7.4.2 Multi-resolution Blending

In Section 6.1, Laplacian and Gaussian pyramids are discussed. These two types of pyramids effectively calculate low-pass-filtered and band-limited octaves of an input image, respectively. In [BURT83], an image blending technique is introduced that uses these pyramids to blend the source and destination image differently for different octaves. This is one example of multi-resolution blending.

This algorithm consists of three steps: decomposition, blending the different components and recomposition. Decomposition consists of calculating the Laplacian pyramid of both the source and destination image. Also, the Gaussian pyramid of the mask is calculated. Then, a new Laplacian pyramid is calculated from these three *src, dest* and *mask* pyramids by calculation the independent result image of a *lerp(src, dest, mask)* per pyramid layer. Finally, the image result is recomposed by summing over the different layers of this resulting Laplacian pyramid. Although originally developed for image mosaicing, it can transparently be applied to heightfields. This blending process is demonstrated in Figure 7.9 for the synthetic terrain cross section of Figure 7.8.

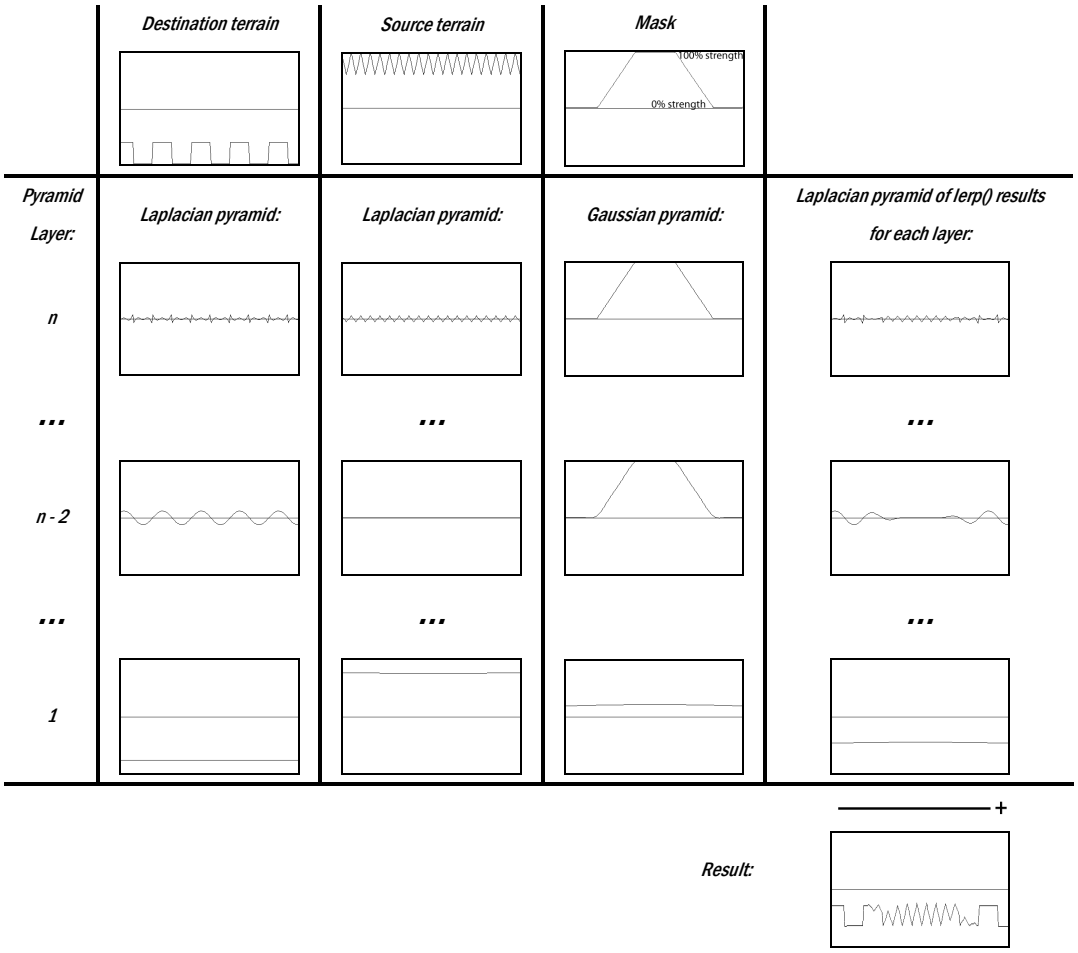


FIGURE 7.9 Multi-resolution blending of a terrain heightfield cross section

This algorithm results in a multi-resolution blend of source and destination where the finest details are interpolated between source and destination over a short distance when a (non-feathered) brush is used. Coarser detail is interpolated over a longer distance. In effect, details will be blended over distances similar to the specific detail size.

This idea can be made more flexible by introducing a scaling factor per layer of the mask pyramid, bound between 0 and 1. Choosing relatively lower scaling factors for the lower octaves would result

in copying less of the lower frequencies of the source image. Likewise, zeroing out the scaling factor for the highest frequencies would leave the higher frequency features of the destination unchanged. See Figure 7.10 for these two scaling examples applied to a more realistic heightfield.

A potential disadvantage of this technique is that the destination heightfield is also adjusted somewhat outside the masked area as the influence mask is spread out for lower resolutions (i.e. lower layers) due to the low-pass filtering. In Figure 7.9 this shows as a change of the mean height. This might or might not be appropriate for different situations. Another approach that only changes the area inside the mask is discussed next.

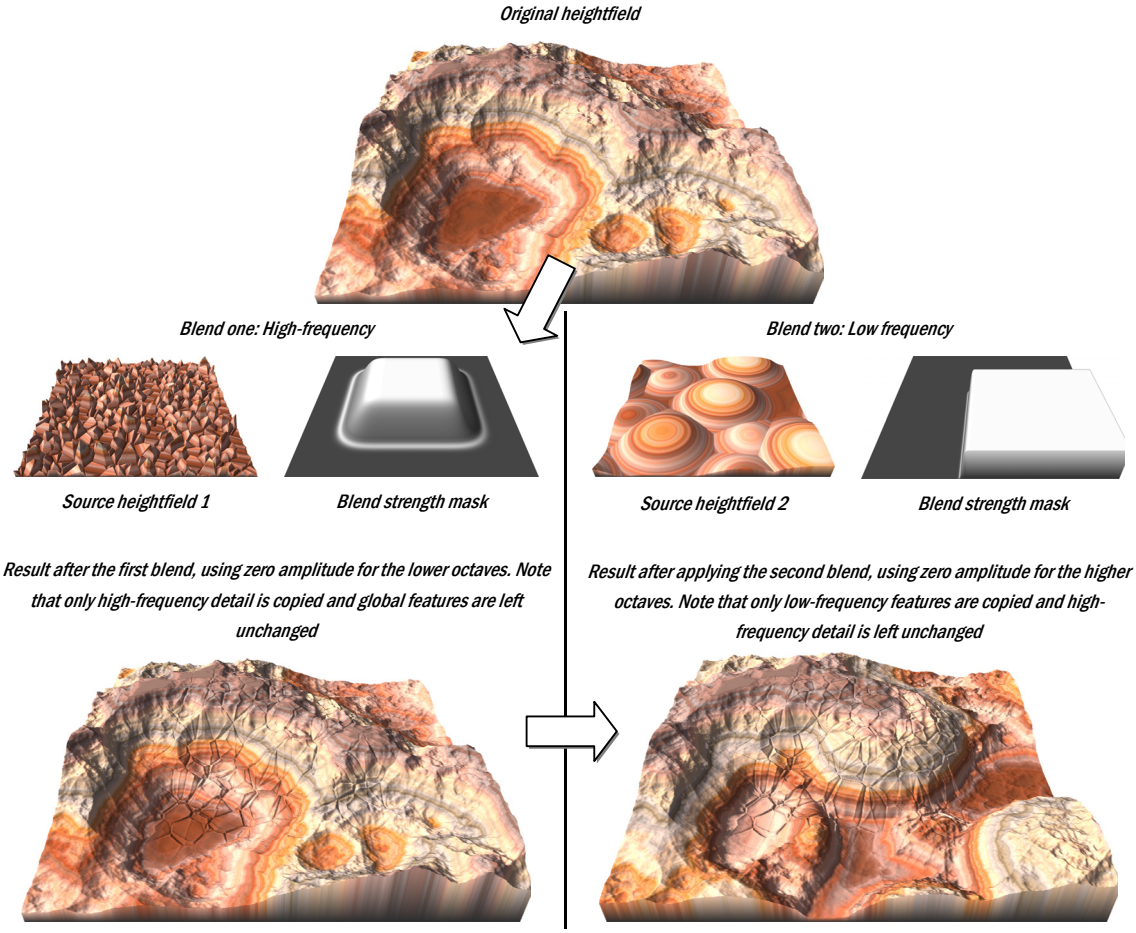


FIGURE 7.10 Example of two differently weighted multi-resolution blending operations applied to a heightfield.

7.4.3 Poisson Editing

The more involved method discussed in this subsection solves a linear system of Poisson-based equations to calculate the best blend of a source area into a destination area. Introduced in [PÉRE03], a more elaborate discussion can be found there on the theory behind it and its different applications to color images. Here, only a short summary of the basic discrete result is given.

The problem of blending source areas into destination areas is most noticeable at the edges of this area. Feathering (i.e. a brush with a falloff curve) helps to hide this, in effect spreading the boundary error over a greater distance. The algorithm discussed in the previous subsection applies this over multiple scales to hide this even further. The method discussed here takes another approach. Instead of spreading the differences at the boundary over a distance explicitly, the destination area inside a (brushed) mask area is calculated by solving a quadratic minimization problem in the images' gradient domain. The destination area outside the masked area is completely left untouched. Given are the pixel values at the mask's boundary edges of the destination area, as well as the source's pixel values inside the mask and at the mask's boundary. For now, a boolean mask is assumed. From these values, a minimization problem is formulated containing a set of quadratic equations. As the minima of quadratic equations can be calculated by the use of their (linear) derivatives, the solution can be found by solving a matrix of linear equations.

Destination image. The white area was cut as it is no longer needed and will be fully replaced by the blend result

Source terrain. Example pattern containing sharp edges through the use of contrasting colors

Result blended terrain. Note that results are a little different than expected from a boolean mask, as the application used to create this image also applied some feathering near the edges

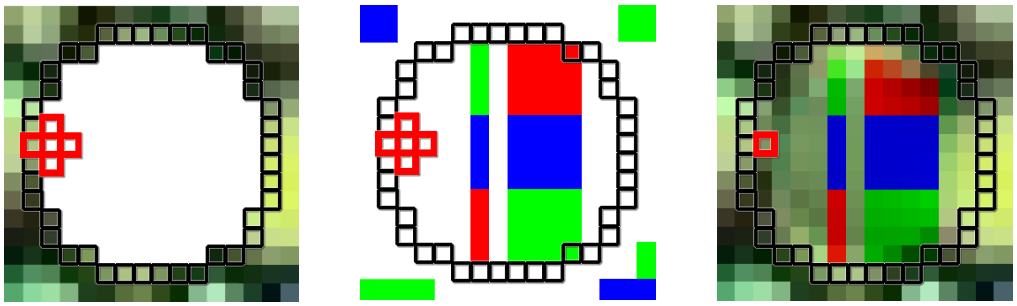


FIGURE 7.11 Poisson blending of two images using a circular mask

The set of equations that needs to be solved can be written as follows:

$$\forall_{p\in\Omega} \quad \sum_{q\in N_p} D_q - |N_p| \cdot D_p = \sum_{q\in N_p} S_q - |N_p| \cdot S_p$$

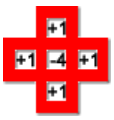


FIGURE 7.12 3x3 approximation of the Laplacian operator

Here, Ω is the set of pixels that are masked and need to be set by the algorithm (white pixels in leftmost image in Figure 7.11). S_p and D_p are pixels from the source image and destination image inside Ω, respectively. N_p are the valid pixels inside the von Neumann neighborhood of p, p itself excluded. Hence, $|N_p|$ is 4, except at any border pixels of S and D. See the red cross surrounding a center pixel p inside the white Ω area in Figure 7.11 for an example for N_p. Of course, all S_p and S_q are known. All D_p-s are unknowns. Likewise, D_q-s are also unknown, except for the neighbors of a p that are at the (outer) border of Ω. Cases of known D_q-s are marked with a black border in Figure 7.11. Both the lhs (i.e. the left-hand side) and the rhs (i.e. the right-hand side) of each equation can be interpreted as a 3x3 approximation of the Laplacian operator. See Figure 7.12. Therefore, this set of equations can

interpreted as matching the local 2nd derivatives of the result (the lhs) with the local 2nd derivatives of the source (the rhs) for all pixels that are part of Ω. The D_q-s that are known (the black bordered squares in Figure 7.11) will bind part of the lhs variables of the equations, resulting in a smooth transition from the boundary inwards when this linear system is solved.

Together, these linear equations form a sparse, banded matrix that is best solved using an iterative solver. Examples of these are Gauss-Seidel solvers with successive overrelaxation or V-cycle multigrid solvers. Both allow a patch of about 256 x 256 pixels (or, heightfield values) to be blended at interactive speeds on today's CPUs. These solvers can also be implemented on the GPU [BOLZ03] to blend even larger patches at reasonable speeds.

7.5 Preliminary Discussion

Many different editing techniques have been introduced and proposed in this chapter. All of these could be offered to the designer by the use of brushes that immediately (should) have effect on the visible landscape. Simple brush strokes, together with user-editable settings and presets, provide a recognizable interface to users of Photoshop and other image editing applications. Offering the possibility to use Photoshop-like effect layers for indirect editing could further enhance the potential.

When compared to the limited heightfield editing functionality offered by most of today's level editing tools, much can be gained by offering (any subset of) the ideas proposed in this chapter. For example, blending terrain allows the reuse of procedural techniques in later stages of the design where medium-sized areas possibly need to be changed. Range and domain mapping might assist designers in creating natural effects in an efficient way. Also, erosion is a valuable tool that allows the creation of realistic features that are otherwise hard to accomplish using procedural techniques and low-level editing. The tools proposed here would all be fairly intuitive as their effect should be directly visible. Also, the types of parameters are intuitive and could be made consistent, except perhaps for the different erosion and blend algorithms. It might be considered to support only one (flexible) type of erosion and blending.

Erosion could be offered as a single tool, applying erosion globally to create a large patch of a certain type of terrain, or be applied only locally as a tweaking tool by, for example, 'brushing' rain that erodes the heightfield underneath it. This would offer the designer a tool that fits nicely somewhere between high-level global procedural generation and low-level local editing. Note that the speed of an erosion tool, or of any brush tool for that matter, is essential, requiring interactive rates to allow the designer to intuitively work with a tool. The results from the erosion algorithm introduced in [BENE02b] (fluvial erosion with water evaporation, Section 7.3.2) are reasonably good as this algorithm has a good balance between quality and speed and allows efficient execution of this algorithm on the GPU. Therefore, it is expected that this algorithm will be the best choice for an

erosion brush. Also, the evaporation will limit the radius of a local rain brush, making it better suited as a local editing tool. If this algorithm would still prove to be too slow to be effective as an interactive tool, the simplifications suggested in [OLSE04] could be tried out and be made optional.

Of the discussed blending tools, multi-resolution blending is probably the most flexible and 'tweakable' algorithm. And it is presumably faster than the Poisson editing technique and easier to implement. Different blending effects can be accomplished by changing the multi-resolution weights of the tool. Offering only this algorithm as a blending tool is expected to be powerful enough for designers. But experiments might be needed to verify this, as few designers previously have had the chance to experiment with blending.

8 Terrain Texture Editing

So far, the focus of the algorithms covered in this report has been on synthesizing and editing heightfields. When a heightfield is to be used in a real-time engine, this heightfield will generally be rendered as a set of polygons. Assigning uniform colors to these polygons will not create very convincing results when rendered. Photorealistic textures can be assigned instead to increase the visual resolution of the material the terrain is made of. Typical textures include mud, snow, dirt, sand, grass and rock. These terrain textures can be created by artists from edited photographs or might even be generated procedurally. Texture creation is not covered here. Instead, this chapter shortly discusses different techniques to locally and/or globally create and edit a mapping of already-made terrain textures to the terrain heightfield.

8.1 Terrain Texturing Methods

Textures are typically applied to triangles during rendering. To triangulate heightfields, the heightfield elements are first used to construct a mesh of quadrilaterals (quads). The vertices of this mesh are spaced regularly when projected on the horizontal plane. The vertical displacement of each vertex is the height value of an associated heightfield grid element. As efficient rendering requires triangles, these quads are subdivided further into two triangles. Three common methods of subdividing quads into triangles are depicted in Figure 8.1, varying in isotropy and control.

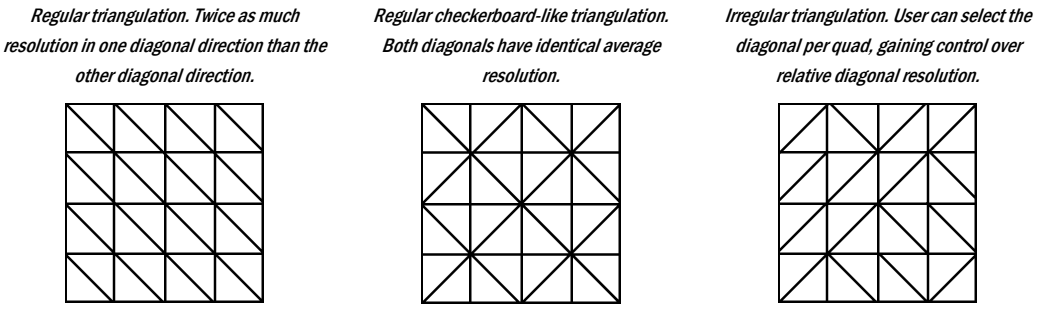


FIGURE 8.1 Different quad mesh triangulation techniques for 16 quadrilaterals.

Different methods can be used to apply texture to heightfield triangles, varying in quality, flexibility and memory requirements. These techniques are covered below.

Global texture

The simplest texturing technique is analogous to the idea of heightfields. A single color image is assigned 1:1 to the whole terrain geometry using a vertical orthographic projection. Obviously, the disadvantage of this technique is memory usage as using resolutions that result in more than only a few pixels per triangle are prohibitively memory intensive. It might suffice for flight simulators, rendering the terrain from a great distance, but looks terrible for applications that show the terrain from only a few meters above ground. Also, creating a global map can be difficult to do by hand.

However, when the used heightfield is actually a realistic model of real existing terrain, an aerial photograph might be used instead. See Figure 8.2.

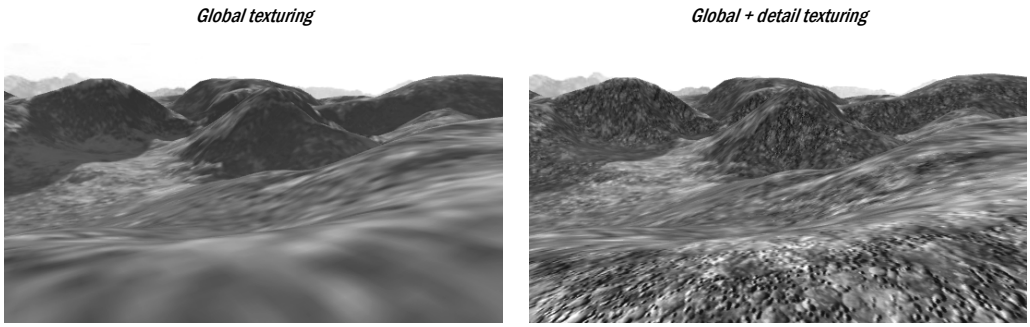


FIGURE 8.2 Global texturing with and without detail texturing. Note the visible lack of detail near the camera in the left image.

Detail texture

A fast and simple improvement of the previous technique is the use of an additional detail texture. A detail texture is a high-resolution texture of a small patch of terrain. This texture is typically tiled (i.e. repeated) at every heightfield quad and blended additively or multiplicatively together with the global texture. This will give the global texture a high-resolution look to it. The disadvantage is that different types of ground materials (represented by the different colors in the global texture) will use the same, globally applied detail texture to improve visual resolution. This could result in, for example, strange looking patches of green grass (from the global texture) with a rock-like look (from the detail texture). See Figure 8.2.

Quad textures

One way of introducing detail that matches the material type (e.g. rock and grass) is to assign a single detail texture from a small, fixed set of detail textures to each heightfield quad. This can be implemented by replacing or extending a global texture to assign a number to each terrain quad, indexing into a detail texture array. Of course, changing the texture per quad will create visible texture seams unless carefully constructed transition textures are placed between adjacent quads of different material types. See Figure 8.3. This technique can be interpreted as a 3D application of classic 2D arcade sprite tiling. Another (or combined) global color texture might be used to blend with the quad texturing to introduce some subtle variance in the color, hence hiding the repetitiousness of the detail textures somewhat. Requiring at least one transition quad between different types of terrain might result in too smooth transitions in some rapid-changing situations. The number of transition textures is quadratic in the number of different terrain types, becoming the limiting factor.

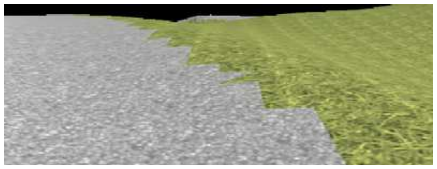


FIGURE 8.3 Quad texturing without transitions. Note the seams between the rock and grass texture. From [DEXT05]

Wang tiling can be considered to be a special case of quad texturing. There, tiles (i.e. a texture per heightfield quad) are selected and assigned from a minimal set of carefully constructed tiles to effectively create an aperiodic tiling pattern [STAM97]. See Figure 8.4. Edges of the tiles in this set

are said to be color coded and need matching edge colors with their neighbors, much like dominos, when laid on the heightfield in order to create seamless a result. A carefully constructed set of tiles, combined with a set of recursive production rules will create a seamless tiling. Although aperiodic tiling greatly reduces the visual repetitiveness of a texture, it isn't clear how to adapt this to use multiple base textures including transitions between these (e.g. grass and rock).

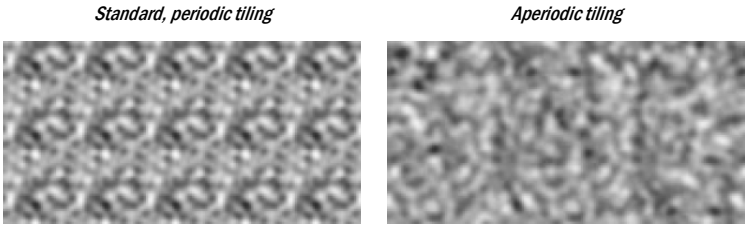


FIGURE 8.4 Example of standard tiling and aperiodic Wang tiling. From [STAM97]

Splatting

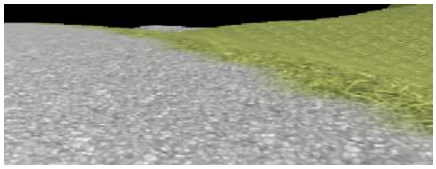


FIGURE 8.5 Splat texturing. Compare to Figure 8.3. From [DEXT05]

Splatting can be seen as an extension to quad texturing, using automatic blending of different textures. Transitions and material blending (e.g. 20% sand and 80% dirt) are done in real time by calculating a weighted average of different material textures. These weights are assigned per vertex, typically using a texture for weight look-ups, and are linearly interpolated between vertices during rendering of the terrain quads [BLOO00] [DEXT05]. See Figure 8.5. Typically, only a few materials are used locally at once because blending too many textures together will result in a muddled appearance. But over larger distances, many other materials might be found. Storing the blending weight for all possible materials per vertex, of which many weights would be zero, would require a lot of memory. Partition techniques can be applied to store only the non-zero weights of textures actually used at different areas of the terrain, resulting in considerable memory conservations. Although the splatting technique can introduce variation through subtle weight perturbations to hide patterns of (identical) texture repetition, Wang tiling could be applied to hide these patterns further. However, its advantage might be outweighed by the increase in difficulty to create a Wang tiling set for each base texture and the increase in algorithm complexity and storage requirements to use Wang tiling.

Procedural techniques

Like the procedural generation of heightfields, textures can be generated and assigned procedurally. Creating textures procedurally could potentially result in infinite detail. However, this would have to be done in real-time, as storing an offline generated texture would only differ in the method of creation, not the representation and mapping. However, real-time generation of complex procedural textures is currently still too compute intensive for most purposes. Also, these procedural techniques would not allow any fine-grained local control over the texturing, similar to procedural techniques discussed in Chapter 4. As processing power increases, more complex

schemes are used in games and other interactive applications. Consequently, a shift from global/detail texturing and quad texturing to splatting has taken place in the last few years. As hardware advances further, high-detail real-time procedural techniques might eventually become the dominant technique. But as this report focuses on the best possibilities for editing terrain on today's hardware, only the de facto splatting is considered in subsequent subsections.

8.2 Splatting Extended

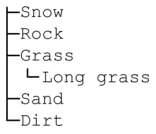


FIGURE 8.6 Example of a user-defined material layer hierarchy.

As explained above, texture splatting uses blend weights to locally blend between textures in real-time. During rendering, the different blend weights are sampled from a global texture containing exactly one sample per heightfield vertex. To aid the designer in designing these local weights, a layered material representation is generally used. These weights then correspond to the opaqueness of a material layer. Typically, the designer is expected to define these materials. Materials typically consist of a texture and a scaling factor. These materials are blended in a pre-designed order during rendering. This ordering is important since blending material *A* with *B* will generally not be identical to blending material *B* with *A*. For example, when the 'higher' layer of the two has 100% opaqueness, the 'lower' layer will be completely covered, independently of the opaqueness setting of this lower layer. An example of a layered set of materials can be found in Figure 8.6. When these specific layers would be used, the grass texture will always be placed on top of the sand texture. Also, any rock texture can only be visible when the weight of 'snow' is locally smaller than 1.0. Some implementations allow a hierarchical parent-child system, where the local opaqueness of child layers is multiplied by the opaqueness of their parents. In effect, a texture assigned for a child layer will only be visible when both parent and child locally have a non-zero weight assigned. Therefore, the 'Long Grass' child in the example above could only be visible where its 'Grass' parent has a non-zero opaqueness.

The weights are typically brushed on the terrain by the designer by selecting a circular brush and a material to paint with. Brushing is intuitive and typically appreciated by designers. This could be extended by constraining the brush using other factors. In natural scenes, the type of visible local surface material depends on many factors including soil type and erosion (e.g. soft sediment or hard rock), temperature, absolute height (height above sea level), local relative height (local valleys generally contain more water and are more sheltered), slope steepness and slope direction, all influencing the local sun, wind and rain conditions [HAMM01]. From these, the local height and slope attributes can directly be calculated for a heightfield. These properties can be used for user-defined brush constraints (e.g. not allowing the snow texture weight to be increased below a certain absolute height). The designer could then select min/max ranges for these height and slope constraints and paint with broader brush strokes while automatically considering the terrain geometry. To prevent these constraints from creating too regular and hard-edged weights, these constraints can be made softer by using a falloff ramp near the ranges' min/max values. Also, local

height and steepness values can be blurred together with values of neighboring quads to create a smoother result. To introduce irregularities, a noise function (see Chapter 5) can be used to locally perturb the selected ranges.

An alternative to using (non-global) geometry-constrained paint brushes would be to enforce these constraints globally. This could be used to generate a user-defined first approximation of the terrain texturing. In some (purely procedural) applications, this is in fact the only option available to the user. Note that applying the constraints globally could undo any previously handcrafted work, similar to procedural heightfield generation. In Section 7.4, a solution was proposed to overcome this problem by allowing procedural results to be blended in, with or without the use of layers. For texturing, a somewhat similar approach could be used. A solution would be to use a double set of layers, the upper half taking precedence over the lower half. Then, the lower half could be assigned procedurally and allow height and slope constraints to be set. The upper half of the layer set is used by the designer to paint on top of the procedurally defined texturing where desired. When the designer would like to make a local change, he could do so by brushing (i.e. increasing the local weight of) one of the layers of the top half. Likewise, undoing any local custom changes could be done by simply erasing any painted weights of the top half. Adjusting and globally applying the procedural settings after local changes have been made is possible as updating the lower half of the layers would not affect the custom painted upper half on top of it. Implementing this directly would double the number of real-time texture lookups. However, the doubled set of material weights, defined by the custom and procedural weight for each of the used textures, can transparently be compiled onto a single set of texture weights as a render preprocess operation without loss of flexibility. In fact, the only difference is editor representation, not renderer implementation.

8.3 Texture Projection

As discussed in Section 3.1, one problem with heightfields is the uniform resolution across the horizontal plane. As a result, steep areas contain less heightfield vertices per area unit because the distances between vertices are increased by vertical differences. Splatting typically renders a complete (blended) texture on each quad. Consequently, textures will be stretched in the steepest direction. This texturing method can be interpreted as an orthographic projection along the vertical axis of a (repeated) texture onto the heightfield.

For arbitrary 3D objects, this problem is normally handled by applying more complex projections or even unwrapping the mesh onto a texture plane, called UV unwrapping. This idea could, in theory, also be used for heightfields. UV unwrapping is time consuming to do by hand. Algorithmically generating optimal unwraps is feasible using, for example, iterative error/energy minimization algorithms. However, these are typically slow and are global, affecting the texturing even far away when a local change is made. Furthermore, texture coordinates are often not stored explicitly in current applications, as these are typically derived directly from the vertex positions

projected on the horizontal plane to save memory. Therefore, automatic UV unwrapping is not very practical for current applications.

A simple and effective alternative would be use a different texture projection direction near very steep terrain, other than the vertical axis. One way of implementing this would be to let the designer assign a single X, Y or Z orthographic projection axis for each of the defined materials. Note that selecting a projection axis would only influence the way texture coordinates are derived from the 3D heightfield vertex information. Therefore, nothing is actually rendered from aside and so occlusion and back faces are never an issue when projecting. The designer can create different materials using the same texture but with a different projection axis. Then, the local projection axis can be chosen freely by brushing with, or procedurally assigning, the most appropriate material. Obviously, using the material of a certain texture that has its projection axis most perpendicular to a quad's surface would cause the least amount of texture stretch. The splatting of the different materials will cause a transitional blend between any neighboring areas that use a different projection, just like any other texture splatting blend. This blending of identical textures using different mappings will not be too noticeable, as terrain textures are already designed to contain as less distinguishable, separable features as possible in an attempt to hide repetitious tiling patterns. The performance penalty is no different than having many different textures applied to a terrain. Smart partitioning into smaller patches of terrain would significantly limit the number of different materials to be blended per quad during rendering, only using more blends near transitional areas.

8.4 Preliminary Discussion

This chapter has given an overview of techniques described in literature and found in practical applications. As computational power and storage capacities increase, more complex render techniques become feasible at real-time frame rates. Currently, texture splatting is the preferred technique as it relieves the designer from explicit creation and assignment of transitions between different types of ground coverage, while limiting the amount of memory and processing power required. Subtle variations are easily added by small changes in weights, possibly combined with a subtle global color texture map. Designers could be enabled to design ground coverage layers using a hierarchical material representation. Height and slope dependent layer parameters could be chosen to procedurally assign material textures, possibly extended with blurring and noise perturbation to create a more varied result. Local modifications could be made to a procedurally generated global material assignment by supporting local brushing with one of the selected materials. These custom changes can be kept separate from the procedural layers by transparently doubling the set of used materials and let the custom changes always take precedence over the procedural assignments. This keeps procedural changes as a result of changed procedural parameters separate from any custom work, allowing for (re)tweaking of these parameters at later stages without destroying any of the handcrafted changes.

Although all of the separate features mentioned above have been implemented in one or more editing applications, most applications only support a subset of these features. However, to support a designer optimally, implementing this full set of features would be very useful. Also, this system can be made more powerful by letting the procedural assignments be dependent on other factors. An example of this would be to have an independent procedural field locally influence the weight of a grass material, possibly combined with already discussed height and slope constraints. This would result in patchy areas of varied amounts of grass. Another example of this would be to have the 'Long grass' layer in Figure 8.6 be influenced by this independent (and possibly otherwise invisible) field instead, creating a complex combination of different grass patches. Even another way of achieving a more varied effect would be to have this field influence (or even decide) the local color of an applied global texture, resulting in a more varied palette of colors. Each of these ideas would result in a more natural, visually complex terrain with the minimum amount of effort. Furthermore, other types of properties and geometry might influence the procedural choice of local ground coverage. For example, grass generally doesn't grow very well in thick forests and on shorelines. So, it makes sense to allow proximity of large amounts of water and large objects (e.g. trees) be used as additional factors in the procedural decision of texturing.

9 Foliage Placement

Both terrain geometry and texture editing has been discussed so far. This chapter covers the last aspect of outdoor terrain discussed in this report: placement of foliage objects (e.g. grass, bushes and trees). In contrast to terrain texturing, terrain foliage objects (and other types of natural objects, like rocks) consist of (textured) 3D geometries, placed on top of the terrain. Foliage geometry creation is not covered in this report. The interested reader is referred to [PRUS90]. Instead, this chapter discusses the effective placement of foliage geometry objects onto the terrain. Please note that placing rocks and stones is not mentioned explicitly in this chapter, as it would suffice to use simplifications of the algorithms discussed below. Hence, support for rock placement could easily and transparently be added.

As virtual foliage consists of 3D geometry, individual objects can be placed into a virtual environment like any other type of geometry. Typical tools used for this would be object importing and translation, rotation and scaling operations. Each object can be placed individually by the designer as he wishes. This might be ideal in some cases that would require exact control over the result. For example, creating a garden with plants placed in some desired pattern, but also, trees that are part of the gameplay in a game and are placed there for a specific purpose. However, creating large patches of grasslands or forests in this way would be very cumbersome.

Once again, procedural techniques can be used to support designers by allowing them to apply foliage on a higher level. Two different techniques of foliage placement are discussed here: L-systems and density evaluation. These two approaches are discussed in the first two subsections. The main disadvantage of both basic techniques and a solution to this disadvantage are discussed in Section 9.3. A preliminary discussion is given in Section 9.4.

9.1 L-Systems

L-systems [PRUS90] are most known for their use in procedural generation of plant geometry. L-systems apply rewriting operators (production rules) to an initial string (the axiom) using a finite symbol alphabet. Complex, natural structures can emerge when this string is interpreted after string rewriting has been completed. For plant generation, symbols like branch commands and radius/length modifiers are used. The applied rewriting rules are designed to result in additional branching after each completed iteration to simulate growth, creating natural virtual plants when the resulting symbol string is interpreted as a geometry construction sequence. Strict L-systems lack context sensitivity and the support for external function evaluation. When extended with these features, L-systems have shown to be remarkably successful in simulating all sort of growth. For example, in [PARI01], L-systems have been used to generate whole cities. In [DEUS98] and [LANE02], the spreading, growth and death of foliage objects is simulated using L-systems. These rules effectively enforce a natural balance between foliage over many iterations. Also, by incorporating nearest neighbor distance functions into the rules, more complex ecological effects can be

simulated. Good result can be obtained through L-systems. However, this approach is rather compute intensive [DIET06] and hard to design. But this is not the only approach capable of naturally placing foliage.

9.2 Density Evaluation

Procedural techniques discussed in previous chapters all worked on images and, therefore, also on heightfields. In contrast, foliage object placement requires placing individual objects, not field construction. However, placement of individual objects can be accomplished by sampling random (local) positions using a (globally defined) probability mass function. For example, creating a forest using such a tool would comprise of brushing the global outline of the forest into the probability field. Then, the probability distribution field is used to take random samples which are then interpreted as positions of individual foliage objects [LANE02]. By interpreting a procedural field not as a heightfield but as a probability function, a link to the earlier procedural algorithms is established. Interpreted as a field, the discussion in Section 8.2 on procedurally selected ground coverage types is directly applicable. For example, geological properties (e.g. local height and slope) can be used as influences on such a 'probability field'. Furthermore, it can be blended with an independent, procedurally generated field to introduce variance. Also, the discussions and suggestions on custom manipulation in Section 7.2 are directly applicable. For example, a designer could brush probabilities, either directly on the procedurally generated result, or indirectly through the use of a layered representation. In the layered representation, a separately kept density field (i.e. a layer) could be combined with the procedural result when required during sampling, while offering a clean separation between custom and procedural placement influences.

To efficiently calculate a position (X, Y) to place a piece of foliage at using a density field, a discrete 2D joint mass density field P, which is essentially a matrix, can be sampled as follows:

1. Calculate the marginal probability $P_x(x \leq X)$ from $P(x, y)$ for each column X in the matrix P
2. Generate a uniformly distributed random number $r_x \in [0,1]$ and find X such that $P_x(x \leq X)$ is closest to r_x
3. Calculate the conditional probability function $P(y \leq Y \mid X)$. Note that Y denotes a row of P
4. Generate a uniformly distributed random number $r_y \in [0,1]$ and find Y such that $P(y \leq Y \mid X)$ is closest to r_y

Note that these X and Y components form an integer coordinate in the horizontal plane. This algorithm can easily be adapted to interpolate between the two Xs and Ys closest to r_x and r_y, respectively, to calculate a continuous position instead of integer indices. And, of course, this two-dimensional coordinate in the horizontal plane can be transformed into a three-dimensional world coordinate by adding a vertical component, looked up from the heightfield.

Another technique to sample *P(x, y)* is called dithering. Although normally used to reduce the repetitive error of quantized digital signals, a standard (Floyd-Steinberg) dither technique can also be used to create a pattern of zeros and ones from *P* [LANE02]. Then, all ones would indicate that an object should be placed there. This algorithm traverses P in raster scan order and propagates any quantization error among its neighbors that are not yet processed, using a fixed set of quantization error distribution weights. As *P* is effectively transformed to a binary matrix, the positions of the objects (i.e. the indices of all 1s in the binary matrix) are all integers. Additional small random perturbations can be used to makes these positions continuous.

Some of the object positions calculated using one of the two algorithms presented above might lie much closer to each other than others. However, natural foliage growth is dependent on sufficient amounts of sun, water and nourishment, preferring a more even distribution. Consequently, spreading the positions of foliage objects more evenly might improve the realism of intended result (e.g. a forest). As suggested in [DEUS98], this might be achieved by iteratively moving each calculated position slightly towards the center of its Voronoi polygon.

9.3 Density Evaluation Extended

The disadvantages of both L-systems and the density evaluation method as described in the previous sections are similar to those of the techniques discussed in Chapter 4. The procedural result can be recalculated using other parameters and can even be influenced locally for L-systems and density evaluation by changing the context sensitive functions or brushing changes to a probability mass function, respectively. However, making local manual modifications to the positions of (some of) the individual foliage objects would have its difficulties. Although changing foliage object locations after a procedural algorithm has finished might be possible, any subsequent calls of the procedural algorithm will recalculate all positions and thus completely override these manual changes. Another disadvantage of these techniques is the difficulty of specifying more complex ecological dependencies and constraints between foliage objects.

A workaround for this would be to use two separate and independent layers of foliage objects, similar to the layered texturing approach discussed in Section 8.2. Foliage could be defined procedurally in one (bottom) layer, while the other (top) layer would contain all foliage objects that are placed manually by the designer. Obviously, this still wouldn't solve the problem of manually editing foliage placed by the procedural algorithms directly. However, the probability function used for the procedural placement can locally be brushed to zero probability for the density evaluation approach in order to clear all procedural foliage objects in a certain area after recalculation. Likewise, the context sensitive L-systems functions could be adapted to leave a designated area clear from foliage when (re)evaluated. Then, this area could be filled with manually placed objects, offering the maximum of control to the designer.

Another, more elegant, solution to this problem is presented in [LANE02] by extending the probability density approach. Just like the original probability density approach it uses a joint probability mass field that can be procedurally determined and influenced through custom brushing. However, instead of an initial phase of (influenced) density field calculation, followed by the calculation of all foliage object positions, the density field is influenced by all already placed foliage and updated for each new object. In effect, foliage objects are placed one by one, each influencing the probability distribution used for the next random sample taken. This way, the procedural algorithm can be used to add new objects to an already (partially) filled terrain where desired, not requiring a complete recalculation of the positions of all placed objects. Consequently, manually placed objects can safely and transparently be mixed with procedurally placed objects and can be edited afterwards on the individual object level where desired.

Also, brushing to affect the density function can be replaced by or complemented with direct object 'brushing', where only objects inside the current area under the brush tool will be affected. Different tool settings could result in adding, deleting or replacing these objects on request at a given change speed (instant or some number of objects per second). The brush tool could, for example, also be complemented with earlier discussed constraints like allowable height and slope steepness ranges. Again, feathering and noise perturbation could help to make transitions between different (constrained) areas more natural.

This extension allows (and needs) the probability mass field to be influenced by each of the individual foliage objects. For this, a 2D modification kernel is applied for each object to modify the density field. Because the density field represents a joint probability mass function, the sum of all elements should be kept normalized to 1 before and after each update. In nature, one is likely to observe local clusters of a specific plant species. See Figure 9.1. This is partly the result of species-specific topographic preference (e.g. soil, groundwater level, height, slope steepness and direction). This effect could already be achieved by letting local values of the terrain elevation and slope steepness influence the procedural density field. Alternatively, this could be achieved by setting direct constraints (e.g. height and slope ranges) on a foliage 'painting' brush. Another factor in typical vegetation clustering is the way many species of plants reproduce. For example, some plant species drop seeds that are likely to fall near their parent plants, while other species propagate by runners. This ecological effect can be simulated by choosing a suitable shape for the kernel when an object is placed. See Figure 9.2 and 9.3. The third and fourth kernel in Figure 9.2 will have a prohibitive (negative) influence on the density function at very close range. However, a promotional (positive) influence is added to the density function at an ideal distance for child plants. By scaling the radius and amplitude of these kernels, the preference of the plant species can be modified.

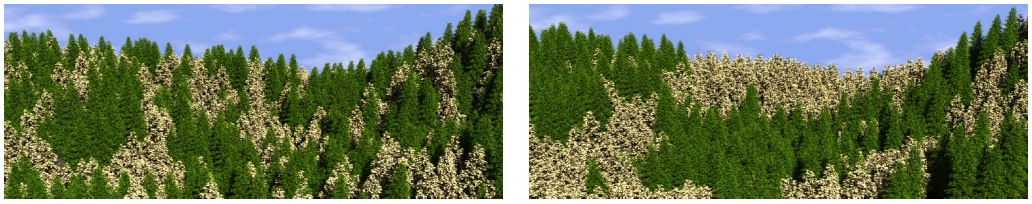


FIGURE 9.1 Random (left) and ecologically motivated (right) placing of trees. From [LANE02]

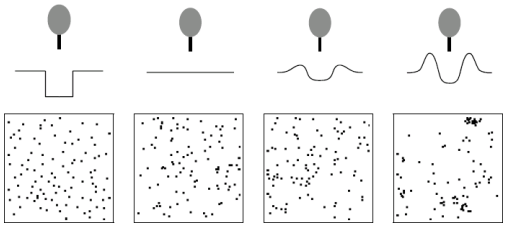


FIGURE 9.2 Four different types of kernels. Kernel effects from left to right: prohibit close placement, random, weak and strong clustering preference. From [LANE02]

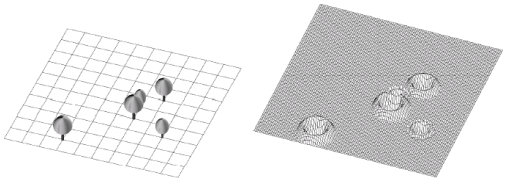


FIGURE 9.3 Tree placement and its probability density function. The kernel used promotes clustering at an ideal distance. From [LANE02]

The above only considers ecological placement of one type of plant species (i.e. foliage object families). When different types of foliage need to be placed in the same area, this idea can be extended naturally to create a density function for each type of foliage used and apply a different kernel for each species-species pair to model interdependencies between species. See Figure 9.4. Note the local interspecies' prohibitive kernel and intraspecies' clustering kernel.

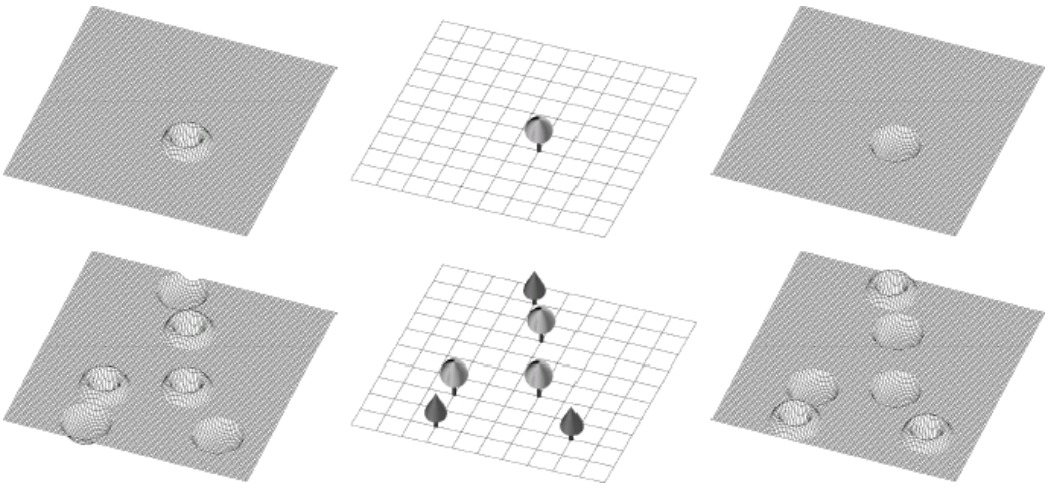


FIGURE 9.4 Dependencies among and between species modeled through the application of different kernels on a species' density function. From left to right after one and six objects have been placed, respectively: resulting density function for (the lighter) species one, terrain containing placed tree objects for species one and two, resulting density function for (the darker) species two. From [LANE02]

9.4 Preliminary Discussion

Two different approaches have been discussed. Extended L-systems can be used to model reproduction, growth and death of individual objects in an ecosystem. Terrain, intraspecies and interspecies dependencies can be modeled by incorporating these dependencies into the production rules. However, the resulting population is emergent from the interactions between these rules and can, therefore, be hard to design. The placement of foliage objects would consist of calculating a produced string and then calculating all object positions from this string at once. Integrating feedback of manually or procedurally placed foliage at an earlier stage into the calculation of new positions is therefore complex and difficult to support. The second approach has the same problem in its basic density evaluation form. However, when extended with a feedback loop by making subsequent changes to the probability density field for each foliage object found or added, foliage can be added transparently by subsequently adding single objects into an area that was either initially empty or contained earlier placed objects. Ecological dependencies can be modeled as direct density field influences (e.g. height and slope constrains [HAMM01]) or as intra- and interspecies kernel pairs [LANE02]. Brushing foliage only inside a certain brush region is easily supported by making all probabilities of the density function zero for all areas outside the area currently covered by the brush. In fact, the density function only needs to be evaluated for the area currently covered by the brush, saving significant calculation time. Consequently, the designer will be able to brush foliage at interactively speeds. Also, growth of stronger individuals and death of weaker individual plants can easily be simulated by scaling up individual plants inside the brush-covered area and by removing individuals that are overpowered (e.g. standing too much in the shade of larger individuals) [BENE02a].

This chapter has been concerned with the procedural placement of foliage. The scale and rotation of the foliage objects has not been covered explicitly. It is expected that taking simple random samples for these two properties using a user selectable distribution would suffice. These distribution settings could be offered to the designer as customizable brush properties, stored as presets or sampled from a selected area. As stated in the introduction of this chapter, other types of natural objects that can be found on terrain (e.g. rocks) often have less complex intra- and interdependencies and, consequently, can be placed with a foliage placing tool with many ecological dependencies disabled.

10 Current Applications

The three major topics covered in this report have been heightfield synthesis and editing, terrain texture assignment and foliage placement. In this chapter, a few different applications that are currently available to designers are shortly reviewed for their support in these areas. This is by no means a complete list of available software. But it does give the reader an idea of the types of applications that are currently available for these purposes, including their typical merits and drawbacks.

Terragen (PlanetSide) *http://www.planetside.co.uk*

Terragen offers a non-real-time heightfield landscape synthesis and rendering system. Its built-in ray tracer is capable of creating very realistic images, including realistic lighting, atmospheric effects, clouds, water reflection and terrain shadowing. Local terrain editing is not supported. So heightfields are either created externally and imported or are completely procedurally synthesized. Heightfield synthesis includes noise synthesis, range mapping and erosion, provided to the user as a limited set of parameterized selectable options. Texturing is supported through texture splatting and is completely procedurally assigned, similarly to the hierarchical representation discussed in Section 8.2. Local texture editing is not supported. Vegetation or other objects are also not supported. The created heightfields and global textures can be exported to be used in other applications (e.g. a game engine or generic 3D editing application capable of placing and rendering objects). Although the heightfields synthesized with Terragen look good, the number of different types of natural terrain that can be created with it is somewhat limited.

World Machine (Stephen Schmitt) *http://www.world-machine.com*

Like Terragen, World Machine is a heightfield synthesis application. However, its main focus is flexibility to create these terrains. Simple real-time 2D and 3D rendering is supported, but this feature is by far not as impressive as Terragen's (non-real-time) renderer. The user can design terrain by placing and connecting heightfield creation, blending and transformation nodes in a flow graph, supporting many synthesis techniques discussed in this report. The image on the cover and many other images in this report have been made with World Machine, indicating its flexibility. A height-based texturing color scheme can be chosen from a limited number of presets. Foliage is not supported. Local editing (e.g. brushing) is also not possible. However, the node-based representation does support (imported or procedurally generated) masks to where procedural modifications should be limited to. Created heightfields can be exported to different formats. Proficient users are able to create various types of natural landscapes with it, but it generally requires much experience and tweaking to do so.

Terragen 2 (PlanetSide) *http://www.planetside.co.uk*

This new version of Terragen is currently still under development. Like the first Terragen, procedural synthesis and rendering are its main focus. Terragen 2 will be extended to allow overhangs. Automatic placement of imported rocks and vegetative models is supported. Foliage placement, texturing, heightfield synthesis and rendering options are represented in a powerful flow-graph system, allowing the user to couple function nodes as desired. Like World Machine, local editing is not supported but can be approximated through the use of node masks. Currently, only a technology preview application is available. Due to the flexibility of this system, synthesis and rendering are relatively slow, although this might be improved in the release version. The actual release date has not yet been announced.

CryENGINE Sandbox 1 & 2 (Crytek) *http://www.crytek.com*

Official WYSIWYG level editors for the Crytek game engines, used for Farcry and the upcoming Crysis game. It offers an impressive set of tools to aid the level designer. It allows heightfield loading and simple procedural generation, Local editing is supported through the use of brushes. However, only the simple brushes discussed in Section 7.1 are available. Hence, no terrain blending tools are offered. Extensive terrain texturing is supported, similar to the layered representation discussed in Section 8.2, including choosing between X, Y and Z projections. Textures can be assigned both manually and procedurally but use the same set of materials. Hence, reapplying a procedural texture assignment at a later stage would overwrite all custom texture modifications. Foliage brushes are well supported, allowing both manual and procedural placement of (imported) individual foliage objects. Sandbox 2 has more advanced features in texturing and placing foliage than the original Sandbox. Both versions offer an easy and intuitive user interface.

UnrealEd 3 (Epic) *http://www.unrealtechnology.com*

Official WYSIWYG level editor for the Unreal Engine 3 game engine and used for Gears of War. Fully integrated level design tool that supports heightfield importing, but offers no form of heightfield synthesis itself. Editing of heightfield is only supported through the basic editing brushes discussed in Section 7.1. Heightfield blending is not supported at all. Texturing splatting is supported through the layered representation and allows a separation of procedurally assigned (base) layers and (overriding) custom layers that can be brushed manually. Procedural foliage placement is (currently) not supported. The user interface of the editor is somewhat hard to use efficiently as it constantly requires manual settings to be set.

From this summary of typical tools, it is clear that there is much potential for improvement. Although some game level editors offer some form of procedural heightfield synthesis, the tools available to designers can roughly be divided into two categories:

- Low-level level editing applications, supporting simple terrain editing tools, while offering little or no support for procedural techniques.

- Procedural landscape generators, capable of synthesizing and previewing terrain to create new images or export resulting heightfields for further use in other applications. These applications generally offer no tools at all to edit terrain locally.

11 Conclusions

This report gives an overview of techniques found in literature and in practice concerning synthesizing and editing virtual outdoor terrain. After an introduction into the field was given, the growing problem of the labor intensity of manually designing outdoor environments was described, focused specifically on computer games. A short overview was given of the applications that are typically available to today's designers, demonstrating the currently limited and fragmented support in the area of efficient terrain design. This report has covered three different areas of terrain design in depth:

- **Heightfields**. Different procedural heightfield synthesis algorithms, synthesis by example algorithms and heightfield editing tools were discussed in Chapter 4 through 7.

- **Texturing**. Different representations of terrain texturing, manual editing techniques and procedural texture assignment techniques were covered in Chapter 8.

- **Foliage**. Different ways of efficiently and realistically adding foliage to terrain, by manually and/or procedurally placing foliage objects, are discussed in Chapter 9.

As both novice and advanced level designers would agree, today's level design tools have many shortcomings. One area of improvement would be the integration of different tools into one coherent and intuitive interface. In contrast, current applications as a whole do cover the larger part of the algorithms, techniques and tools discussed in this report, but seldom offer a substantial subset of the covered topics within one application. To maximize the efficiency of a designer's workflow, a broad range of intuitive and consistent tools should be offered within a single framework.

As the design process of terrain is a creative art that often requires many iterations to be made, it is essential to support this paradigm and provide tools that preferably work at interactive or even real-time rates. For simple, data-intensive tasks, the graphics processing unit (GPU) of today's PCs can be used to quickly process vast amounts of data. When tool calculations are (partly) transferred from the CPU to the GPU, a speedup of somewhere between a half and almost two magnitudes is expected, depending on the task at hand. Furthermore, many alternative approaches for potential tools are surveyed in this report. This allows the implementation of tools to be chosen based on the tradeoffs between accuracy, complexity, scalability and speed of the different approaches. The preliminary discussions throughout this report offer practical insights and comparisons for each of the covered topics, respectively.

As this report shows, ideas and techniques from other fields could be reused and integrated into a user-friendly interface to provide the user with more powerful tools of proven quality. For example, techniques described in the chapters on heightfields by example and heightfield blending (Chapter

6 and 7) were originally introduced and used for image manipulation. Powerful interactive blending tools are expected to aid designers in reusing procedural techniques in later stages of partial redesign, iterative design or simply in the process of tweaking. Thus, heightfield design would greatly benefit from these techniques, yet no serious application exists that incorporates these features.

Tool flexibility is another area that could be improved. For example, global erosion and domain perturbation have been part of many procedural generation applications as a step in the global generation procedure, yet these techniques have never been offered as editing tools. When these techniques would also be offered as brushes, they would efficiently aid the designer in the task of creating custom, natural-looking areas. Also, tools could be applied only to one of a multitude of layers or nodes as another means to increase the flexibility of the workflow and speed up the process of experimentation and tweaking. When the designer is allowed to subdivide different areas, stages, experiments or detail levels into separable layers or nodes, these layers or nodes could be reedited separately and recombined on the fly. Furthermore, when the recombination feature supports complex operations (e.g. perturbation using a second input field or multi-resolution blending two heightfields) to be chosen by the user, a new and powerful range of editing possibilities becomes available.

Offering both local editing tools and more global generation tools to design heightfields enables the designer to choose the level of control that is most appropriate for the current task. Using global procedural tools provide a quick way to create landscapes with little effort that look very natural. On the other hand, local editing tools require more effort from the designer but offer the fine control that is needed to locally tweak an area. These different levels of editing are highly complementary and support the designer at different terrain scales and editing tasks. This is true for heightfield editing, but also for terrain texturing and foliage placement. The discussed terrain texturing and foliage placement techniques potentially offer a high-level, global procedural assignment of ground coverage and vegetation that can be tweaked at a medium level of control through the use of influencing brushes or at the maximum amount of control through local editing, all working interchangeably while considering geometrical and ecological constraints.

In short, natural terrain creation is a field that still offers much potential for improvement. Even though the game industry is rapidly evolving and otherwise uses cutting-edge technology, the tools that are available for terrain editing only improve at a relatively slow pace. This report has exposed many of the problems with these tools, but more importantly, it has provided valuable insights on how to potentially improve these tools in order to aid the level designer in new and promising ways.

Bibliography

[ADEL84] E.H. ADELSON, C.H. ANDERSON, J.R. BERGEN, P.J. BURT, AND J.M. OGDEN. Pyramid methods in image processing. In *RCA Engineer*, vol. 29, no. 6, 33-41. 1984.

[ASHI01] M. ASHIKHMIN. Synthesizing natural textures. In *Proceedings of the 2001 Symposium on interactive 3D Graphics* SI3D '01. ACM Press, New York, NY, 217-226. 2001.

[BELH05] F. BELHADJ, P. AUDIBERT. Modeling landscapes with ridges and rivers. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology '05*. ACM Press, New York, NY, 151-154. 2005.

[BENE01a] B. BENEŠ, R. FORSBACH. Layered Data Representation for Visual Simulation of Terrain Erosion. In *Proceedings of the 17th Spring Conference on Computer Graphics (April 25 - 28, 2001)*. IEEE Computer Society, Washington, DC, 80. 2001.

[BENE01b] B. BENEŠ, R. FORSBACH. Parallel implementation of terrain erosion applied to the surface of Mars. In *Proceedings of the 1st international Conference on Computer Graphics, Virtual Reality and Visualisation*. AFRIGRAPH '01. ACM Press, New York, NY, 53-57. 2001.

[BENE02a] B. BENEŠ, A Stable Modeling of Large Plant Ecosystems. In *Proceedings of the International Conference on Computer Vision and Graphics*, 94–101. Association for Image Processing. 2002.

[BENE02b] B. BENEŠ, R. FORSBACH. Visual Simulation of Hydraulic Erosion. In *Journal of WSCG 2002*, 10. 2002.

[BENE06] B. BENEŠ, V. TĚŠINSKY, J. HORNYŠ, S.K. BHATIA. Hydraulic Erosion. Computer Animation and Virtual Worlds, 16, 1-10. 2006.

[BLOO00] C. BLOOM. Terrain Texture Compositing by Blending in the Frame-Buffer. 2000. *http://www.cbloom.com/3d/techdocs/splatting.txt*.

[BOLZ03] J. BOLZ, I. FARMER, E. GRINSPUN, P. SCHRÖDER. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In *ACM Transactions on Graphics*, vol. 22 , no. 3. Jul. 2003.

[BONE97] J.S. DE BONET. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Proceedings of the SIGGRAPH '97*. ACM Press/Addison-Wesley Publishing Co., New York, NY, 361-368. 1997.

[BURT83] P.J. BURT, E.H. ADELSON. A multiresolution spline with application to image mosaics. In *ACM Transactions on Graphics (TOG)*, vol. 2, no. 4, 217-236. Oct. 1983.

[COOL69] J. COOLEY, P. LEWIS, P. WELCH. The finite Fourier transform. In *IEEE Transactions on Audio and Electroacoustics, v*ol.17, no. 2, 77-85. Jun. 1969.

[COOK05] R.L. COOK, T. DEROSE. Wavelet noise. In *ACM SIGGRAPH 2005 Papers*. J. Marks, Ed. ACM Press, New York, NY, 803-811. 2005.

[CHIB98] N.CHIBA, K.MURAOKA, K.FUJITA. An Erosion Model Based on Velocity Fields for the Visual Simulation of Mountain Scenery. In *The Journal of Visualization and Computer Animation*, vol. 9, no. 1, 185-194. 1998.

[DEUS98] O. DEUSSEN, P. HANRAHAN, B. LINTERMANN, R. MĚCH, M. PHARR, P. PRUSINKIEWICZ. Realistic modeling and rendering of plant ecosystems. In *Proceedings of* SIGGRAPH '98. ACM Press, New York, NY, 275-286. 1998.

[DEXT05] J. DEXTER. Texturing Heightmaps. GameDev.net. 2005. *http://www.gamedev.net/reference/articles/article2246.asp*

[DIET06] A. DIETRICH, G. MARMITT, P. SLUSALLEK. Terrain Guided Multi-Level Instancing of Highly Complex Plant Populations. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 169-176. Salt Lake City, USA. Sep. 2006.

[DIXO94] A.R. DIXON, G.H. KIRBY. Data Structures for Artificial Terrain Generation. In *Computer Graphics Forum*, vol. 13, no. 1, 73-48. 1994.

[EBER03] D.S. EBERT, F.K. MUSGRAVE, D. PEACHEY, K. PERLIN, S. WORLEY. Texturing & Modeling: A Procedural Approach. Third Edition. The Morgan Kaufmann Series in Computer Graphics. 2003.

[EFRO99] A.A. EFROS, T.K. LEUNG. Texture Synthesis by Non-Parametric Sampling. In *Proceedings of the international Conference on Computer Vision (ICCV), vol. 2*. IEEE Computer Society, Washington, DC, 1033. 1999.

[ELIA01] H. ELIAS. Spherical Landscapes. 2001. *http://freespace.virgin.net/hugo.elias/models/m_landsp.htm*

[FOUR82] A. FOURNIER, D. FUSSELL, L. CARPENTER. Computer Rendering of Stochastic Models. In *Communications of the ACM*, vol. 25, no. 6, 371-384, Jun 1982.

[GAMI01] M.N. GAMITO. F.K. MUSGRAVE. Procedural Landscapes with Overhangs. 10th Portuguese Computer Graphics Meeting. Lisbon. 2001.

[HAMM01] J. HAMMES. Modeling of Ecosystems as a Data Source for Real-Time Terrain Rendering. In *Proceedings of the First international Symposium on Digital Earth*. C. Y. Westort, Ed. Lecture Notes. In *Computer Science*, vol. 2181. Springer-Verlag, London, 98-111. 2001.

[HEEG95] D.J. HEEGER, J.R. BERGEN. Pyramid-based texture analysis/synthesis. In *Proceedings of the SIGGRAPH '95*. S. G, Mair and R. Cook, Eds., ACM Press, New York, NY, 229-238. 1995.

[KELL88] A.D. KELLEY, M.C. MALIN, G.M. NIELSON. Terrain simulation using a model of stream erosion. In *Proceedings of the SIGGRAPH '88,* R. J. Beach, Ed., ACM Press, New York, NY, 263-268. 1988.

[KRTE01] R. KRTEN. Generating Realistic Terrain. In *Dr. Dobb's Journal: Software Tools for the Professional Programmer.* Jul. 2001.

[LANE02] B. LANE, P. PRUSINKIEWICZ. Generating spatial distributions for multilevel models of plant communities. In *Proceedings of Graphics Interface 2002*, 69-80. May 2002.

[LEFE05] S. LEFEBVRE, H. HOPPE. Parallel controllable texture synthesis. In *ACM SIGGRAPH 2005 Papers*. J. Marks, Ed. ACM Press, New York, NY, 777-786. 2005.

[LEWI87] J.P. LEWIS. Generalized stochastic subdivision. *ACM Transactions on Graphics (TOG), v*ol. 6, no. 3, 167-190. Jul. 1987.

[LEWI89] J.P. LEWIS. Algorithms for solid noise synthesis. In *Proceedings of* SIGGRAPH '89. ACM Press, New York, NY, 263-270. 1989.

[LEWI90] J. P. LEWIS, Is the Fractal Model Appropriate for Terrain? Disney Secret Lab, 1990. 3100 Thornton Ave., Burbank CA 91506 USA. 1990.

[MAND82] B.B. MANDELBROT. The Fractal Geometry of Nature, New York, W. H. Freeman and Co. 1982.

[MAND88] B.B. MANDELBROT. Fractal landscapes without creases and with rivers. In *the Science of Fractal Images*, H. Peitgen, D. Saupe, Eds., Springer-Verlag, New York, NY, 243-260. 1988.

[MILL86] G.S. MILLER. The definition and rendering of terrain maps. In *Proceedings of the SIGGRAPH '86*, D. C. Evans and R. J. Athay, Eds., ACM Press, New York, NY, 39-48. 1986.

[MUSG89] F.K. MUSGRAVE, C.E. KOLB, R.S. MACE. The synthesis and rendering of eroded fractal terrains. In *Proceedings of the SIGGRAPH '89*. ACM Press, New York, NY, 41-50. 1989.

[MUSG93] F.K. MUSGRAVE. Methods for Realistic Landscape Imaging. Doctoral Thesis. Yale University. 1993.

[NEAL03] A. NEALEN, M. ALEXA. Hybrid texture synthesis. In *Proceedings of the 14th Eurographics Workshop on Rendering*. ACM International Conference Proceeding Series, vol. 44. Eurographics Association, Aire-la-Ville, Switzerland, 97-105. 2003.

[OLSE04] Realtime Procedural Terrain Generation; Realtime Synthesis of Eroded Fractal Terrain for Use in Computer Games. Jacob Olsen, Department of Mathematics And Computer Science (IMADA). University of Southern Denmark. Oct. 2004.

[PARI01] Y.I. PARISH, P. MÜLLER. Procedural modeling of cities. In *Proceedings of the SIGGRAPH '01*. ACM Press, New York, NY, 301-308. 2001.

[PÉRE03] P. PÉREZ, M. GANGNET, A. BLAKE. Poisson image editing. In *ACM SIGGRAPH 2003 Papers*. ACM Press, New York, NY, 313-318. 2003.

[PERL85] K. PERLIN. An image synthesizer. In *ACM SIGGRAPH 1985 Proceedings*. ACM Press, New York, NY, 287-296. 1985.

[PERL89] K. PERLIN, E.M. HOFFERT. Hypertexture. In *Proceedings of the SIGGRAPH '89*. ACM Press, New York, NY, 253-262. 1989.

[PERL02] K. PERLIN. Improving noise. In *Proceedings of SIGGRAPH '02*. ACM Press, New York, NY, 681-682. 2002.

[PERL04] K. PERLIN. Implementing Improved Perlin Noise. In *GPU Gems*. R. Fernando, Ed., Addison Wesley, 73-85. 2004.

[PRUS90] P. PRUSINKIEWICZ, A. LINDENMAYER. The Algorithmic Beauty of Plants. Published by Springer-Verlag New York, Inc. 1990.

[SHAN00] J. SHANKEL. Fractal Terrain Generation. In *Game Programming Gems, M. Deloura, Ed.,* Charles River Media, Inc. 499-511. 2000.

[STAM97] J. STAM, Aperiodic texture mapping. Tech. rep., R046. European Research Consortium for Informatics and Mathematics (ERCIM). 1997.

[TATA05] N. TATARCHUK. Richer Worlds for Next Gen Games. Game Developers Conference Europe '05. ATI Research. London, England. 2005.

[VOSS85] R.F. VOSS. Random Fractal Forgeries, in *Fundamental Algorithms for Computer Graphics,* R.A. Earnshaw, Ed., Springer-Verlag, Berlin. 1985.

[VOSS89] R.F. VOSS. Random fractals: self-affinity in noise, music, mountains, and clouds. In *Phyica. D* 38, 1-3 (Sep. 1989), 362-371. 1989.

[WEI00] L. WEI, M. LEVOY. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of SIGGRAPH 2000*. ACM Press, New York, NY, 479-488. 2000.

[WORL96] S. WORLEY. A cellular texture basis function. In *Proceedings of the SIGGRAPH '96*. ACM Press, New York, NY, 291-294. 1996.

[ZELI02] S. ZELINKA, M. GARLAND. Towards real-time texture synthesis with the jump map. In *Proceedings of the 13th Eurographics Workshop on Rendering*, ACM International Conference Proceeding Series, vol. 28. Eurographics Association, Aire-la-Ville, Switzerland, 99-104. 2002.